```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                include/ansi.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

00000    /* The <ansi.h> header attempts to decide whether the compiler has enough
00001     * conformance to Standard C for Minix to take advantage of.  If so, the
00002     * symbol _ANSI is defined (as 31415).  Otherwise _ANSI is not defined
00003     * here, but it may be defined by applications that want to bend the rules.
00004     * The magic number in the definition is to inhibit unnecessary bending
00005     * of the rules.  (For consistency with the new '#ifdef _ANSI" tests in
00006     * the headers, _ANSI should really be defined as nothing, but that would
00007     * break many library routines that use "#if _ANSI".)
00008     *
00009     * If _ANSI ends up being defined, a macro
00010     *
00011     *      _PROTOTYPE(function, params)
00012     *
00013     * is defined.  This macro expands in different ways, generating either
00014     * ANSI Standard C prototypes or old-style K&R (Kernighan & Ritchie)
00015     * prototypes, as needed.  Finally, some programs use _CONST, _VOIDSTAR etc
00016     * in such a way that they are portable over both ANSI and K&R compilers.
00017     * The appropriate macros are defined here.
00018     */
00019
00020    #ifndef _ANSI_H
00021    #define _ANSI_H
00022
00023    #if __STDC__ == 1
00024    #define _ANSI           31459   /* compiler claims full ANSI conformance */
00025    #endif
00026
00027    #ifdef __GNUC__
00028    #define _ANSI           31459   /* gcc conforms enough even in non-ANSI mode */
00029    #endif
00030
00031    #ifdef _ANSI
00032
00033    /* Keep everything for ANSI prototypes. */
00034    #define _PROTOTYPE(function, params)    function params
00035    #define _ARGS(params)                   params
00036
00037    #define _VOIDSTAR       void *
00038    #define _VOID           void
00039    #define _CONST          const
00040    #define _VOLATILE       volatile
00041    #define _SIZET          size_t
00042
00043    #else
00044
00045    /* Throw away the parameters for K&R prototypes. */
00046    #define _PROTOTYPE(function, params)    function()
00047    #define _ARGS(params)                   ()
00048
00049    #define _VOIDSTAR       void *
00050    #define _VOID           void
00051    #define _CONST
00052    #define _VOLATILE
00053    #define _SIZET          int
00054
```



```
00055    #endif /* _ANSI */
00056
00057    #endif /* ANSI_H */


++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                include/limits.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

00100    /* The <limits.h> header defines some basic sizes, both of the language types
00101     * (e.g., the number of bits in an integer), and of the operating system (e.g.
00102     * the number of characters in a file name.
00103     */
00104
00105    #ifndef _LIMITS_H
00106    #define _LIMITS_H
00107
00108    /* Definitions about chars (8 bits in MINIX, and signed). */
00109    #define CHAR_BIT           8    /* # bits in a char */
00110    #define CHAR_MIN        -128    /* minimum value of a char */
00111    #define CHAR_MAX         127    /* maximum value of a char */
00112    #define SCHAR_MIN       -128    /* minimum value of a signed char */
00113    #define SCHAR_MAX        127    /* maximum value of a signed char */
00114    #define UCHAR_MAX        255    /* maximum value of an unsigned char */
00115    #define MB_LEN_MAX         1    /* maximum length of a multibyte char */
00116
00117    /* Definitions about shorts (16 bits in MINIX). */
00118    #define SHRT_MIN  (-32767-1)    /* minimum value of a short */
00119    #define SHRT_MAX       32767    /* maximum value of a short */
00120    #define USHRT_MAX     0xFFFF    /* maximum value of unsigned short */
00121
00122    /* _EM_WSIZE is a compiler-generated symbol giving the word size in bytes. */
00123    #if _EM_WSIZE == 2
00124    #define INT_MIN   (-32767-1)    /* minimum value of a 16-bit int */
00125    #define INT_MAX        32767    /* maximum value of a 16-bit int */
00126    #define UINT_MAX      0xFFFF    /* maximum value of an unsigned 16-bit int */
00127    #endif
00128
00129    #if _EM_WSIZE == 4
00130    #define INT_MIN (-2147483647-1) /* minimum value of a 32-bit int */
00131    #define INT_MAX   2147483647    /* maximum value of a 32-bit int */
00132    #define UINT_MAX  0xFFFFFFFF    /* maximum value of an unsigned 32-bit int */
00133    #endif
00134
00135    /*Definitions about longs (32 bits in MINIX). */
00136    #define LONG_MIN (-2147483647L-1)/* minimum value of a long */
00137    #define LONG_MAX  2147483647L   /* maximum value of a long */
00138    #define ULONG_MAX 0xFFFFFFFFL   /* maximum value of an unsigned long */
00139
00140    /* Minimum sizes required by the POSIX P1003.1 standard (Table 2-3). */
00141    #ifdef _POSIX_SOURCE            /* these are only visible for POSIX */
00142    #define _POSIX_ARG_MAX    4096  /* exec() may have 4K worth of args */
00143    #define _POSIX_CHILD_MAX     6  /* a process may have 6 children */
00144    #define _POSIX_LINK_MAX      8  /* a file may have 8 links */
00145    #define _POSIX_MAX_CANON   255  /* size of the canonical input queue */
00146    #define _POSIX_MAX_INPUT   255  /* you can type 255 chars ahead */
00147    #define _POSIX_NAME_MAX     14  /* a file name may have 14 chars */
00148    #define _POSIX_NGROUPS_MAX   0  /* supplementary group IDs are optional */
00149    #define _POSIX_OPEN_MAX     16  /* a process may have 16 files open */
```

```
00150    #define _POSIX_PATH_MAX    255  /* a pathname may contain 255 chars */
00151    #define _POSIX_PIPE_BUF    512  /* pipes writes of 512 bytes must be atomic */
00152    #define _POSIX_STREAM_MAX    8  /* at least 8 FILEs can be open at once */
00153    #define _POSIX_TZNAME_MAX    3  /* time zone names can be at least 3 chars */
00154    #define _POSIX_SSIZE_MAX 32767  /* read() must support 32767 byte reads */
00155
00156    /* Values actually implemented by MINIX (Tables 2-4, 2-5, 2-6, and 2-7). */
00157    /* Some of these old names had better be defined when not POSIX. */
00158    #define _NO_LIMIT        100    /* arbitrary number; limit not enforced */
00159
00160    #define NGROUPS_MAX        0    /* supplemental group IDs not available */
00161    #if _EM_WSIZE > 2
00162    #define ARG_MAX        16384    /* # bytes of args + environ for exec() */
00163    #else
00164    #define ARG_MAX         4096    /* args + environ on small machines */
00165    #endif
00166    #define CHILD_MAX  _NO_LIMIT    /* MINIX does not limit children */
00167    #define OPEN_MAX          20    /* # open files a process may have */
00168    #define LINK_MAX         127    /* # links a file may have */
00169    #define MAX_CANON        255    /* size of the canonical input queue */
00170    #define MAX_INPUT        255    /* size of the type-ahead buffer */
00171    #define NAME_MAX          14    /* # chars in a file name */
00172    #define PATH_MAX         255    /* # chars in a path name */
00173    #define PIPE_BUF        7168    /* # bytes in atomic write to a pipe */
00174    #define STREAM_MAX        20    /* must be the same as FOPEN_MAX in stdio.h */
00175    #define TZNAME_MAX         3    /* maximum bytes in a time zone name is 3 */
00176    #define SSIZE_MAX      32767    /* max defined byte count for read() */
00177
00178    #endif /* _POSIX_SOURCE */
00179
00180    #endif /* _LIMITS_H */
```

++

include/errno.h

++

```
00200    /* The <errno.h> header defines the numbers of the various errors that can
00201     * occur during program execution.  They are visible to user programs and
00202     * should be small positive integers.  However, they are also used within
00203     * MINIX, where they must be negative.  For example, the READ system call is
00204     * executed internally by calling do_read().  This function returns either a
00205     * (negative) error number or a (positive) number of bytes actually read.
00206     *
00207     * To solve the problem of having the error numbers be negative inside the
00208     * the system and positive outside, the following mechanism is used.  All the
00209     * definitions are are the form:
00210     *
00211     *      #define EPERM           (_SIGN 1)
00212     *
00213     * If the macro _SYSTEM is defined, then  _SIGN is set to "-", otherwise it is
00214     * set to "".  Thus when compiling the operating system, the  macro _SYSTEM
00215     * will be defined, setting EPERM to (- 1), whereas when when this
00216     * file is included in an ordinary user program, EPERM has the value ( 1).
00217     */
00218
00219    #ifndef _ERRNO_H                /* check if <errno.h> is already included */
```



```
00220    #define _ERRNO_H                /* it is not included; note that fact */
00221
00222    /* Now define _SIGN as "" or "-" depending on _SYSTEM. */
00223    #ifdef _SYSTEM
00224    #   define _SIGN         -
00225    #   define OK            0
00226    #else
00227    #   define _SIGN
00228    #endif
00229
00230    extern int errno;               /* place where the error numbers go */
00231
00232    /* Here are the numerical values of the error numbers. */
00233    #define _NERROR                70  /* number of errors */
00234
00235    #define EGENERIC      (_SIGN 99)  /* generic error */
00236    #define EPERM         (_SIGN  1)  /* operation not permitted */
00237    #define ENOENT        (_SIGN  2)  /* no such file or directory */
00238    #define ESRCH         (_SIGN  3)  /* no such process */
00239    #define EINTR         (_SIGN  4)  /* interrupted function call */
00240    #define EIO           (_SIGN  5)  /* input/output error */
00241    #define ENXIO         (_SIGN  6)  /* no such device or address */
00242    #define E2BIG         (_SIGN  7)  /* arg list too long */
00243    #define ENOEXEC       (_SIGN  8)  /* exec format error */
00244    #define EBADF         (_SIGN  9)  /* bad file descriptor */
00245    #define ECHILD        (_SIGN 10)  /* no child process */
00246    #define EAGAIN        (_SIGN 11)  /* resource temporarily unavailable */
00247    #define ENOMEM        (_SIGN 12)  /* not enough space */
00248    #define EACCES        (_SIGN 13)  /* permission denied */
00249    #define EFAULT        (_SIGN 14)  /* bad address */
00250    #define ENOTBLK       (_SIGN 15)  /* Extension: not a block special file */
00251    #define EBUSY         (_SIGN 16)  /* resource busy */
00252    #define EEXIST        (_SIGN 17)  /* file exists */
00253    #define EXDEV         (_SIGN 18)  /* improper link */
00254    #define ENODEV        (_SIGN 19)  /* no such device */
00255    #define ENOTDIR       (_SIGN 20)  /* not a directory */
00256    #define EISDIR        (_SIGN 21)  /* is a directory */
00257    #define EINVAL        (_SIGN 22)  /* invalid argument */
00258    #define ENFILE        (_SIGN 23)  /* too many open files in system */
00259    #define EMFILE        (_SIGN 24)  /* too many open files */
00260    #define ENOTTY        (_SIGN 25)  /* inappropriate I/O control operation */
00261    #define ETXTBSY       (_SIGN 26)  /* no longer used */
00262    #define EFBIG         (_SIGN 27)  /* file too large */
00263    #define ENOSPC        (_SIGN 28)  /* no space left on device */
00264    #define ESPIPE        (_SIGN 29)  /* invalid seek */
00265    #define EROFS         (_SIGN 30)  /* read-only file system */
00266    #define EMLINK        (_SIGN 31)  /* too many links */
00267    #define EPIPE         (_SIGN 32)  /* broken pipe */
00268    #define EDOM          (_SIGN 33)  /* domain error       (from ANSI C std) */
00269    #define ERANGE        (_SIGN 34)  /* result too large   (from ANSI C std) */
00270    #define EDEADLK       (_SIGN 35)  /* resource deadlock avoided */
00271    #define ENAMETOOLONG  (_SIGN 36)  /* file name too long */
00272    #define ENOLCK        (_SIGN 37)  /* no locks available */
00273    #define ENOSYS        (_SIGN 38)  /* function not implemented */
00274    #define ENOTEMPTY     (_SIGN 39)  /* directory not empty */
00275
00276    /* The following errors relate to networking. */
00277    #define EPACKSIZE     (_SIGN 50)  /* invalid packet size for some protocol */
00278    #define EOUTOFBUFS    (_SIGN 51)  /* not enough buffers left */
00279    #define EBADIOCTL     (_SIGN 52)  /* illegal ioctl for device */
```

```
00280    #define EBADMODE      (_SIGN 53)  /* badmode in ioctl */
00281    #define EWOULDBLOCK   (_SIGN 54)
00282    #define EBADDEST      (_SIGN 55)  /* not a valid destination address */
00283    #define EDSTNOTRCH    (_SIGN 56)  /* destination not reachable */
00284    #define EISCONN       (_SIGN 57)  /* all ready connected */
00285    #define EADDRINUSE    (_SIGN 58)  /* address in use */
00286    #define ECONNREFUSED  (_SIGN 59)  /* connection refused */
00287    #define ECONNRESET    (_SIGN 60)  /* connection reset */
00288    #define ETIMEDOUT     (_SIGN 61)  /* connection timed out */
00289    #define EURG          (_SIGN 62)  /* urgent data present */
00290    #define ENOURG        (_SIGN 63)  /* no urgent data present */
00291    #define ENOTCONN      (_SIGN 64)  /* no connection (yet or anymore) */
00292    #define ESHUTDOWN     (_SIGN 65)  /* a write call to a shutdown connection */
00293    #define ENOCONN       (_SIGN 66)  /* no such connection */
00294
00295    /* The following are not POSIX errors, but they can still happen. */
00296    #define ELOCKED      (_SIGN 101)  /* can't send message */
00297    #define EBADCALL     (_SIGN 102)  /* error on send/receive */
00298
00299    /* The following error codes are generated by the kernel itself. */
00300    #ifdef _SYSTEM
00301    #define E_BAD_DEST     -1001    /* destination address illegal */
00302    #define E_BAD_SRC      -1002    /* source address illegal */
00303    #define E_TRY_AGAIN    -1003    /* can't send-- tables full */
00304    #define E_OVERRUN      -1004    /* interrupt for task that is not waiting */
00305    #define E_BAD_BUF      -1005    /* message buf outside caller's addr space */
00306    #define E_TASK         -1006    /* can't send to task */
00307    #define E_NO_MESSAGE   -1007    /* RECEIVE failed: no message present */
00308    #define E_NO_PERM      -1008    /* ordinary users can't send to tasks */
00309    #define E_BAD_FCN      -1009    /* only valid fcns are SEND, RECEIVE, BOTH */
00310    #define E_BAD_ADDR     -1010    /* bad address given to utility routine */
00311    #define E_BAD_PROC     -1011    /* bad proc number given to utility */
00312    #endif /* _SYSTEM */
00313
00314    #endif /* _ERRNO_H */
```

++

include/unistd.h

++

```
00400    /* The <unistd.h> header contains a few miscellaneous manifest constants. */
00401
00402    #ifndef _UNISTD_H
00403    #define _UNISTD_H
00404
00405    /* POSIX requires size_t and ssize_t in <unistd.h> and elsewhere. */
00406    #ifndef _SIZE_T
00407    #define _SIZE_T
00408    typedef unsigned int size_t;
00409    #endif
00410
00411    #ifndef _SSIZE_T
00412    #define _SSIZE_T
00413    typedef int ssize_t;
00414    #endif
```



```
00415
00416    /* Values used by access().  POSIX Table 2-8. */
00417    #define F_OK               0    /* test if file exists */
00418    #define X_OK               1    /* test if file is executable */
00419    #define W_OK               2    /* test if file is writable */
00420    #define R_OK               4    /* test if file is readable */
00421
00422    /* Values used for whence in lseek(fd, offset, whence).  POSIX Table 2-9. */
00423    #define SEEK_SET           0    /* offset is absolute  */
00424    #define SEEK_CUR           1    /* offset is relative to current position */
00425    #define SEEK_END           2    /* offset is relative to end of file */
00426
00427    /* This value is required by POSIX Table 2-10. */
00428    #define _POSIX_VERSION 199009L  /* which standard is being conformed to */
00429
00430    /* These three definitions are required by POSIX Sec. 8.2.1.2. */
00431    #define STDIN_FILENO       0    /* file descriptor for stdin */
00432    #define STDOUT_FILENO      1    /* file descriptor for stdout */
00433    #define STDERR_FILENO      2    /* file descriptor for stderr */
00434
00435    #ifdef _MINIX
00436    /* How to exit the system. */
00437    #define RBT_HALT           0
00438    #define RBT_REBOOT         1
00439    #define RBT_PANIC          2    /* for servers */
00440    #define RBT_MONITOR        3    /* let the monitor do this */
00441    #define RBT_RESET          4    /* hard reset the system */
00442    #endif
00443
00444    /* NULL must be defined in <unistd.h> according to POSIX Sec. 2.7.1. */
00445    #define NULL    ((void *)0)
00446
00447    /* The following relate to configurable system variables. POSIX Table 4-2. */
00448    #define _SC_ARG_MAX             1
00449    #define _SC_CHILD_MAX           2
00450    #define _SC_CLOCKS_PER_SEC      3
00451    #define _SC_CLK_TCK             3
00452    #define _SC_NGROUPS_MAX         4
00453    #define _SC_OPEN_MAX            5
00454    #define _SC_JOB_CONTROL         6
00455    #define _SC_SAVED_IDS           7
00456    #define _SC_VERSION             8
00457    #define _SC_STREAM_MAX          9
00458    #define _SC_TZNAME_MAX         10
00459
00460    /* The following relate to configurable pathname variables. POSIX Table 5-2. */
00461    #define _PC_LINK_MAX            1       /* link count */
00462    #define _PC_MAX_CANON           2       /* size of the canonical input queue */
00463    #define _PC_MAX_INPUT           3       /* type-ahead buffer size */
00464    #define _PC_NAME_MAX            4       /* file name size */
00465    #define _PC_PATH_MAX            5       /* pathname size */
00466    #define _PC_PIPE_BUF            6       /* pipe size */
00467    #define _PC_NO_TRUNC            7       /* treatment of long name components */
00468    #define _PC_VDISABLE            8       /* tty disable */
00469    #define _PC_CHOWN_RESTRICTED    9       /* chown restricted or not */
00470
00471    /* POSIX defines several options that may be implemented or not, at the
00472     * implementer's whim.  This implementer has made the following choices:
00473     *
00474     * _POSIX_JOB_CONTROL       not defined:        no job control
```

```
00475    * _POSIX_SAVED_IDS         not defined:        no saved uid/gid
00476    * _POSIX_NO_TRUNC          defined as -1:      long path names are truncated
00477    * _POSIX_CHOWN_RESTRICTED  defined:            you can't give away files
00478    * _POSIX_VDISABLE          defined:            tty functions can be disabled
00479    */
00480   #define _POSIX_NO_TRUNC       (-1)
00481   #define _POSIX_CHOWN_RESTRICTED  1
00482
00483   /* Function Prototypes. */
00484   #ifndef _ANSI_H
00485   #include <ansi.h>
00486   #endif
00487
00488   _PROTOTYPE( void _exit, (int _status)                                   );
00489   _PROTOTYPE( int access, (const char *_path, int _amode)                 );
00490   _PROTOTYPE( unsigned int alarm, (unsigned int _seconds)                 );
00491   _PROTOTYPE( int chdir, (const char *_path)                              );
00492   _PROTOTYPE( int chown, (const char *_path, Uid_t _owner, Gid_t _group)  );
00493   _PROTOTYPE( int close, (int _fd)                                        );
00494   _PROTOTYPE( char *ctermid, (char *_s)                                   );
00495   _PROTOTYPE( char *cuserid, (char *_s)                                   );
00496   _PROTOTYPE( int dup, (int _fd)                                          );
00497   _PROTOTYPE( int dup2, (int _fd, int _fd2)                               );
00498   _PROTOTYPE( int execl, (const char *_path, const char *_arg, ...)       );
00499   _PROTOTYPE( int execle, (const char *_path, const char *_arg, ...)      );
00500   _PROTOTYPE( int execlp, (const char *_file, const char *arg, ...)       );
00501   _PROTOTYPE( int execv, (const char *_path, char *const _argv[])         );
00502   _PROTOTYPE( int execve, (const char *_path, char *const _argv[],
00503                                               char *const _envp[])        );
00504   _PROTOTYPE( int execvp, (const char *_file, char *const _argv[])        );
00505   _PROTOTYPE( pid_t fork, (void)                                          );
00506   _PROTOTYPE( long fpathconf, (int _fd, int _name)                        );
00507   _PROTOTYPE( char *getcwd, (char *_buf, size_t _size)                    );
00508   _PROTOTYPE( gid_t getegid, (void)                                       );
00509   _PROTOTYPE( uid_t geteuid, (void)                                       );
00510   _PROTOTYPE( gid_t getgid, (void)                                        );
00511   _PROTOTYPE( int getgroups, (int _gidsetsize, gid_t _grouplist[])        );
00512   _PROTOTYPE( char *getlogin, (void)                                      );
00513   _PROTOTYPE( pid_t getpgrp, (void)                                       );
00514   _PROTOTYPE( pid_t getpid, (void)                                        );
00515   _PROTOTYPE( pid_t getppid, (void)                                       );
00516   _PROTOTYPE( uid_t getuid, (void)                                        );
00517   _PROTOTYPE( int isatty, (int _fd)                                       );
00518   _PROTOTYPE( int link, (const char *_existing, const char *_new)         );
00519   _PROTOTYPE( off_t lseek, (int _fd, off_t _offset, int _whence)          );
00520   _PROTOTYPE( long pathconf, (const char *_path, int _name)               );
00521   _PROTOTYPE( int pause, (void)                                           );
00522   _PROTOTYPE( int pipe, (int _fildes[2])                                  );
00523   _PROTOTYPE( ssize_t read, (int _fd, void *_buf, size_t _n)              );
00524   _PROTOTYPE( int rmdir, (const char *_path)                              );
00525   _PROTOTYPE( int setgid, (Gid_t _gid)                                    );
00526   _PROTOTYPE( int setpgid, (pid_t _pid, pid_t _pgid)                      );
00527   _PROTOTYPE( pid_t setsid, (void)                                        );
00528   _PROTOTYPE( int setuid, (Uid_t _uid)                                    );
00529   _PROTOTYPE( unsigned int sleep, (unsigned int _seconds)                 );
00530   _PROTOTYPE( long sysconf, (int _name)                                   );
00531   _PROTOTYPE( pid_t tcgetpgrp, (int _fd)                                  );
00532   _PROTOTYPE( int tcsetpgrp, (int _fd, pid_t _pgrp_id)                    );
00533   _PROTOTYPE( char *ttyname, (int _fd)                                    );
00534   _PROTOTYPE( int unlink, (const char *_path)                             );
```



```
00535   _PROTOTYPE( ssize_t write, (int _fd, const void *_buf, size_t _n)       );
00536
00537   #ifdef _MINIX
00538   _PROTOTYPE( int brk, (char *_addr)                                      );
00539   _PROTOTYPE( int chroot, (const char *_name)                             );
00540   _PROTOTYPE( int mknod, (const char *_name, Mode_t _mode, Dev_t _addr)   );
00541   _PROTOTYPE( int mknod4, (const char *_name, Mode_t _mode, Dev_t _addr,
00542                long _size)                                                );
00543   _PROTOTYPE( char *mktemp, (char *_template)                             );
00544   _PROTOTYPE( int mount, (char *_spec, char *_name, int _flag)            );
00545   _PROTOTYPE( long ptrace, (int _req, pid_t _pid, long _addr, long _data) );
00546   _PROTOTYPE( char *sbrk, (int _incr)                                     );
00547   _PROTOTYPE( int sync, (void)                                            );
00548   _PROTOTYPE( int umount, (const char *_name)                             );
00549   _PROTOTYPE( int reboot, (int _how, ...)                                 );
00550   _PROTOTYPE( int gethostname, (char *_hostname, size_t _len)             );
00551   _PROTOTYPE( int getdomainname, (char *_domain, size_t _len)             );
00552   _PROTOTYPE( int ttyslot, (void)                                         );
00553   _PROTOTYPE( int fttyslot, (int _fd)                                     );
00554   _PROTOTYPE( char *crypt, (const char *_key, const char *_salt)          );
00555   #endif
00556
00557   #endif /* _UNISTD_H */
```

include/string.h

```
00600   /* The <string.h> header contains prototypes for the string handling
00601    * functions.
00602    */
00603
00604   #ifndef _STRING_H
00605   #define _STRING_H
00606
00607   #define NULL    ((void *)0)
00608
00609   #ifndef _SIZE_T
00610   #define _SIZE_T
00611   typedef unsigned int size_t;    /* type returned by sizeof */
00612   #endif /*_SIZE_T */
00613
00614   /* Function Prototypes. */
00615   #ifndef _ANSI_H
00616   #include <ansi.h>
00617   #endif
00618
00619   _PROTOTYPE( void *memchr, (const void *_s, int _c, size_t _n)           );
00620   _PROTOTYPE( int memcmp, (const void *_s1, const void *_s2, size_t _n)   );
00621   _PROTOTYPE( void *memcpy, (void *_s1, const void *_s2, size_t _n)       );
00622   _PROTOTYPE( void *memmove, (void *_s1, const void *_s2, size_t _n)      );
00623   _PROTOTYPE( void *memset, (void *_s, int _c, size_t _n)                 );
00624   _PROTOTYPE( char *strcat, (char *_s1, const char *_s2)                  );
00625   _PROTOTYPE( char *strchr, (const char *_s, int _c)                      );
00626   _PROTOTYPE( int strncmp, (const char *_s1, const char *_s2, size_t _n)  );
00627   _PROTOTYPE( int strcmp, (const char *_s1, const char *_s2)              );
00628   _PROTOTYPE( int strcoll, (const char *_s1, const char *_s2)             );
00629   _PROTOTYPE( char *strcpy, (char *_s1, const char *_s2)                  );
```

```
00630    _PROTOTYPE( size_t strcspn, (const char *_s1, const char *_s2)          );
00631    _PROTOTYPE( char *strerror, (int _errnum)                               );
00632    _PROTOTYPE( size_t strlen, (const char *_s)                             );
00633    _PROTOTYPE( char *strncat, (char *_s1, const char *_s2, size_t _n)      );
00634    _PROTOTYPE( char *strncpy, (char *_s1, const char *_s2, size_t _n)      );
00635    _PROTOTYPE( char *strpbrk, (const char *_s1, const char *_s2)           );
00636    _PROTOTYPE( char *strrchr, (const char *_s, int _c)                     );
00637    _PROTOTYPE( size_t strspn, (const char *_s1, const char *_s2)           );
00638    _PROTOTYPE( char *strstr, (const char *_s1, const char *_s2)            );
00639    _PROTOTYPE( char *strtok, (char *_s1, const char *_s2)                  );
00640    _PROTOTYPE( size_t strxfrm, (char *_s1, const char *_s2, size_t _n)     );
00641
00642    #ifdef _MINIX
00643    /* For backward compatibility. */
00644    _PROTOTYPE( char *index, (const char *_s, int _charwanted)              );
00645    _PROTOTYPE( char *rindex, (const char *_s, int _charwanted)             );
00646    _PROTOTYPE( void bcopy, (const void *_src, void *_dst, size_t _length)  );
00647    _PROTOTYPE( int bcmp, (const void *_s1, const void *_s2, size_t _length));
00648    _PROTOTYPE( void bzero, (void *_dst, size_t _length)                    );
00649    _PROTOTYPE( void *memccpy, (char *_dst, const char *_src, int _ucharstop,
00650                                                        size_t _size)       );
00651    /* BSD functions */
00652    _PROTOTYPE( int strcasecmp, (const char *_s1, const char *_s2)          );
00653    #endif
00654
00655    #endif /* _STRING_H */
```

++

include/signal.h

++

```
00700    /* The <signal.h> header defines all the ANSI and POSIX signals.
00701     * MINIX supports all the signals required by POSIX. They are defined below.
00702     * Some additional signals are also supported.
00703     */
00704
00705    #ifndef _SIGNAL_H
00706    #define _SIGNAL_H
00707
00708    #ifndef _ANSI_H
00709    #include <ansi.h>
00710    #endif
00711
00712    /* Here are types that are closely associated with signal handling. */
00713    typedef int sig_atomic_t;
00714
00715    #ifdef _POSIX_SOURCE
00716    #ifndef _SIGSET_T
00717    #define _SIGSET_T
00718    typedef unsigned long sigset_t;
00719    #endif
00720    #endif
00721
00722    #define _NSIG             16    /* number of signals used */
00723
00724    #define SIGHUP             1    /* hangup */
```



```
00725    #define SIGINT             2    /* interrupt (DEL) */
00726    #define SIGQUIT            3    /* quit (ASCII FS) */
00727    #define SIGILL             4    /* illegal instruction */
00728    #define SIGTRAP            5    /* trace trap (not reset when caught) */
00729    #define SIGABRT            6    /* IOT instruction */
00730    #define SIGIOT             6    /* SIGABRT for people who speak PDP-11 */
00731    #define SIGUNUSED          7    /* spare code */
00732    #define SIGFPE             8    /* floating point exception */
00733    #define SIGKILL            9    /* kill (cannot be caught or ignored) */
00734    #define SIGUSR1           10    /* user defined signal # 1 */
00735    #define SIGSEGV           11    /* segmentation violation */
00736    #define SIGUSR2           12    /* user defined signal # 2 */
00737    #define SIGPIPE           13    /* write on a pipe with no one to read it */
00738    #define SIGALRM           14    /* alarm clock */
00739    #define SIGTERM           15    /* software termination signal from kill */
00740
00741    #define SIGEMT             7    /* obsolete */
00742    #define SIGBUS            10    /* obsolete */
00743
00744    /* POSIX requires the following signals to be defined, even if they are
00745     * not supported.  Here are the definitions, but they are not supported.
00746     */
00747    #define SIGCHLD           17    /* child process terminated or stopped */
00748    #define SIGCONT           18    /* continue if stopped */
00749    #define SIGSTOP           19    /* stop signal */
00750    #define SIGTSTP           20    /* interactive stop signal */
00751    #define SIGTTIN           21    /* background process wants to read */
00752    #define SIGTTOU           22    /* background process wants to write */
00753
00754    /* The sighandler_t type is not allowed unless _POSIX_SOURCE is defined. */
00755    #ifdef _POSIX_SOURCE
00756    #define __sighandler_t sighandler_t
00757    #else
00758    typedef void (*__sighandler_t) (int);
00759    #endif
00760
00761    /* Macros used as function pointers. */
00762    #define SIG_ERR    ((__sighandler_t) -1)        /* error return */
00763    #define SIG_DFL    ((__sighandler_t)  0)        /* default signal handling */
00764    #define SIG_IGN    ((__sighandler_t)  1)        /* ignore signal */
00765    #define SIG_HOLD   ((__sighandler_t)  2)        /* block signal */
00766    #define SIG_CATCH  ((__sighandler_t)  3)        /* catch signal */
00767
00768    #ifdef _POSIX_SOURCE
00769    struct sigaction {
00770      __sighandler_t sa_handler;    /* SIG_DFL, SIG_IGN, or pointer to function */
00771      sigset_t sa_mask;             /* signals to be blocked during handler */
00772      int sa_flags;                 /* special flags */
00773    };
00774
00775    /* Fields for sa_flags. */
00776    #define SA_ONSTACK   0x0001     /* deliver signal on alternate stack */
00777    #define SA_RESETHAND 0x0002     /* reset signal handler when signal caught */
00778    #define SA_NODEFER   0x0004     /* don't block signal while catching it */
00779    #define SA_RESTART   0x0008     /* automatic system call restart */
00780    #define SA_SIGINFO   0x0010     /* extended signal handling */
00781    #define SA_NOCLDWAIT 0x0020     /* don't create zombies */
00782    #define SA_NOCLDSTOP 0x0040     /* don't receive SIGCHLD when child stops */
00783
00784    /* POSIX requires these values for use with sigprocmask(2). */
```

```
00785    #define SIG_BLOCK          0    /* for blocking signals */
00786    #define SIG_UNBLOCK        1    /* for unblocking signals */
00787    #define SIG_SETMASK        2    /* for setting the signal mask */
00788    #define SIG_INQUIRE        4    /* for internal use only */
00789    #endif  /* _POSIX_SOURCE */
00790
00791    /* POSIX and ANSI function prototypes. */
00792    _PROTOTYPE( int raise, (int _sig)                                       );
00793    _PROTOTYPE( __sighandler_t signal, (int _sig, __sighandler_t _func)     );
00794
00795    #ifdef _POSIX_SOURCE
00796    _PROTOTYPE( int kill, (pid_t _pid, int _sig)                            );
00797    _PROTOTYPE( int sigaction,
00798        (int _sig, const struct sigaction *_act, struct sigaction *_oact)   );
00799    _PROTOTYPE( int sigaddset, (sigset_t *_set, int _sig)                   );
00800    _PROTOTYPE( int sigdelset, (sigset_t *_set, int _sig)                   );
00801    _PROTOTYPE( int sigemptyset, (sigset_t *_set)                           );
00802    _PROTOTYPE( int sigfillset, (sigset_t *_set)                            );
00803    _PROTOTYPE( int sigismember, (sigset_t *_set, int _sig)                 );
00804    _PROTOTYPE( int sigpending, (sigset_t *_set)                            );
00805    _PROTOTYPE( int sigprocmask,
00806                (int _how, const sigset_t *_set, sigset_t *_oset)           );
00807    _PROTOTYPE( int sigsuspend, (const sigset_t *_sigmask)                  );
00808    #endif
00809
00810    #endif /* _SIGNAL_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                              include/fcntl.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

00900    /* The <fcntl.h> header is needed by the open() and fcntl() system calls,
00901     * which  have a variety of parameters and flags.  They are described here.
00902     * The formats of the calls to each of these are:
00903     *
00904     *      open(path, oflag [,mode])       open a file
00905     *      fcntl(fd, cmd [,arg])           get or set file attributes
00906     *
00907     */
00908
00909    #ifndef _FCNTL_H
00910    #define _FCNTL_H
00911
00912    /* These values are used for cmd in fcntl().  POSIX Table 6-1.  */
00913    #define F_DUPFD            0    /* duplicate file descriptor */
00914    #define F_GETFD            1    /* get file descriptor flags */
00915    #define F_SETFD            2    /* set file descriptor flags */
00916    #define F_GETFL            3    /* get file status flags */
00917    #define F_SETFL            4    /* set file status flags */
00918    #define F_GETLK            5    /* get record locking information */
00919    #define F_SETLK            6    /* set record locking information */
00920    #define F_SETLKW           7    /* set record locking info; wait if blocked */
00921
00922    /* File descriptor flags used for fcntl().  POSIX Table 6-2. */
00923    #define FD_CLOEXEC         1    /* close on exec flag for third arg of fcntl */
00924
```



```
00925    /* L_type values for record locking with fcntl().  POSIX Table 6-3. */
00926    #define F_RDLCK            1    /* shared or read lock */
00927    #define F_WRLCK            2    /* exclusive or write lock */
00928    #define F_UNLCK            3    /* unlock */
00929
00930    /* Oflag values for open().  POSIX Table 6-4. */
00931    #define O_CREAT        00100    /* creat file if it doesn't exist */
00932    #define O_EXCL         00200    /* exclusive use flag */
00933    #define O_NOCTTY       00400    /* do not assign a controlling terminal */
00934    #define O_TRUNC        01000    /* truncate flag */
00935
00936    /* File status flags for open() and fcntl().  POSIX Table 6-5. */
00937    #define O_APPEND       02000    /* set append mode */
00938    #define O_NONBLOCK     04000    /* no delay */
00939
00940    /* File access modes for open() and fcntl().  POSIX Table 6-6. */
00941    #define O_RDONLY           0    /* open(name, O_RDONLY) opens read only */
00942    #define O_WRONLY           1    /* open(name, O_WRONLY) opens write only */
00943    #define O_RDWR             2    /* open(name, O_RDWR) opens read/write */
00944
00945    /* Mask for use with file access modes.  POSIX Table 6-7. */
00946    #define O_ACCMODE         03    /* mask for file access modes */
00947
00948    /* Struct used for locking.  POSIX Table 6-8. */
00949    struct flock {
00950      short l_type;                 /* type: F_RDLCK, F_WRLCK, or F_UNLCK */
00951      short l_whence;               /* flag for starting offset */
00952      off_t l_start;                /* relative offset in bytes */
00953      off_t l_len;                  /* size; if 0, then until EOF */
00954      pid_t l_pid;                  /* process id of the locks' owner */
00955    };
00956
00957
00958    /* Function Prototypes. */
00959    #ifndef _ANSI_H
00960    #include <ansi.h>
00961    #endif
00962
00963    _PROTOTYPE( int creat, (const char *_path, Mode_t _mode)                );
00964    _PROTOTYPE( int fcntl, (int _filedes, int _cmd, ...)                    );
00965    _PROTOTYPE( int open,  (const char *_path, int _oflag, ...)             );
00966
00967    #endif /* _FCNTL_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                              include/stdlib.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

01000    /* The <stdlib.h> header defines certain common macros, types, and functions.*/
01001
01002    #ifndef _STDLIB_H
01003    #define _STDLIB_H
01004
01005    /* The macros are NULL, EXIT_FAILURE, EXIT_SUCCESS, RAND_MAX, and MB_CUR_MAX.*/
01006    #define NULL    ((void *)0)
01007
01008    #define EXIT_FAILURE       1    /* standard error return using exit() */
01009    #define EXIT_SUCCESS       0    /* successful return using exit() */
```

```
01010   #define RAND_MAX       32767    /* largest value generated by rand() */
01011   #define MB_CUR_MAX         1    /* max value of multibyte character in MINIX */
01012
01013   typedef struct { int quot, rem; } div_t;
01014   typedef struct { long quot, rem; } ldiv_t;
01015
01016   /* The types are size_t, wchar_t, div_t, and ldiv_t. */
01017   #ifndef _SIZE_T
01018   #define _SIZE_T
01019   typedef unsigned int size_t;    /* type returned by sizeof */
01020   #endif
01021
01022   #ifndef _WCHAR_T
01023   #define _WCHAR_T
01024   typedef char wchar_t;           /* type expanded character set */
01025   #endif
01026
01027   /* Function Prototypes. */
01028   #ifndef _ANSI_H
01029   #include <ansi.h>
01030   #endif
01031
01032   _PROTOTYPE( void abort, (void)                                          );
01033   _PROTOTYPE( int abs, (int _j)                                           );
01034   _PROTOTYPE( int atexit, (void (*_func)(void))                           );
01035   _PROTOTYPE( double atof, (const char *_nptr)                            );
01036   _PROTOTYPE( int atoi, (const char *_nptr)                               );
01037   _PROTOTYPE( long atol, (const char *_nptr)                              );
01038   _PROTOTYPE( void *calloc, (size_t _nmemb, size_t _size)                 );
01039   _PROTOTYPE( div_t div, (int _numer, int _denom)                         );
01040   _PROTOTYPE( void exit, (int _status)                                    );
01041   _PROTOTYPE( void free, (void *_ptr)                                     );
01042   _PROTOTYPE( char *getenv, (const char *_name)                           );
01043   _PROTOTYPE( long labs, (long _j)                                        );
01044   _PROTOTYPE( ldiv_t ldiv, (long _numer, long _denom)                     );
01045   _PROTOTYPE( void *malloc, (size_t _size)                                );
01046   _PROTOTYPE( int mblen, (const char *_s, size_t _n)                      );
01047   _PROTOTYPE( size_t mbstowcs, (wchar_t *_pwcs, const char *_s, size_t _n));
01048   _PROTOTYPE( int mbtowc, (wchar_t *_pwc, const char *_s, size_t _n)      );
01049   _PROTOTYPE( int rand, (void)                                            );
01050   _PROTOTYPE( void *realloc, (void *_ptr, size_t _size)                   );
01051   _PROTOTYPE( void srand, (unsigned int _seed)                            );
01052   _PROTOTYPE( double strtod, (const char *_nptr, char **_endptr)          );
01053   _PROTOTYPE( long strtol, (const char *_nptr, char **_endptr, int _base) );
01054   _PROTOTYPE( int system, (const char *_string)                           );
01055   _PROTOTYPE( size_t wcstombs, (char *_s, const wchar_t *_pwcs, size_t _n));
01056   _PROTOTYPE( int wctomb, (char *_s, wchar_t _wchar)                      );
01057   _PROTOTYPE( void *bsearch, (const void *_key, const void *_base,
01058           size_t _nmemb, size_t _size,
01059           int (*compar) (const void *, const void *))                     );
01060   _PROTOTYPE( void qsort, (void *_base, size_t _nmemb, size_t _size,
01061           int (*compar) (const void *, const void *))                     );
01062   _PROTOTYPE( unsigned long int strtoul,
01063                           (const char *_nptr, char **_endptr, int _base)  );
01064
01065   #ifdef _MINIX
01066   _PROTOTYPE( int putenv, (const char *_name)                             );
01067   _PROTOTYPE(int getopt, (int _argc, char **_argv, char *_opts));
01068   extern char *optarg;
01069   extern int optind, opterr, optopt;
```



```
01070   #endif /* _MINIX */
01071
01072   #endif /* STDLIB_H */
```

++

include/termios.h

++

```
01100   /* The <termios.h> header is used for controlling tty modes. */
01101
01102   #ifndef _TERMIOS_H
01103   #define _TERMIOS_H
01104
01105   typedef unsigned short tcflag_t;
01106   typedef unsigned char cc_t;
01107   typedef unsigned int speed_t;
01108
01109   #define NCCS            20      /* size of cc_c array, some extra space
01110                                    * for extensions. */
01111
01112   /* Primary terminal control structure. POSIX Table 7-1. */
01113   struct termios {
01114     tcflag_t c_iflag;             /* input modes */
01115     tcflag_t c_oflag;             /* output modes */
01116     tcflag_t c_cflag;             /* control modes */
01117     tcflag_t c_lflag;             /* local modes */
01118     speed_t  c_ispeed;            /* input speed */
01119     speed_t  c_ospeed;            /* output speed */
01120     cc_t c_cc[NCCS];              /* control characters */
01121   };
01122
01123   /* Values for termios c_iflag bit map.  POSIX Table 7-2. */
01124   #define BRKINT          0x0001  /* signal interrupt on break */
01125   #define ICRNL           0x0002  /* map CR to NL on input */
01126   #define IGNBRK          0x0004  /* ignore break */
01127   #define IGNCR           0x0008  /* ignore CR */
01128   #define IGNPAR          0x0010  /* ignore characters with parity errors */
01129   #define INLCR           0x0020  /* map NL to CR on input */
01130   #define INPCK           0x0040  /* enable input parity check */
01131   #define ISTRIP          0x0080  /* mask off 8th bit */
01132   #define IXOFF           0x0100  /* enable start/stop input control */
01133   #define IXON            0x0200  /* enable start/stop output control */
01134   #define PARMRK          0x0400  /* mark parity errors in the input queue */
01135
01136   /* Values for termios c_oflag bit map.  POSIX Sec. 7.1.2.3. */
01137   #define OPOST           0x0001  /* perform output processing */
01138
01139   /* Values for termios c_cflag bit map.  POSIX Table 7-3. */
01140   #define CLOCAL          0x0001  /* ignore modem status lines */
01141   #define CREAD           0x0002  /* enable receiver */
01142   #define CSIZE           0x000C  /* number of bits per character */
01143   #define         CS5     0x0000  /* if CSIZE is CS5, characters are 5 bits */
01144   #define         CS6     0x0004  /* if CSIZE is CS6, characters are 6 bits */
01145   #define         CS7     0x0008  /* if CSIZE is CS7, characters are 7 bits */
01146   #define         CS8     0x000C  /* if CSIZE is CS8, characters are 8 bits */
01147   #define CSTOPB          0x0010  /* send 2 stop bits if set, else 1 */
01148   #define HUPCL           0x0020  /* hang up on last close */
01149   #define PARENB          0x0040  /* enable parity on output */
```

```
#define PARODD          0x0080  /* use odd parity if set, else even */

/* Values for termios c_lflag bit map.  POSIX Table 7-4. */
#define ECHO            0x0001  /* enable echoing of input characters */
#define ECHOE           0x0002  /* echo ERASE as backspace */
#define ECHOK           0x0004  /* echo KILL */
#define ECHONL          0x0008  /* echo NL */
#define ICANON          0x0010  /* canonical input (erase and kill enabled) */
#define IEXTEN          0x0020  /* enable extended functions */
#define ISIG            0x0040  /* enable signals */
#define NOFLSH          0x0080  /* disable flush after interrupt or quit */
#define TOSTOP          0x0100  /* send SIGTTOU (job control, not implemented*/

/* Indices into c_cc array.  Default values in parentheses. POSIX Table 7-5. */
#define VEOF              0     /* cc_c[VEOF] = EOF char (^D) */
#define VEOL              1     /* cc_c[VEOL] = EOL char (undef) */
#define VERASE            2     /* cc_c[VERASE] = ERASE char (^H) */
#define VINTR             3     /* cc_c[VINTR] = INTR char (DEL) */
#define VKILL             4     /* cc_c[VKILL] = KILL char (^U) */
#define VMIN              5     /* cc_c[VMIN] = MIN value for timer */
#define VQUIT             6     /* cc_c[VQUIT] = QUIT char (^\) */
#define VTIME             7     /* cc_c[VTIME] = TIME value for timer */
#define VSUSP             8     /* cc_c[VSUSP] = SUSP (^Z, ignored) */
#define VSTART            9     /* cc_c[VSTART] = START char (^S) */
#define VSTOP            10     /* cc_c[VSTOP] = STOP char (^Q) */

#define _POSIX_VDISABLE   (cc_t)0xFF    /* You can't even generate this
                                         * character with 'normal' keyboards.
                                         * But some language specific keyboards
                                         * can generate 0xFF. It seems that all
                                         * 256 are used, so cc_t should be a
                                         * short...
                                         */

/* Values for the baud rate settings.  POSIX Table 7-6. */
#define B0              0x0000  /* hang up the line */
#define B50             0x1000  /* 50 baud */
#define B75             0x2000  /* 75 baud */
#define B110            0x3000  /* 110 baud */
#define B134            0x4000  /* 134.5 baud */
#define B150            0x5000  /* 150 baud */
#define B200            0x6000  /* 200 baud */
#define B300            0x7000  /* 300 baud */
#define B600            0x8000  /* 600 baud */
#define B1200           0x9000  /* 1200 baud */
#define B1800           0xA000  /* 1800 baud */
#define B2400           0xB000  /* 2400 baud */
#define B4800           0xC000  /* 4800 baud */
#define B9600           0xD000  /* 9600 baud */
#define B19200          0xE000  /* 19200 baud */
#define B38400          0xF000  /* 38400 baud */

/* Optional actions for tcsetattr().  POSIX Sec. 7.2.1.2. */
#define TCSANOW           1     /* changes take effect immediately */
#define TCSADRAIN         2     /* changes take effect after output is done */
#define TCSAFLUSH         3     /* wait for output to finish and flush input */

/* Queue_selector values for tcflush().  POSIX Sec. 7.2.2.2. */
#define TCIFLUSH          1     /* flush accumulated input data */
#define TCOFLUSH          2     /* flush accumulated output data */
```



```
#define TCIOFLUSH         3     /* flush accumulated input and output data */

/* Action values for tcflow().  POSIX Sec. 7.2.2.2. */
#define TCOOFF            1     /* suspend output */
#define TCOON             2     /* restart suspended output */
#define TCIOFF            3     /* transmit a STOP character on the line */
#define TCION             4     /* transmit a START character on the line */


/* Function Prototypes. */
#ifndef _ANSI_H
#include <ansi.h>
#endif

_PROTOTYPE( int tcsendbreak, (int _fildes, int _duration)                   );
_PROTOTYPE( int tcdrain, (int _filedes)                                     );
_PROTOTYPE( int tcflush, (int _filedes, int _queue_selector)                );
_PROTOTYPE( int tcflow, (int _filedes, int _action)                         );
_PROTOTYPE( speed_t cfgetispeed, (const struct termios *_termios_p)         );
_PROTOTYPE( speed_t cfgetospeed, (const struct termios *_termios_p)         );
_PROTOTYPE( int cfsetispeed, (struct termios *_termios_p, speed_t _speed)   );
_PROTOTYPE( int cfsetospeed, (struct termios *_termios_p, speed_t _speed)   );
_PROTOTYPE( int tcgetattr, (int _filedes, struct termios *_termios_p)       );
_PROTOTYPE( int tcsetattr, \
        (int _filedes, int _opt_actions, const struct termios *_termios_p)  );

#define cfgetispeed(termios_p)          ((termios_p)->c_ispeed)
#define cfgetospeed(termios_p)          ((termios_p)->c_ospeed)
#define cfsetispeed(termios_p, speed)   ((termios_p)->c_ispeed = (speed), 0)
#define cfsetospeed(termios_p, speed)   ((termios_p)->c_ospeed = (speed), 0)

#ifdef _MINIX
/* Here are the local extensions to the POSIX standard for Minix. Posix
 * conforming programs are not able to access these, and therefore they are
 * only defined when a Minix program is compiled.
 */

/* Extensions to the termios c_iflag bit map.  */
#define IXANY           0x0800  /* allow any key to continue ouptut */

/* Extensions to the termios c_oflag bit map. They are only active iff
 * OPOST is enabled. */
#define ONLCR           0x0002  /* Map NL to CR-NL on output */
#define XTABS           0x0004  /* Expand tabs to spaces */
#define ONOEOT          0x0008  /* discard EOT's (^D) on output) */

/* Extensions to the termios c_lflag bit map.  */
#define LFLUSHO         0x0200  /* Flush output. */

/* Extensions to the c_cc array. */
#define VREPRINT          11    /* cc_c[VREPRINT] (^R) */
#define VLNEXT            12    /* cc_c[VLNEXT] (^V) */
#define VDISCARD          13    /* cc_c[VDISCARD] (^O) */

/* Extensions to baud rate settings. */
#define B57600          0x0100  /* 57600 baud */
#define B115200         0x0200  /* 115200 baud */

/* These are the default settings used by the kernel and by 'stty sane' */
```

```
01270   #define TCTRL_DEF       (CREAD | CS8 | HUPCL)
01271   #define TINPUT_DEF      (BRKINT | ICRNL | IXON | IXANY)
01272   #define TOUTPUT_DEF     (OPOST | ONLCR)
01273   #define TLOCAL_DEF      (ISIG | IEXTEN | ICANON | ECHO | ECHOE)
01274   #define TSPEED_DEF      B9600
01275
01276   #define TEOF_DEF        '\4'    /* ^D */
01277   #define TEOL_DEF        _POSIX_VDISABLE
01278   #define TERASE_DEF      '\10'   /* ^H */
01279   #define TINTR_DEF       '\177'  /* ^? */
01280   #define TKILL_DEF       '\25'   /* ^U */
01281   #define TMIN_DEF        1
01282   #define TQUIT_DEF       '\34'   /* ^\ */
01283   #define TSTART_DEF      '\21'   /* ^Q */
01284   #define TSTOP_DEF       '\23'   /* ^S */
01285   #define TSUSP_DEF       '\32'   /* ^Z */
01286   #define TTIME_DEF       0
01287   #define TREPRINT_DEF    '\22'   /* ^R */
01288   #define TLNEXT_DEF      '\26'   /* ^V */
01289   #define TDISCARD_DEF    '\17'   /* ^O */
01290
01291   /* Window size. This information is stored in the TTY driver but not used.
01292    * This can be used for screen based applications in a window environment.
01293    * The ioctls TIOCGWINSZ and TIOCSWINSZ can be used to get and set this
01294    * information.
01295    */
01296
01297   struct winsize
01298   {
01299           unsigned short  ws_row;         /* rows, in characters */
01300           unsigned short  ws_col;         /* columns, in characters */
01301           unsigned short  ws_xpixel;      /* horizontal size, pixels */
01302           unsigned short  ws_ypixel;      /* vertical size, pixels */
01303   };
01304   #endif /* _MINIX */
01305
01306   #endif /* _TERMIOS_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                              include/a.out.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

01400   /* The <a.out> header file describes the format of executable files. */
01401
01402   #ifndef _AOUT_H
01403   #define _AOUT_H
01404
01405   struct  exec {                  /* a.out header */
01406     unsigned char a_magic[2];     /* magic number */
01407     unsigned char a_flags;        /* flags, see below */
01408     unsigned char a_cpu;          /* cpu id */
01409     unsigned char a_hdrlen;       /* length of header */
01410     unsigned char a_unused;       /* reserved for future use */
01411     unsigned short a_version;     /* version stamp (not used at present) */
01412     long          a_text;         /* size of text segement in bytes */
01413     long          a_data;         /* size of data segment in bytes */
01414     long          a_bss;          /* size of bss segment in bytes */
```



```
01415     long          a_entry;        /* entry point */
01416     long          a_total;        /* total memory allocated */
01417     long          a_syms;         /* size of symbol table */
01418
01419     /* SHORT FORM ENDS HERE */
01420     long          a_trsize;       /* text relocation size */
01421     long          a_drsize;       /* data relocation size */
01422     long          a_tbase;        /* text relocation base */
01423     long          a_dbase;        /* data relocation base */
01424   };
01425
01426   #define A_MAGIC0      (unsigned char) 0x01
01427   #define A_MAGIC1      (unsigned char) 0x03
01428   #define BADMAG(X)     ((X).a_magic[0] != A_MAGIC0 ||(X).a_magic[1] != A_MAGIC1)
01429
01430   /* CPU Id of TARGET machine (byte order coded in low order two bits) */
01431   #define A_NONE  0x00    /* unknown */
01432   #define A_I8086 0x04    /* intel i8086/8088 */
01433   #define A_M68K  0x0B    /* motorola m68000 */
01434   #define A_NS16K 0x0C    /* national semiconductor 16032 */
01435   #define A_I80386 0x10   /* intel i80386 */
01436   #define A_SPARC 0x17    /* Sun SPARC */
01437
01438   #define A_BLR(cputype)  ((cputype&0x01)!=0) /* TRUE if bytes left-to-right */
01439   #define A_WLR(cputype)  ((cputype&0x02)!=0) /* TRUE if words left-to-right */
01440
01441   /* Flags. */
01442   #define A_UZP   0x01    /* unmapped zero page (pages) */
01443   #define A_PAL   0x02    /* page aligned executable */
01444   #define A_NSYM  0x04    /* new style symbol table */
01445   #define A_EXEC  0x10    /* executable */
01446   #define A_SEP   0x20    /* separate I/D */
01447   #define A_PURE  0x40    /* pure text */         /* not used */
01448   #define A_TOVLY 0x80    /* text overlay */      /* not used */
01449
01450   /* Offsets of various things. */
01451   #define A_MINHDR        32
01452   #define A_TEXTPOS(X)    ((long)(X).a_hdrlen)
01453   #define A_DATAPOS(X)    (A_TEXTPOS(X) + (X).a_text)
01454   #define A_HASRELS(X)    ((X).a_hdrlen > (unsigned char) A_MINHDR)
01455   #define A_HASEXT(X)     ((X).a_hdrlen > (unsigned char) (A_MINHDR +  8))
01456   #define A_HASLNS(X)     ((X).a_hdrlen > (unsigned char) (A_MINHDR + 16))
01457   #define A_HASTOFF(X)    ((X).a_hdrlen > (unsigned char) (A_MINHDR + 24))
01458   #define A_TRELPOS(X)    (A_DATAPOS(X) + (X).a_data)
01459   #define A_DRELPOS(X)    (A_TRELPOS(X) + (X).a_trsize)
01460   #define A_SYMPOS(X)     (A_TRELPOS(X) + (A_HASRELS(X) ? \
01461                           ((X).a_trsize + (X).a_drsize) : 0))
01462
01463   struct reloc {
01464     long r_vaddr;                 /* virtual address of reference */
01465     unsigned short r_symndx;      /* internal segnum or extern symbol num */
01466     unsigned short r_type;        /* relocation type */
01467   };
01468
01469   /* r_tyep values: */
01470   #define R_ABBS          0
01471   #define R_RELLBYTE      2
01472   #define R_PCRBYTE       3
01473   #define R_RELWORD       4
01474   #define R_PCRWORD       5
```

```
01475   #define R_RELLONG       6
01476   #define R_PCRLONG       7
01477   #define R_REL3BYTE      8
01478   #define R_KBRANCHE      9
01479
01480   /* r_symndx for internal segments */
01481   #define S_ABS           ((unsigned short)-1)
01482   #define S_TEXT          ((unsigned short)-2)
01483   #define S_DATA          ((unsigned short)-3)
01484   #define S_BSS           ((unsigned short)-4)
01485
01486   struct nlist {                  /* symbol table entry */
01487     char n_name[8];               /* symbol name */
01488     long n_value;                 /* value */
01489     unsigned char n_sclass;       /* storage class */
01490     unsigned char n_numaux;       /* number of auxiliary entries (not used) */
01491     unsigned short n_type;        /* language base and derived type (not used) */
01492   };
01493
01494   /* Low bits of storage class (section). */
01495   #define N_SECT            07    /* section mask */
01496   #define N_UNDF            00    /* undefined */
01497   #define N_ABS             01    /* absolute */
01498   #define N_TEXT            02    /* text */
01499   #define N_DATA            03    /* data */
01500   #define N_BSS             04    /* bss */
01501   #define N_COMM            05    /* (common) */
01502
01503   /* High bits of storage class. */
01504   #define N_CLASS         0370    /* storage class mask */
01505   #define C_NULL
01506   #define C_EXT           0020    /* external symbol */
01507   #define C_STAT          0030    /* static */
01508
01509   /* Function prototypes. */
01510   #ifndef _ANSI_H
01511   #include <ansi.h>
01512   #endif
01513
01514   _PROTOTYPE( int nlist, (char *_file, struct nlist *_nl)                );
01515
01516   #endif /* _AOUT_H */
```



```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                              include/sys/types.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

01600   /* The <sys/types.h> header contains important data type definitions.
01601    * It is considered good programming practice to use these definitions,
01602    * instead of the underlying base type.  By convention, all type names end
01603    * with _t.
01604    */
01605
01606   #ifndef _TYPES_H
01607   #define _TYPES_H
01608
01609   /* _ANSI is somehow used to determine whether or not the compiler is a
01610    * 16 bit compiler
01611    */
01612   #ifndef _ANSI
01613   #include <ansi.h>
01614   #endif
01615
01616   /* The type size_t holds all results of the sizeof operator.  At first glance,
01617    * it seems obvious that it should be an unsigned int, but this is not always
01618    * the case. For example, MINIX-ST (68000) has 32-bit pointers and 16-bit
01619    * integers. When one asks for the size of a 70K struct or array, the result
01620    * requires 17 bits to express, so size_t must be a long type.  The type
01621    * ssize_t is the signed version of size_t.
01622    */
01623   #ifndef _SIZE_T
01624   #define _SIZE_T
01625   typedef unsigned int size_t;
01626   #endif
01627
01628   #ifndef _SSIZE_T
01629   #define _SSIZE_T
01630   typedef int ssize_t;
01631   #endif
01632
01633   #ifndef _TIME_T
01634   #define _TIME_T
01635   typedef long time_t;              /* time in sec since 1 Jan 1970 0000 GMT */
01636   #endif
01637
01638   #ifndef _CLOCK_T
01639   #define _CLOCK_T
01640   typedef long clock_t;             /* unit for system accounting */
01641   #endif
01642
01643   #ifndef _SIGSET_T
01644   #define _SIGSET_T
01645   typedef unsigned long sigset_t;
01646   #endif
01647
01648   /* Types used in disk, inode, etc. data structures. */
01649   typedef short          dev_t;     /* holds (major|minor) device pair */
01650   typedef char           gid_t;     /* group id */
01651   typedef unsigned short ino_t;     /* i-node number */
01652   typedef unsigned short mode_t;    /* file type and permissions bits */
01653   typedef char         nlink_t;     /* number of links to a file */
01654   typedef unsigned long  off_t;     /* offset within a file */
01655   typedef int            pid_t;     /* process id (must be signed) */
01656   typedef short          uid_t;     /* user id */
01657   typedef unsigned long zone_t;     /* zone number */
01658   typedef unsigned long block_t;    /* block number */
01659   typedef unsigned long  bit_t;     /* bit number in a bit map */
01660   typedef unsigned short zone1_t;   /* zone number for V1 file systems */
01661   typedef unsigned short bitchunk_t; /* collection of bits in a bitmap */
01662
01663   typedef unsigned char   u8_t;     /* 8 bit type */
01664   typedef unsigned short u16_t;     /* 16 bit type */
01665   typedef unsigned long  u32_t;     /* 32 bit type */
01666
01667   typedef char            i8_t;     /* 8 bit signed type */
01668   typedef short          i16_t;     /* 16 bit signed type */
01669   typedef long           i32_t;     /* 32 bit signed type */
```

```
01670
01671   /* The following types are needed because MINIX uses K&R style function
01672    * definitions (for maximum portability).  When a short, such as dev_t, is
01673    * passed to a function with a K&R definition, the compiler automatically
01674    * promotes it to an int.  The prototype must contain an int as the parameter,
01675    * not a short, because an int is what an old-style function definition
01676    * expects.  Thus using dev_t in a prototype would be incorrect.  It would be
01677    * sufficient to just use int instead of dev_t in the prototypes, but Dev_t
01678    * is clearer.
01679    */
01680   typedef int            Dev_t;
01681   typedef int            Gid_t;
01682   typedef int          Nlink_t;
01683   typedef int            Uid_t;
01684   typedef int             U8_t;
01685   typedef unsigned long  U32_t;
01686   typedef int             I8_t;
01687   typedef int            I16_t;
01688   typedef long            I32_t;
01689
01690   /* ANSI C makes writing down the promotion of unsigned types very messy.  When
01691    * sizeof(short) == sizeof(int), there is no promotion, so the type stays
01692    * unsigned.  When the compiler is not ANSI, there is usually no loss of
01693    * unsignedness, and there are usually no prototypes so the promoted type
01694    * doesn't matter.  The use of types like Ino_t is an attempt to use ints
01695    * (which are not promoted) while providing information to the reader.
01696    */
01697
01698   #ifndef _ANSI_H
01699   #include <ansi.h>
01700   #endif
01701
01702   #if _EM_WSIZE == 2 || !defined(_ANSI)
01703   typedef unsigned int      Ino_t;
01704   typedef unsigned int    Zone1_t;
01705   typedef unsigned int Bitchunk_t;
01706   typedef unsigned int      U16_t;
01707   typedef unsigned int  Mode_t;
01708
01709   #else /* _EM_WSIZE == 4, or _EM_WSIZE undefined, or _ANSI defined */
01710   typedef int               Ino_t;
01711   typedef int             Zone1_t;
01712   typedef int          Bitchunk_t;
01713   typedef int               U16_t;
01714   typedef int           Mode_t;
01715
01716   #endif /* _EM_WSIZE == 2, etc */
01717
01718   /* Signal handler type, e.g. SIG_IGN */
01719   #if defined(_ANSI)
01720   typedef void (*sighandler_t) (int);
01721   #else
01722   typedef void (*sighandler_t)();
01723   #endif
01724
01725   #endif /* _TYPES_H */
```



```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                include/sys/ioctl.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

01800   /* The ioctl.h header declares device controlling operations. */
01801
01802   #ifndef _IOCTL_H
01803   #define _IOCTL_H
01804
01805   #if _EM_WSIZE >= 4
01806   /* Ioctls have the command encoded in the low-order word, and the size
01807    * of the parameter in the high-order word. The 3 high bits of the high-
01808    * order word are used to encode the in/out/void status of the parameter.
01809    */
01810
01811   #define _IOCPARM_MASK   0x1FFF
01812   #define _IOC_VOID       0x20000000
01813   #define _IOCTYPE_MASK   0xFFFF
01814   #define _IOC_IN         0x40000000
01815   #define _IOC_OUT        0x80000000
01816   #define _IOC_INOUT      (_IOC_IN | _IOC_OUT)
01817
01818   #define _IO(x,y)        ((x << 8) | y | _IOC_VOID)
01819   #define _IOR(x,y,t)     ((x << 8) | y | ((sizeof(t) & _IOCPARM_MASK) << 16) |\
01820                                   _IOC_OUT)
01821   #define _IOW(x,y,t)     ((x << 8) | y | ((sizeof(t) & _IOCPARM_MASK) << 16) |\
01822                                   _IOC_IN)
01823   #define _IORW(x,y,t)    ((x << 8) | y | ((sizeof(t) & _IOCPARM_MASK) << 16) |\
01824                                   _IOC_INOUT)
01825   #else
01826   /* No fancy encoding on a 16-bit machine. */
01827
01828   #define _IO(x,y)        ((x << 8) | y)
01829   #define _IOR(x,y,t)     _IO(x,y)
01830   #define _IOW(x,y,t)     _IO(x,y)
01831   #define _IORW(x,y,t)    _IO(x,y)
01832   #endif
01833
01834
01835   /* Terminal ioctls. */
01836   #define TCGETS          _IOR('T',  8, struct termios) /* tcgetattr */
01837   #define TCSETS          _IOW('T',  9, struct termios) /* tcsetattr, TCSANOW */
01838   #define TCSETSW         _IOW('T', 10, struct termios) /* tcsetattr, TCSADRAIN */
01839   #define TCSETSF         _IOW('T', 11, struct termios) /* tcsetattr, TCSAFLUSH */
01840   #define TCSBRK          _IOW('T', 12, int)            /* tcsendbreak */
01841   #define TCDRAIN         _IO ('T', 13)                 /* tcdrain */
01842   #define TCFLOW          _IOW('T', 14, int)            /* tcflow */
01843   #define TCFLSH          _IOW('T', 15, int)            /* tcflush */
01844   #define TIOCGWINSZ      _IOR('T', 16, struct winsize)
01845   #define TIOCSWINSZ      _IOW('T', 17, struct winsize)
01846   #define TIOCGPGRP       _IOW('T', 18, int)
01847   #define TIOCSPGRP       _IOW('T', 19, int)
01848   #define TIOCSFON        _IOW('T', 20, u8_t [8192])
01849
01850   #define TIOCGETP        _IOR('t',  1, struct sgttyb)
01851   #define TIOCSETP        _IOW('t',  2, struct sgttyb)
01852   #define TIOCGETC        _IOR('t',  3, struct tchars)
01853   #define TIOCSETC        _IOW('t',  4, struct tchars)
01854
```

```
01855
01856   /* Network ioctls. */
01857   #define NWIOSETHOPT     _IOW('n', 16, struct nwio_ethopt)
01858   #define NWIOGETHOPT     _IOR('n', 17, struct nwio_ethopt)
01859   #define NWIOGETHSTAT    _IOR('n', 18, struct nwio_ethstat)
01860
01861   #define NWIOSIPCONF     _IOW('n', 32, struct nwio_ipconf)
01862   #define NWIOGIPCONF     _IOR('n', 33, struct nwio_ipconf)
01863   #define NWIOSIPOPT      _IOW('n', 34, struct nwio_ipopt)
01864   #define NWIOGIPOPT      _IOR('n', 35, struct nwio_ipopt)
01865
01866   #define NWIOIPGROUTE    _IORW('n', 40, struct nwio_route)
01867   #define NWIOIPSROUTE    _IOW ('n', 41, struct nwio_route)
01868   #define NWIOIPDROUTE    _IOW ('n', 42, struct nwio_route)
01869
01870   #define NWIOSTCPCONF    _IOW('n', 48, struct nwio_tcpconf)
01871   #define NWIOGTCPCONF    _IOR('n', 49, struct nwio_tcpconf)
01872   #define NWIOTCPCONN     _IOW('n', 50, struct nwio_tcpcl)
01873   #define NWIOTCPLISTEN   _IOW('n', 51, struct nwio_tcpcl)
01874   #define NWIOTCPATTACH   _IOW('n', 52, struct nwio_tcpatt)
01875   #define NWIOTCPSHUTDOWN _IO ('n', 53)
01876   #define NWIOSTCPOPT     _IOW('n', 54, struct nwio_tcpopt)
01877   #define NWIOGTCPOPT     _IOR('n', 55, struct nwio_tcpopt)
01878
01879   #define NWIOSUDPOPT     _IOW('n', 64, struct nwio_udpopt)
01880   #define NWIOGUDPOPT     _IOR('n', 65, struct nwio_udpopt)
01881
01882   /* Disk ioctls. */
01883   #define DIOCEJECT       _IO ('d', 5)
01884   #define DIOCSETP        _IOW('d', 6, struct partition)
01885   #define DIOCGETP        _IOR('d', 7, struct partition)
01886
01887   /* Keyboard ioctls. */
01888   #define KIOCSMAP        _IOW('k', 3, keymap_t)
01889
01890   /* Memory ioctls. */
01891   #define MIOCRAMSIZE     _IOW('m', 3, u32_t)     /* Size of the ramdisk */
01892   #define MIOCSPSINFO     _IOW('m', 4, void *)
01893   #define MIOCGPSINFO     _IOR('m', 5, struct psinfo)
01894
01895   /* Magnetic tape ioctls. */
01896   #define MTIOCTOP        _IOW('M', 1, struct mtop)
01897   #define MTIOCGET        _IOR('M', 2, struct mtget)
01898
01899   /* SCSI command. */
01900   #define SCIOCCMD        _IOW('S', 1, struct scsicmd)
01901
01902   /* CD-ROM ioctls. */
01903   #define CDIOPLAYTI      _IOR('c', 1, struct cd_play_track)
01904   #define CDIOPLAYMSS     _IOR('c', 2, struct cd_play_mss)
01905   #define CDIOREADTOCHDR  _IOW('c', 3, struct cd_toc_entry)
01906   #define CDIOREADTOC     _IOW('c', 4, struct cd_toc_entry)
01907   #define CDIOREADSUBCH   _IOW('c', 5, struct cd_toc_entry)
01908   #define CDIOSTOP        _IO ('c', 10)
01909   #define CDIOPAUSE       _IO ('c', 11)
01910   #define CDIORESUME      _IO ('c', 12)
01911   #define CDIOEJECT       DIOCEJECT
01912
01913   /* Soundcard DSP ioctls. */
01914   #define DSPIORATE       _IOR('s', 1, unsigned int)
```



```
01915   #define DSPIOSTEREO     _IOR('s', 2, unsigned int)
01916   #define DSPIOSIZE       _IOR('s', 3, unsigned int)
01917   #define DSPIOBITS       _IOR('s', 4, unsigned int)
01918   #define DSPIOSIGN       _IOR('s', 5, unsigned int)
01919   #define DSPIOMAX        _IOW('s', 6, unsigned int)
01920   #define DSPIORESET      _IO ('s', 7)
01921
01922   /* Soundcard mixer ioctls. */
01923   #define MIXIOGETVOLUME          _IORW('s', 10, struct volume_level)
01924   #define MIXIOGETINPUTLEFT       _IORW('s', 11, struct inout_ctrl)
01925   #define MIXIOGETINPUTRIGHT      _IORW('s', 12, struct inout_ctrl)
01926   #define MIXIOGETOUTPUT          _IORW('s', 13, struct inout_ctrl)
01927   #define MIXIOSETVOLUME          _IORW('s', 20, struct volume_level)
01928   #define MIXIOSETINPUTLEFT       _IORW('s', 21, struct inout_ctrl)
01929   #define MIXIOSETINPUTRIGHT      _IORW('s', 22, struct inout_ctrl)
01930   #define MIXIOSETOUTPUT          _IORW('s', 23, struct inout_ctrl)
01931
01932   #ifndef _ANSI
01933   #include <ansi.h>
01934   #endif
01935
01936   _PROTOTYPE( int ioctl, (int _fd, int _request, void *_data)             );
01937
01938   #endif /* _IOCTL_H */
```

++

include/sys/sigcontext.h

++

```
02000   #ifndef _SIGCONTEXT_H
02001   #define _SIGCONTEXT_H
02002
02003   /* The sigcontext structure is used by the sigreturn(2) system call.
02004    * sigreturn() is seldom called by user programs, but it is used internally
02005    * by the signal catching mechanism.
02006    */
02007
02008   #ifndef _ANSI_H
02009   #include <ansi.h>
02010   #endif
02011
02012   #ifndef _CONFIG_H
02013   #include <minix/config.h>
02014   #endif
02015
02016   #if !defined(CHIP)
02017   #include "error, configuration is not known"
02018   #endif
02019
02020   /* The following structure should match the stackframe_s structure used
02021    * by the kernel's context switching code.  Floating point registers should
02022    * be added in a different struct.
02023    */
02024   #if (CHIP == INTEL)
02025   struct sigregs {
02026   #if _WORD_SIZE == 4
02027     short sr_gs;
02028     short sr_fs;
02029   #endif /* _WORD_SIZE == 4 */
```

```
02030     short sr_es;
02031     short sr_ds;
02032     int sr_di;
02033     int sr_si;
02034     int sr_bp;
02035     int sr_st;                    /* stack top -- used in kernel */
02036     int sr_bx;
02037     int sr_dx;
02038     int sr_cx;
02039     int sr_retreg;
02040     int sr_retadr;                /* return address to caller of save -- used
02041                                    * in kernel */
02042     int sr_pc;
02043     int sr_cs;
02044     int sr_psw;
02045     int sr_sp;
02046     int sr_ss;
02047   };
02048
02049   struct sigframe {               /* stack frame created for signalled process */
02050     _PROTOTYPE( void (*sf_retadr), (void) );
02051     int sf_signo;
02052     int sf_code;
02053     struct sigcontext *sf_scp;
02054     int sf_fp;
02055     _PROTOTYPE( void (*sf_retadr2), (void) );
02056     struct sigcontext *sf_scpcopy;
02057   };
02058
02059   #else
02060   #if (CHIP == M68000)
02061   struct sigregs {
02062     long sr_retreg;                     /* d0 */
02063     long sr_d1;
02064     long sr_d2;
02065     long sr_d3;
02066     long sr_d4;
02067     long sr_d5;
02068     long sr_d6;
02069     long sr_d7;
02070     long sr_a0;
02071     long sr_a1;
02072     long sr_a2;
02073     long sr_a3;
02074     long sr_a4;
02075     long sr_a5;
02076     long sr_a6;
02077     long sr_sp;                   /* also known as a7 */
02078     long sr_pc;
02079     short sr_psw;
02080     short sr_dummy;               /* make size multiple of 4 for system.c */
02081   };
02082   #else
02083   #include "error, CHIP is not supported"
02084   #endif
02085   #endif /* CHIP == INTEL */
02086
02087   struct sigcontext {
02088     int sc_flags;                 /* sigstack state to restore */
02089     long sc_mask;                 /* signal mask to restore */
```



```
02090     struct sigregs sc_regs;       /* register set to restore */
02091   };
02092
02093   #if (CHIP == INTEL)
02094   #if _WORD_SIZE == 4
02095   #define sc_gs sc_regs.sr_gs
02096   #define sc_fs sc_regs.sr_fs
02097   #endif /* _WORD_SIZE == 4 */
02098   #define sc_es sc_regs.sr_es
02099   #define sc_ds sc_regs.sr_ds
02100   #define sc_di sc_regs.sr_di
02101   #define sc_si sc_regs.sr_si
02102   #define sc_fp sc_regs.sr_bp
02103   #define sc_st sc_regs.sr_st             /* stack top -- used in kernel */
02104   #define sc_bx sc_regs.sr_bx
02105   #define sc_dx sc_regs.sr_dx
02106   #define sc_cx sc_regs.sr_cx
02107   #define sc_retreg sc_regs.sr_retreg
02108   #define sc_retadr sc_regs.sr_retadr     /* return address to caller of
02109                                              save -- used in kernel */
02110   #define sc_pc sc_regs.sr_pc
02111   #define sc_cs sc_regs.sr_cs
02112   #define sc_psw sc_regs.sr_psw
02113   #define sc_sp sc_regs.sr_sp
02114   #define sc_ss sc_regs.sr_ss
02115   #endif /* CHIP == INTEL */
02116
02117   #if (CHIP == M68000)
02118   #define sc_retreg sc_regs.sr_retreg
02119   #define sc_d1 sc_regs.sr_d1
02120   #define sc_d2 sc_regs.sr_d2
02121   #define sc_d3 sc_regs.sr_d3
02122   #define sc_d4 sc_regs.sr_d4
02123   #define sc_d5 sc_regs.sr_d5
02124   #define sc_d6 sc_regs.sr_d6
02125   #define sc_d7 sc_regs.sr_d7
02126   #define sc_a0 sc_regs.sr_a0
02127   #define sc_a1 sc_regs.sr_a1
02128   #define sc_a2 sc_regs.sr_a2
02129   #define sc_a3 sc_regs.sr_a3
02130   #define sc_a4 sc_regs.sr_a4
02131   #define sc_a5 sc_regs.sr_a5
02132   #define sc_fp sc_regs.sr_a6
02133   #define sc_sp sc_regs.sr_sp
02134   #define sc_pc sc_regs.sr_pc
02135   #define sc_psw sc_regs.sr_psw
02136   #endif /* CHIP == M68000 */
02137
02138   /* Values for sc_flags.  Must agree with <minix/jmp_buf.h>. */
02139   #define SC_SIGCONTEXT   2       /* nonzero when signal context is included */
02140   #define SC_NOREGLOCALS  4       /* nonzero when registers are not to be
02141                                      saved and restored */
02142
02143   _PROTOTYPE( int sigreturn, (struct sigcontext *_scp)                    );
02144
02145   #endif /* _SIGCONTEXT_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                              include/sys/ptrace.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

02200    /* <sys/ptrace.h>
02201     * definitions for ptrace(2)
02202     */
02203
02204    #ifndef _PTRACE_H
02205    #define _PTRACE_H
02206
02207    #define T_STOP          -1      /* stop the process */
02208    #define T_OK             0      /* enable tracing by parent for this process */
02209    #define T_GETINS         1      /* return value from instruction space */
02210    #define T_GETDATA        2      /* return value from data space */
02211    #define T_GETUSER        3      /* return value from user process table */
02212    #define T_SETINS         4      /* set value from instruction space */
02213    #define T_SETDATA        5      /* set value from data space */
02214    #define T_SETUSER        6      /* set value in user process table */
02215    #define T_RESUME         7      /* resume execution */
02216    #define T_EXIT           8      /* exit */
02217    #define T_STEP           9      /* set trace bit */
02218
02219    /* Function Prototypes. */
02220    #ifndef _ANSI_H
02221    #include <ansi.h>
02222    #endif
02223
02224    _PROTOTYPE( long ptrace, (int _req, pid_t _pid, long _addr, long _data) );
02225
02226    #endif /* _PTRACE_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                              include/sys/stat.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

02300    /* The <sys/stat.h> header defines a struct that is used in the stat() and
02301     * fstat functions.  The information in this struct comes from the i-node of
02302     * some file.  These calls are the only approved way to inspect i-nodes.
02303     */
02304
02305    #ifndef _STAT_H
02306    #define _STAT_H
02307
02308    struct stat {
02309      dev_t st_dev;                 /* major/minor device number */
02310      ino_t st_ino;                 /* i-node number */
02311      mode_t st_mode;               /* file mode, protection bits, etc. */
02312      short int st_nlink;           /* # links; TEMPORARY HACK: should be nlink_t*/
02313      uid_t st_uid;                 /* uid of the file's owner */
02314      short int st_gid;             /* gid; TEMPORARY HACK: should be gid_t */
02315      dev_t st_rdev;
02316      off_t st_size;                /* file size */
02317      time_t st_atime;              /* time of last access */
02318      time_t st_mtime;              /* time of last data modification */
02319      time_t st_ctime;              /* time of last file status change */
```



```
02320    };
02321
02322    /* Traditional mask definitions for st_mode. */
02323    /* The ugly casts on only some of the definitions are to avoid suprising sign
02324     * extensions such as S_IFREG != (mode_t) S_IFREG when ints are 32 bits.
02325     */
02326    #define S_IFMT  ((mode_t) 0170000)      /* type of file */
02327    #define S_IFREG ((mode_t) 0100000)      /* regular */
02328    #define S_IFBLK 0060000         /* block special */
02329    #define S_IFDIR 0040000         /* directory */
02330    #define S_IFCHR 0020000         /* character special */
02331    #define S_IFIFO 0010000         /* this is a FIFO */
02332    #define S_ISUID 0004000         /* set user id on execution */
02333    #define S_ISGID 0002000         /* set group id on execution */
02334                                    /* next is reserved for future use */
02335    #define S_ISVTX   01000         /* save swapped text even after use */
02336
02337    /* POSIX masks for st_mode. */
02338    #define S_IRWXU   00700         /* owner:  rwx------ */
02339    #define S_IRUSR   00400         /* owner:  r-------- */
02340    #define S_IWUSR   00200         /* owner:  -w------- */
02341    #define S_IXUSR   00100         /* owner:  --x------ */
02342
02343    #define S_IRWXG   00070         /* group:  ---rwx--- */
02344    #define S_IRGRP   00040         /* group:  ---r----- */
02345    #define S_IWGRP   00020         /* group:  ----w---- */
02346    #define S_IXGRP   00010         /* group:  -----x--- */
02347
02348    #define S_IRWXO   00007         /* others: ------rwx */
02349    #define S_IROTH   00004         /* others: ------r-- */
02350    #define S_IWOTH   00002         /* others: -------w- */
02351    #define S_IXOTH   00001         /* others: --------x */
02352
02353    /* The following macros test st_mode (from POSIX Sec. 5.6.1.1). */
02354    #define S_ISREG(m)      (((m) & S_IFMT) == S_IFREG)     /* is a reg file */
02355    #define S_ISDIR(m)      (((m) & S_IFMT) == S_IFDIR)     /* is a directory */
02356    #define S_ISCHR(m)      (((m) & S_IFMT) == S_IFCHR)     /* is a char spec */
02357    #define S_ISBLK(m)      (((m) & S_IFMT) == S_IFBLK)     /* is a block spec */
02358    #define S_ISFIFO(m)     (((m) & S_IFMT) == S_IFIFO)     /* is a pipe/FIFO */
02359
02360
02361    /* Function Prototypes. */
02362    #ifndef _ANSI_H
02363    #include <ansi.h>
02364    #endif
02365
02366    _PROTOTYPE( int chmod, (const char *_path, Mode_t _mode)                );
02367    _PROTOTYPE( int fstat, (int _fildes, struct stat *_buf)                 );
02368    _PROTOTYPE( int mkdir, (const char *_path, Mode_t _mode)                );
02369    _PROTOTYPE( int mkfifo, (const char *_path, Mode_t _mode)               );
02370    _PROTOTYPE( int stat, (const char *_path, struct stat *_buf)            );
02371    _PROTOTYPE( mode_t umask, (Mode_t _cmask)                               );
02372
02373    #endif /* _STAT_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                              include/sys/dir.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

02400   /* The <dir.h> header gives the layout of a directory. */
02401
02402   #ifndef _DIR_H
02403   #define _DIR_H
02404
02405   #define DIRBLKSIZ       512     /* size of directory block */
02406
02407   #ifndef DIRSIZ
02408   #define DIRSIZ  14
02409   #endif
02410
02411   struct direct {
02412     ino_t d_ino;
02413     char d_name[DIRSIZ];
02414   };
02415
02416   #endif /* _DIR_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                              include/sys/wait.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

02500   /* The <sys/wait.h> header contains macros related to wait(). The value
02501    * returned by wait() and waitpid() depends on whether the process
02502    * terminated by an exit() call, was killed by a signal, or was stopped
02503    * due to job control, as follows:
02504    *
02505    *                               High byte   Low byte
02506    *                              +----------------------+
02507    *      exit(status)            |  status  |    0      |
02508    *                              +----------------------+
02509    *      killed by signal        |    0     |  signal   |
02510    *                              +----------------------+
02511    *      stopped (job control)   |  signal  |   0177    |
02512    *                              +----------------------+
02513    */
02514
02515   #ifndef _WAIT_H
02516   #define _WAIT_H
02517
02518   #define _LOW(v)         ( (v) & 0377)
02519   #define _HIGH(v)        ( ((v) >> 8) & 0377)
02520
02521   #define WNOHANG         1       /* do not wait for child to exit */
02522   #define WUNTRACED       2       /* for job control; not implemented */
02523
02524   #define WIFEXITED(s)    (_LOW(s) == 0)                  /* normal exit */
02525   #define WEXITSTATUS(s)  (_HIGH(s))                      /* exit status */
02526   #define WTERMSIG(s)     (_LOW(s) & 0177)                /* sig value */
02527   #define WIFSIGNALED(s)  (((unsigned int)(s)-1 & 0xFFFF) < 0xFF) /* signaled */
02528   #define WIFSTOPPED(s)   (_LOW(s) == 0177)               /* stopped */
02529   #define WSTOPSIG(s)     (_HIGH(s) & 0377)               /* stop signal */
```



```
02530
02531   /* Function Prototypes. */
02532   #ifndef _ANSI_H
02533   #include <ansi.h>
02534   #endif
02535
02536   _PROTOTYPE( pid_t wait, (int *_stat_loc)                                );
02537   _PROTOTYPE( pid_t waitpid, (pid_t _pid, int *_stat_loc, int _options)   );
02538
02539   #endif /* _WAIT_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                              include/minix/config.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

02600   #ifndef _CONFIG_H
02601   #define _CONFIG_H
02602
02603   /* Minix release and version numbers. */
02604   #define OS_RELEASE "2.0"
02605   #define OS_VERSION "0"
02606
02607   /* This file sets configuration parameters for the MINIX kernel, FS, and MM.
02608    * It is divided up into two main sections.  The first section contains
02609    * user-settable parameters.  In the second section, various internal system
02610    * parameters are set based on the user-settable parameters.
02611    */
02612
02613   /*===========================================================================*
02614    *              This section contains user-settable parameters               *
02615    *===========================================================================*/
02616   #define MACHINE       IBM_PC    /* Must be one of the names listed below */
02617
02618   #define IBM_PC             1    /* any  8088 or 80x86-based system */
02619   #define SUN_4             40    /* any Sun SPARC-based system */
02620   #define SUN_4_60          40    /* Sun-4/60 (aka SparcStation 1 or Campus) */
02621   #define ATARI             60    /* ATARI ST/STe/TT (68000/68030) */
02622   #define AMIGA             61    /* Commodore Amiga (68000) */
02623   #define MACINTOSH         62    /* Apple Macintosh (68000) */
02624
02625   /* Word size in bytes (a constant equal to sizeof(int)). */
02626   #if __ACK__
02627   #define _WORD_SIZE      _EM_WSIZE
02628   #endif
02629
02630
02631   /* If ROBUST is set to 1, writes of i-node, directory, and indirect blocks
02632    * from the cache happen as soon as the blocks are modified.  This gives a more
02633    * robust, but slower, file system.  If it is set to 0, these blocks are not
02634    * given any special treatment, which may cause problems if the system crashes.
02635    */
02636   #define ROBUST             0    /* 0 for speed, 1 for robustness */
02637
02638   /* Number of slots in the process table for user processes. */
02639   #define NR_PROCS          32
```

```
02640
02641   /* The buffer cache should be made as large as you can afford. */
02642   #if (MACHINE == IBM_PC && _WORD_SIZE == 2)
02643   #define NR_BUFS           40    /* # blocks in the buffer cache */
02644   #define NR_BUF_HASH       64    /* size of buf hash table; MUST BE POWER OF 2*/
02645   #endif
02646
02647   #if (MACHINE == IBM_PC && _WORD_SIZE == 4)
02648   #define NR_BUFS          512    /* # blocks in the buffer cache */
02649   #define NR_BUF_HASH     1024    /* size of buf hash table; MUST BE POWER OF 2*/
02650   #endif
02651
02652   #if (MACHINE == SUN_4_60)
02653   #define NR_BUFS          512    /* # blocks in the buffer cache (<=1536) */
02654   #define NR_BUF_HASH      512    /* size of buf hash table; MUST BE POWER OF 2*/
02655   #endif
02656
02657   #if (MACHINE == ATARI)
02658   #define NR_BUFS         1536    /* # blocks in the buffer cache (<=1536) */
02659   #define NR_BUF_HASH     2048    /* size of buf hash table; MUST BE POWER OF 2*/
02660   #endif
02661
02662   /* Defines for kernel configuration. */
02663   #define AUTO_BIOS          0    /* xt_wini.c - use Western's autoconfig BIOS */
02664   #define LINEWRAP           1    /* console.c - wrap lines at column 80 */
02665   #define ALLOW_GAP_MESSAGES 1    /* proc.c - allow messages in the gap between
02666                                    * the end of bss and lowest stack address */
02667
02668   /* Enable or disable the second level file system cache on the RAM disk. */
02669   #define ENABLE_CACHE2      0
02670
02671   /* Include or exclude device drivers.  Set to 1 to include, 0 to exclude. */
02672   #define ENABLE_NETWORKING  0    /* enable TCP/IP code */
02673   #define ENABLE_AT_WINI     1    /* enable AT winchester driver */
02674   #define ENABLE_BIOS_WINI   0    /* enable BIOS winchester driver */
02675   #define ENABLE_ESDI_WINI   0    /* enable ESDI winchester driver */
02676   #define ENABLE_XT_WINI     0    /* enable XT winchester driver */
02677   #define ENABLE_ADAPTEC_SCSI 0   /* enable ADAPTEC SCSI driver */
02678   #define ENABLE_MITSUMI_CDROM 0  /* enable Mitsumi CD-ROM driver */
02679   #define ENABLE_SB_AUDIO    0    /* enable Soundblaster audio driver */
02680
02681   /* DMA_SECTORS may be increased to speed up DMA based drivers. */
02682   #define DMA_SECTORS        1    /* DMA buffer size (must be >= 1) */
02683
02684   /* Include or exclude backwards compatibility code. */
02685   #define ENABLE_BINCOMPAT   0    /* for binaries using obsolete calls */
02686   #define ENABLE_SRCCOMPAT   0    /* for sources using obsolete calls */
02687
02688   /* Determine which device to use for pipes. */
02689   #define PIPE_DEV    ROOT_DEV    /* put pipes on root device */
02690
02691   /* NR_CONS, NR_RS_LINES, and NR_PTYS determine the number of terminals the
02692    * system can handle.
02693    */
02694   #define NR_CONS            2    /* # system consoles (1 to 8) */
02695   #define NR_RS_LINES        0    /* # rs232 terminals (0, 1, or 2) */
02696   #define NR_PTYS            0    /* # pseudo terminals (0 to 64) */
02697
02698   #if (MACHINE == ATARI)
02699   /* The next define says if you have an ATARI ST or TT */
```



```
02700   #define ATARI_TYPE        TT
02701   #define ST                 1    /* all ST's and Mega ST's */
02702   #define STE                2    /* all STe and Mega STe's */
02703   #define TT                 3
02704
02705   /* if SCREEN is set to 1 graphical screen operations are possible */
02706   #define SCREEN             1
02707
02708   /* This define says whether the keyboard generates VT100 or IBM_PC escapes. */
02709   #define KEYBOARD       VT100    /* either VT100 or IBM_PC */
02710   #define VT100            100
02711
02712   /* The next define determines the kind of partitioning. */
02713   #define PARTITIONING   SUPRA    /* one of the following or ATARI */
02714   #define SUPRA              1    /*ICD, SUPRA and BMS are all the same */
02715   #define BMS                1
02716   #define ICD                1
02717   #define CBHD               2
02718   #define EICKMANN           3
02719
02720   /* Define the number of hard disk drives on your system. */
02721   #define NR_ACSI_DRIVES     3    /* typically 0 or 1 */
02722   #define NR_SCSI_DRIVES     1    /* typically 0 (ST, STe) or 1 (TT) */
02723
02724   /* Some systems need to have a little delay after each winchester
02725    * commands. These systems need FAST_DISK set to 0. Other disks do not
02726    * need this delay, and thus can have FAST_DISK set to 1 to avoid this delay.
02727    */
02728   #define FAST_DISK          1    /* 0 or 1 */
02729
02730   /* Note: if you want to make your kernel smaller, you can set NR_FD_DRIVES
02731    * to 0. You will still be able to boot minix.img from floppy. However, you
02732    * MUST fetch both the root and usr filesystem from a hard disk
02733    */
02734
02735   /* Define the number of floppy disk drives on your system. */
02736   #define NR_FD_DRIVES       1    /* 0, 1, 2 */
02737
02738   /* This configuration define controls parallel printer code. */
02739   #define PAR_PRINTER        1    /* disable (0) / enable (1) parallel printer */
02740
02741   /* This configuration define controls disk controller clock code. */
02742   #define HD_CLOCK           1    /* disable (0) / enable (1) hard disk clock */
02743
02744   #endif
02745
02746
02747   /*===========================================================================*
02748    *      There are no user-settable parameters after this line                *
02749    *===========================================================================*/
02750   /* Set the CHIP type based on the machine selected. The symbol CHIP is actually
02751    * indicative of more than just the CPU.  For example, machines for which
02752    * CHIP == INTEL are expected to have 8259A interrrupt controllers and the
02753    * other properties of IBM PC/XT/AT/386 types machines in general. */
02754   #define INTEL              1     /* CHIP type for PC, XT, AT, 386 and clones */
02755   #define M68000             2     /* CHIP type for Atari, Amiga, Macintosh    */
02756   #define SPARC              3     /* CHIP type for SUN-4 (e.g. SPARCstation)  */
02757
02758   /* Set the FP_FORMAT type based on the machine selected, either hw or sw    */
02759   #define FP_NONE            0     /* no floating point support                */
```

```
02760   #define FP_IEEE           1     /* conform IEEE floating point standard     */
02761
02762   #if (MACHINE == IBM_PC)
02763   #define CHIP          INTEL
02764   #define SHADOWING         0
02765   #define ENABLE_WINI     (ENABLE_AT_WINI || ENABLE_BIOS_WINI || \
02766                            ENABLE_ESDI_WINI || ENABLE_XT_WINI)
02767   #define ENABLE_SCSI     (ENABLE_ADAPTEC_SCSI)
02768   #define ENABLE_CDROM    (ENABLE_MITSUMI_CDROM)
02769   #define ENABLE_AUDIO    (ENABLE_SB_AUDIO)
02770   #endif
02771
02772   #if (MACHINE == ATARI) || (MACHINE == AMIGA) || (MACHINE == MACINTOSH)
02773   #define CHIP         M68000
02774   #define SHADOWING         1
02775   #endif
02776
02777   #if (MACHINE == SUN_4) || (MACHINE == SUN_4_60)
02778   #define CHIP          SPARC
02779   #define FP_FORMAT   FP_IEEE
02780   #define SHADOWING         0
02781   #endif
02782
02783   #if (MACHINE == ATARI) || (MACHINE == SUN_4)
02784   #define ASKDEV            1     /* ask for boot device */
02785   #define FASTLOAD          1     /* use multiple block transfers to init ram */
02786   #endif
02787
02788   #if (ATARI_TYPE == TT) /* and all other 68030's */
02789   #define FPP
02790   #undef SHADOWING
02791   #define SHADOWING 0
02792   #endif
02793
02794   #ifndef FP_FORMAT
02795   #define FP_FORMAT   FP_NONE
02796   #endif
02797
02798   /* The file buf.h uses MAYBE_WRITE_IMMED. */
02799   #if ROBUST
02800   #define MAYBE_WRITE_IMMED  WRITE_IMMED  /* slower but perhaps safer */
02801   #else
02802   #define MAYBE_WRITE_IMMED 0             /* faster */
02803   #endif
02804
02805   #ifndef MACHINE
02806   error "In <minix/config.h> please define MACHINE"
02807   #endif
02808
02809   #ifndef CHIP
02810   error "In <minix/config.h> please define MACHINE to have a legal value"
02811   #endif
02812
02813   #if (MACHINE == 0)
02814   error "MACHINE has incorrect value (0)"
02815   #endif
02816
02817   #endif /* _CONFIG_H */
```



```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                include/minix/const.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

02900   /* Copyright (C) 1997 by Prentice-Hall, Inc.  Permission is hereby granted
02901    * to redistribute the binary and source programs of this system for
02902    * educational or research purposes.  For other use, written permission from
02903    * Prentice-Hall is required.
02904    */
02905
02906   #define EXTERN        extern    /* used in *.h files */
02907   #define PRIVATE       static    /* PRIVATE x limits the scope of x */
02908   #define PUBLIC                  /* PUBLIC is the opposite of PRIVATE */
02909   #define FORWARD       static    /* some compilers require this to be 'static'*/
02910
02911   #define TRUE               1    /* used for turning integers into Booleans */
02912   #define FALSE              0    /* used for turning integers into Booleans */
02913
02914   #define HZ                60    /* clock freq (software settable on IBM-PC) */
02915   #define BLOCK_SIZE      1024    /* # bytes in a disk block */
02916   #define SUPER_USER (uid_t) 0    /* uid_t of superuser */
02917
02918   #define MAJOR              8    /* major device = (dev>>MAJOR) & 0377 */
02919   #define MINOR              0    /* minor device = (dev>>MINOR) & 0377 */
02920
02921   #define NULL     ((void *)0)    /* null pointer */
02922   #define CPVEC_NR          16    /* max # of entries in a SYS_VCOPY request */
02923   #define NR_IOREQS       MIN(NR_BUFS, 64)
02924                                   /* maximum number of entries in an iorequest */
02925
02926   #define NR_SEGS            3    /* # segments per process */
02927   #define T                  0    /* proc[i].mem_map[T] is for text */
02928   #define D                  1    /* proc[i].mem_map[D] is for data */
02929   #define S                  2    /* proc[i].mem_map[S] is for stack */
02930
02931   /* Process numbers of some important processes. */
02932   #define MM_PROC_NR         0    /* process number of memory manager */
02933   #define FS_PROC_NR         1    /* process number of file system */
02934   #define INET_PROC_NR       2    /* process number of the TCP/IP server */
02935   #define INIT_PROC_NR    (INET_PROC_NR + ENABLE_NETWORKING)
02936                                   /* init -- the process that goes multiuser */
02937   #define LOW_USER        (INET_PROC_NR + ENABLE_NETWORKING)
02938                                   /* first user not part of operating system */
02939
02940   /* Miscellaneous */
02941   #define BYTE            0377    /* mask for 8 bits */
02942   #define READING            0    /* copy data to user */
02943   #define WRITING            1    /* copy data from user */
02944   #define NO_NUM        0x8000    /* used as numerical argument to panic() */
02945   #define NIL_PTR   (char *) 0    /* generally useful expression */
02946   #define HAVE_SCATTERED_IO  1    /* scattered I/O is now standard */
02947
02948   /* Macros. */
02949   #define MAX(a, b)   ((a) > (b) ? (a) : (b))
02950   #define MIN(a, b)   ((a) < (b) ? (a) : (b))
02951
02952   /* Number of tasks. */
02953   #define NR_TASKS        (9 + ENABLE_WINI + ENABLE_SCSI + ENABLE_CDROM \
02954                           + ENABLE_NETWORKING + 2 * ENABLE_AUDIO)
```

```
02955
02956   /* Memory is allocated in clicks. */
02957   #if (CHIP == INTEL)
02958   #define CLICK_SIZE       256    /* unit in which memory is allocated */
02959   #define CLICK_SHIFT        8    /* log2 of CLICK_SIZE */
02960   #endif
02961
02962   #if (CHIP == SPARC) || (CHIP == M68000)
02963   #define CLICK_SIZE      4096    /* unit in which memory is alocated */
02964   #define CLICK_SHIFT       12    /* 2log of CLICK_SIZE */
02965   #endif
02966
02967   #define click_to_round_k(n) \
02968           ((unsigned) ((((unsigned long) (n) << CLICK_SHIFT) + 512) / 1024))
02969   #if CLICK_SIZE < 1024
02970   #define k_to_click(n) ((n) * (1024 / CLICK_SIZE))
02971   #else
02972   #define k_to_click(n) ((n) / (CLICK_SIZE / 1024))
02973   #endif
02974
02975   #define ABS             -999    /* this process means absolute memory */
02976
02977   /* Flag bits for i_mode in the inode. */
02978   #define I_TYPE          0170000 /* this field gives inode type */
02979   #define I_REGULAR       0100000 /* regular file, not dir or special */
02980   #define I_BLOCK_SPECIAL 0060000 /* block special file */
02981   #define I_DIRECTORY     0040000 /* file is a directory */
02982   #define I_CHAR_SPECIAL  0020000 /* character special file */
02983   #define I_NAMED_PIPE    0010000 /* named pipe (FIFO) */
02984   #define I_SET_UID_BIT   0004000 /* set effective uid_t on exec */
02985   #define I_SET_GID_BIT   0002000 /* set effective gid_t on exec */
02986   #define ALL_MODES       0006777 /* all bits for user, group and others */
02987   #define RWX_MODES       0000777 /* mode bits for RWX only */
02988   #define R_BIT           0000004 /* Rwx protection bit */
02989   #define W_BIT           0000002 /* rWx protection bit */
02990   #define X_BIT           0000001 /* rwX protection bit */
02991   #define I_NOT_ALLOC     0000000 /* this inode is free */
02992
02993   /* Some limits. */
02994   #define MAX_BLOCK_NR  ((block_t) 077777777)     /* largest block number */
02995   #define HIGHEST_ZONE   ((zone_t) 077777777)     /* largest zone number */
02996   #define MAX_INODE_NR      ((ino_t) 0177777)     /* largest inode number */
02997   #define MAX_FILE_POS ((off_t) 037777777777)     /* largest legal file offset */
02998
02999   #define NO_BLOCK              ((block_t) 0)     /* absence of a block number */
03000   #define NO_ENTRY                ((ino_t) 0)     /* absence of a dir entry */
03001   #define NO_ZONE                ((zone_t) 0)     /* absence of a zone number */
03002   #define NO_DEV                  ((dev_t) 0)     /* absence of a device numb */
```



```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                include/minix/type.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

03100   #ifndef _TYPE_H
03101   #define _TYPE_H
03102   #ifndef _MINIX_TYPE_H
03103   #define _MINIX_TYPE_H
03104
03105   /* Type definitions. */
03106   typedef unsigned int vir_clicks; /* virtual  addresses and lengths in clicks */
03107   typedef unsigned long phys_bytes;/* physical addresses and lengths in bytes */
03108   typedef unsigned int phys_clicks;/* physical addresses and lengths in clicks */
03109
03110   #if (CHIP == INTEL)
03111   typedef unsigned int vir_bytes; /* virtual addresses and lengths in bytes */
03112   #endif
03113
03114   #if (CHIP == M68000)
03115   typedef unsigned long vir_bytes;/* virtual addresses and lengths in bytes */
03116   #endif
03117
03118   #if (CHIP == SPARC)
03119   typedef unsigned long vir_bytes;/* virtual addresses and lengths in bytes */
03120   #endif
03121
03122   /* Types relating to messages. */
03123   #define M1                 1
03124   #define M3                 3
03125   #define M4                 4
03126   #define M3_STRING         14
03127
03128   typedef struct {int m1i1, m1i2, m1i3; char *m1p1, *m1p2, *m1p3;} mess_1;
03129   typedef struct {int m2i1, m2i2, m2i3; long m2l1, m2l2; char *m2p1;} mess_2;
03130   typedef struct {int m3i1, m3i2; char *m3p1; char m3ca1[M3_STRING];} mess_3;
03131   typedef struct {long m4l1, m4l2, m4l3, m4l4, m4l5;} mess_4;
03132   typedef struct {char m5c1, m5c2; int m5i1, m5i2; long m5l1, m5l2, m5l3;}mess_5;
03133   typedef struct {int m6i1, m6i2, m6i3; long m6l1; sighandler_t m6f1;} mess_6;
03134
03135   typedef struct {
03136     int m_source;                 /* who sent the message */
03137     int m_type;                   /* what kind of message is it */
03138     union {
03139           mess_1 m_m1;
03140           mess_2 m_m2;
03141           mess_3 m_m3;
03142           mess_4 m_m4;
03143           mess_5 m_m5;
03144           mess_6 m_m6;
03145     } m_u;
03146   } message;
03147
03148   /* The following defines provide names for useful members. */
03149   #define m1_i1  m_u.m_m1.m1i1
03150   #define m1_i2  m_u.m_m1.m1i2
03151   #define m1_i3  m_u.m_m1.m1i3
03152   #define m1_p1  m_u.m_m1.m1p1
03153   #define m1_p2  m_u.m_m1.m1p2
03154   #define m1_p3  m_u.m_m1.m1p3
```

```
03155
03156   #define m2_i1  m_u.m_m2.m2i1
03157   #define m2_i2  m_u.m_m2.m2i2
03158   #define m2_i3  m_u.m_m2.m2i3
03159   #define m2_l1  m_u.m_m2.m2l1
03160   #define m2_l2  m_u.m_m2.m2l2
03161   #define m2_p1  m_u.m_m2.m2p1
03162
03163   #define m3_i1  m_u.m_m3.m3i1
03164   #define m3_i2  m_u.m_m3.m3i2
03165   #define m3_p1  m_u.m_m3.m3p1
03166   #define m3_ca1 m_u.m_m3.m3ca1
03167
03168   #define m4_l1  m_u.m_m4.m4l1
03169   #define m4_l2  m_u.m_m4.m4l2
03170   #define m4_l3  m_u.m_m4.m4l3
03171   #define m4_l4  m_u.m_m4.m4l4
03172   #define m4_l5  m_u.m_m4.m4l5
03173
03174   #define m5_c1  m_u.m_m5.m5c1
03175   #define m5_c2  m_u.m_m5.m5c2
03176   #define m5_i1  m_u.m_m5.m5i1
03177   #define m5_i2  m_u.m_m5.m5i2
03178   #define m5_l1  m_u.m_m5.m5l1
03179   #define m5_l2  m_u.m_m5.m5l2
03180   #define m5_l3  m_u.m_m5.m5l3
03181
03182   #define m6_i1  m_u.m_m6.m6i1
03183   #define m6_i2  m_u.m_m6.m6i2
03184   #define m6_i3  m_u.m_m6.m6i3
03185   #define m6_l1  m_u.m_m6.m6l1
03186   #define m6_f1  m_u.m_m6.m6f1
03187
03188   struct mem_map {
03189     vir_clicks mem_vir;           /* virtual address */
03190     phys_clicks mem_phys;         /* physical address */
03191     vir_clicks mem_len;           /* length */
03192   };
03193
03194   struct iorequest_s {
03195     long io_position;             /* position in device file (really off_t) */
03196     char *io_buf;                 /* buffer in user space */
03197     int io_nbytes;                /* size of request */
03198     unsigned short io_request;    /* read, write (optionally) */
03199   };
03200   #endif /* _TYPE_H */
03201
03202   typedef struct {
03203     vir_bytes iov_addr;           /* address of an I/O buffer */
03204     vir_bytes iov_size;           /* sizeof an I/O buffer */
03205   } iovec_t;
03206
03207   typedef struct {
03208     vir_bytes cpv_src;            /* src address of data */
03209     vir_bytes cpv_dst;            /* dst address of data */
03210     vir_bytes cpv_size;           /* size of data */
03211   } cpvec_t;
03212
03213   /* MM passes the address of a structure of this type to KERNEL when
03214    * do_sendsig() is invoked as part of the signal catching mechanism.
```



```
03215    * The structure contain all the information that KERNEL needs to build
03216    * the signal stack.
03217    */
03218   struct sigmsg {
03219     int sm_signo;               /* signal number being caught */
03220     unsigned long sm_mask;      /* mask to restore when handler returns */
03221     vir_bytes sm_sighandler;    /* address of handler */
03222     vir_bytes sm_sigreturn;     /* address of _sigreturn in C library */
03223     vir_bytes sm_stkptr;        /* user stack pointer */
03224   };
03225
03226   #define MESS_SIZE (sizeof(message))     /* might need usizeof from fs here */
03227   #define NIL_MESS ((message *) 0)
03228
03229   struct psinfo {         /* information for the ps(1) program */
03230     u16_t nr_tasks, nr_procs;     /* NR_TASKS and NR_PROCS constants. */
03231     vir_bytes proc, mproc, fproc; /* addresses of the main process tables. */
03232   };
03233
03234   #endif /* _MINIX_TYPE_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                include/minix/syslib.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

03300   /* Prototypes for system library functions. */
03301
03302   #ifndef _SYSLIB_H
03303   #define _SYSLIB_H
03304
03305   /* Hide names to avoid name space pollution. */
03306   #define sendrec         _sendrec
03307   #define receive         _receive
03308   #define send            _send
03309
03310   /* Minix user+system library. */
03311   _PROTOTYPE( void printk, (char *_fmt, ...)                              );
03312   _PROTOTYPE( int sendrec, (int _src_dest, message *_m_ptr)               );
03313   _PROTOTYPE( int _taskcall, (int _who, int _syscallnr, message *_msgptr) );
03314
03315   /* Minix system library. */
03316   _PROTOTYPE( int receive, (int _src, message *_m_ptr)                    );
03317   _PROTOTYPE( int send, (int _dest, message *_m_ptr)                      );
03318
03319   _PROTOTYPE( int sys_abort, (int _how, ...)                              );
03320   _PROTOTYPE( int sys_adjmap, (int _proc, struct mem_map *_ptr,
03321                   vir_clicks _data_clicks, vir_clicks _sp)                );
03322   _PROTOTYPE( int sys_copy, (int _src_proc, int _src_seg, phys_bytes _src_vir,
03323           int _dst_proc, int _dst_seg, phys_bytes _dst_vir, phys_bytes _bytes));
03324   _PROTOTYPE( int sys_exec, (int _proc, char *_ptr, int _traced,
03325                                   char *_aout, vir_bytes _initpc)         );
03326   _PROTOTYPE( int sys_execmap, (int _proc, struct mem_map *_ptr)          );
03327   _PROTOTYPE( int sys_fork, (int _parent, int _child, int _pid,
03328                                   phys_clicks _shadow)                    );
03329   _PROTOTYPE( int sys_fresh, (int _proc, struct mem_map *_ptr,
```

```
03330           phys_clicks _dc, phys_clicks *_basep, phys_clicks *_sizep)      );
03331   _PROTOTYPE( int sys_getsp, (int _proc, vir_bytes *_newsp)               );
03332   _PROTOTYPE( int sys_newmap, (int _proc, struct mem_map *_ptr)           );
03333   _PROTOTYPE( int sys_getmap, (int _proc, struct mem_map *_ptr)           );
03334   _PROTOTYPE( int sys_sendsig, (int _proc, struct sigmsg *_ptr)           );
03335   _PROTOTYPE( int sys_oldsig, (int _proc, int _sig, sighandler_t _sighandler));
03336   _PROTOTYPE( int sys_endsig, (int _proc)                                 );
03337   _PROTOTYPE( int sys_sigreturn, (int _proc, vir_bytes _scp, int _flags)  );
03338   _PROTOTYPE( int sys_trace, (int _req, int _procnr, long _addr, long *_data_p));
03339   _PROTOTYPE( int sys_xit, (int _parent, int _proc, phys_clicks *_basep,
03340                                                 phys_clicks *_sizep));
03341   _PROTOTYPE( int sys_kill, (int _proc, int _sig)                         );
03342   _PROTOTYPE( int sys_times, (int _proc, clock_t _ptr[5])                 );
03343
03344   #endif /* _SYSLIB_H */
```

++

include/minix/callnr.h

++

```
03400   #define NCALLS            77    /* number of system calls allowed */
03401
03402   #define EXIT               1
03403   #define FORK               2
03404   #define READ               3
03405   #define WRITE              4
03406   #define OPEN               5
03407   #define CLOSE              6
03408   #define WAIT               7
03409   #define CREAT              8
03410   #define LINK               9
03411   #define UNLINK            10
03412   #define WAITPID           11
03413   #define CHDIR             12
03414   #define TIME              13
03415   #define MKNOD             14
03416   #define CHMOD             15
03417   #define CHOWN             16
03418   #define BRK               17
03419   #define STAT              18
03420   #define LSEEK             19
03421   #define GETPID            20
03422   #define MOUNT             21
03423   #define UMOUNT            22
03424   #define SETUID            23
03425   #define GETUID            24
03426   #define STIME             25
03427   #define PTRACE            26
03428   #define ALARM             27
03429   #define FSTAT             28
03430   #define PAUSE             29
03431   #define UTIME             30
03432   #define ACCESS            33
03433   #define SYNC              36
03434   #define KILL              37
```



```
03435   #define RENAME            38
03436   #define MKDIR             39
03437   #define RMDIR             40
03438   #define DUP               41
03439   #define PIPE              42
03440   #define TIMES             43
03441   #define SETGID            46
03442   #define GETGID            47
03443   #define SIGNAL            48
03444   #define IOCTL             54
03445   #define FCNTL             55
03446   #define EXEC              59
03447   #define UMASK             60
03448   #define CHROOT            61
03449   #define SETSID            62
03450   #define GETPGRP           63
03451
03452   /* The following are not system calls, but are processed like them. */
03453   #define KSIG              64    /* kernel detected a signal */
03454   #define UNPAUSE           65    /* to MM or FS: check for EINTR */
03455   #define REVIVE            67    /* to FS: revive a sleeping process */
03456   #define TASK_REPLY        68    /* to FS: reply code from tty task */
03457
03458   /* Posix signal handling. */
03459   #define SIGACTION         71
03460   #define SIGSUSPEND        72
03461   #define SIGPENDING        73
03462   #define SIGPROCMASK       74
03463   #define SIGRETURN         75
03464
03465   #define REBOOT            76
```

++

include/minix/com.h

++

```
03500   /* System calls. */
03501   #define SEND               1    /* function code for sending messages */
03502   #define RECEIVE            2    /* function code for receiving messages */
03503   #define BOTH               3    /* function code for SEND + RECEIVE */
03504   #define ANY   (NR_PROCS+100)    /* receive(ANY, buf) accepts from any source */
03505
03506   /* Task numbers, function codes and reply codes. */
03507
03508   /* The values of several task numbers depend on whether they or other tasks
03509    * are enabled.  They are defined as (PREVIOUS_TASK - ENABLE_TASK) in general.
03510    * ENABLE_TASK is either 0 or 1, so a task either gets a new number, or gets
03511    * the same number as the previous task and is further unused.
03512    * The TTY task must always have the most negative number so that it is
03513    * initialized first.  Many of the TTY function codes are shared with other
03514    * tasks.
03515    */
03516
03517   #define TTY             (DL_ETH - 1)
03518                                   /* terminal I/O class */
03519   #       define CANCEL       0   /* general req to force a task to cancel */
```

```
03520   #       define HARD_INT     2   /* fcn code for all hardware interrupts */
03521   #       define DEV_READ     3   /* fcn code for reading from tty */
03522   #       define DEV_WRITE    4   /* fcn code for writing to tty */
03523   #       define DEV_IOCTL    5   /* fcn code for ioctl */
03524   #       define DEV_OPEN     6   /* fcn code for opening tty */
03525   #       define DEV_CLOSE    7   /* fcn code for closing tty */
03526   #       define SCATTERED_IO 8   /* fcn code for multiple reads/writes */
03527   #       define TTY_SETPGRP  9   /* fcn code for setpgroup */
03528   #       define TTY_EXIT    10   /* a process group leader has exited */
03529   #       define OPTIONAL_IO 16   /* modifier to DEV_* codes within vector */
03530   #       define SUSPEND   -998   /* used in interrupts when tty has no data */
03531
03532   #define DL_ETH          (CDROM - ENABLE_NETWORKING)
03533                                   /* networking task */
03534
03535   /* Message type for data link layer reqests. */
03536   #       define DL_WRITE         3
03537   #       define DL_WRITEV        4
03538   #       define DL_READ          5
03539   #       define DL_READV         6
03540   #       define DL_INIT          7
03541   #       define DL_STOP          8
03542   #       define DL_GETSTAT       9
03543
03544   /* Message type for data link layer replies. */
03545   #       define DL_INIT_REPLY    20
03546   #       define DL_TASK_REPLY    21
03547
03548   #       define DL_PORT          m2_i1
03549   #       define DL_PROC          m2_i2
03550   #       define DL_COUNT         m2_i3
03551   #       define DL_MODE          m2_l1
03552   #       define DL_CLCK          m2_l2
03553   #       define DL_ADDR          m2_p1
03554   #       define DL_STAT          m2_l1
03555
03556   /* Bits in 'DL_STAT' field of DL replies. */
03557   #       define DL_PACK_SEND     0x01
03558   #       define DL_PACK_RECV     0x02
03559   #       define DL_READ_IP       0x04
03560
03561   /* Bits in 'DL_MODE' field of DL requests. */
03562   #       define DL_NOMODE        0x0
03563   #       define DL_PROMISC_REQ   0x2
03564   #       define DL_MULTI_REQ     0x4
03565   #       define DL_BROAD_REQ     0x8
03566
03567   #       define NW_OPEN          DEV_OPEN
03568   #       define NW_CLOSE         DEV_CLOSE
03569   #       define NW_READ          DEV_READ
03570   #       define NW_WRITE         DEV_WRITE
03571   #       define NW_IOCTL         DEV_IOCTL
03572   #       define NW_CANCEL        CANCEL
03573
03574   #define CDROM           (AUDIO - ENABLE_CDROM)
03575                                   /* cd-rom device task */
03576
03577   #define AUDIO           (MIXER - ENABLE_AUDIO)
03578   #define MIXER           (SCSI - ENABLE_AUDIO)
03579                                   /* audio & mixer device tasks */
```



```
03580
03581   #define SCSI            (WINCHESTER - ENABLE_SCSI)
03582                                   /* scsi device task */
03583
03584   #define WINCHESTER      (SYN_ALRM_TASK - ENABLE_WINI)
03585                                   /* winchester (hard) disk class */
03586
03587   #define SYN_ALRM_TASK     -8    /* task to send CLOCK_INT messages */
03588
03589   #define IDLE              -7    /* task to run when there's nothing to run */
03590
03591   #define PRINTER           -6    /* printer I/O class */
03592
03593   #define FLOPPY            -5    /* floppy disk class */
03594
03595   #define MEM               -4    /* /dev/ram, /dev/(k)mem and /dev/null class */
03596   #       define NULL_MAJOR  1    /* major device for /dev/null */
03597   #       define RAM_DEV     0    /* minor device for /dev/ram */
03598   #       define MEM_DEV     1    /* minor device for /dev/mem */
03599   #       define KMEM_DEV    2    /* minor device for /dev/kmem */
03600   #       define NULL_DEV    3    /* minor device for /dev/null */
03601
03602   #define CLOCK             -3    /* clock class */
03603   #       define SET_ALARM   1    /* fcn code to CLOCK, set up alarm */
03604   #       define GET_TIME    3    /* fcn code to CLOCK, get real time */
03605   #       define SET_TIME    4    /* fcn code to CLOCK, set real time */
03606   #       define GET_UPTIME  5    /* fcn code to CLOCK, get uptime */
03607   #       define SET_SYNC_AL 6    /* fcn code to CLOCK, set up alarm which */
03608                                   /* times out with a send */
03609   #       define REAL_TIME   1    /* reply from CLOCK: here is real time */
03610   #       define CLOCK_INT   HARD_INT
03611                                   /* this code will only be sent by */
03612                                   /* SYN_ALRM_TASK to a task that requested a */
03613                                   /* synchronous alarm */
03614
03615   #define SYSTASK           -2    /* internal functions */
03616   #       define SYS_XIT        1 /* fcn code for sys_xit(parent, proc) */
03617   #       define SYS_GETSP      2 /* fcn code for sys_sp(proc, &new_sp) */
03618   #       define SYS_OLDSIG     3 /* fcn code for sys_oldsig(proc, sig) */
03619   #       define SYS_FORK       4 /* fcn code for sys_fork(parent, child) */
03620   #       define SYS_NEWMAP     5 /* fcn code for sys_newmap(procno, map_ptr) */
03621   #       define SYS_COPY       6 /* fcn code for sys_copy(ptr) */
03622   #       define SYS_EXEC       7 /* fcn code for sys_exec(procno, new_sp) */
03623   #       define SYS_TIMES      8 /* fcn code for sys_times(procno, bufptr) */
03624   #       define SYS_ABORT      9 /* fcn code for sys_abort() */
03625   #       define SYS_FRESH     10 /* fcn code for sys_fresh()  (Atari only) */
03626   #       define SYS_KILL      11 /* fcn code for sys_kill(proc, sig) */
03627   #       define SYS_GBOOT     12 /* fcn code for sys_gboot(procno, bootptr) */
03628   #       define SYS_UMAP      13 /* fcn code for sys_umap(procno, etc) */
03629   #       define SYS_MEM       14 /* fcn code for sys_mem() */
03630   #       define SYS_TRACE     15 /* fcn code for sys_trace(req,pid,addr,data) */
03631   #       define SYS_VCOPY     16 /* fnc code for sys_vcopy(src_proc, dest_proc,
03632                                      vcopy_s, vcopy_ptr) */
03633   #       define SYS_SENDSIG   17 /* fcn code for sys_sendsig(&sigmsg) */
03634   #       define SYS_SIGRETURN 18 /* fcn code for sys_sigreturn(&sigmsg) */
03635   #       define SYS_ENDSIG    19 /* fcn code for sys_endsig(procno) */
03636   #       define SYS_GETMAP    20 /* fcn code for sys_getmap(procno, map_ptr) */
03637
03638   #define HARDWARE          -1    /* used as source on interrupt generated msgs*/
03639
```

```
03640   /* Names of message fields for messages to CLOCK task. */
03641   #define DELTA_TICKS    m6_l1    /* alarm interval in clock ticks */
03642   #define FUNC_TO_CALL   m6_f1    /* pointer to function to call */
03643   #define NEW_TIME       m6_l1    /* value to set clock to (SET_TIME) */
03644   #define CLOCK_PROC_NR  m6_i1    /* which proc (or task) wants the alarm? */
03645   #define SECONDS_LEFT   m6_l1    /* how many seconds were remaining */
03646
03647   /* Names of message fields used for messages to block and character tasks. */
03648   #define DEVICE         m2_i1    /* major-minor device */
03649   #define PROC_NR        m2_i2    /* which (proc) wants I/O? */
03650   #define COUNT          m2_i3    /* how many bytes to transfer */
03651   #define REQUEST        m2_i3    /* ioctl request code */
03652   #define POSITION       m2_l1    /* file offset */
03653   #define ADDRESS        m2_p1    /* core buffer address */
03654
03655   /* Names of message fields for messages to TTY task. */
03656   #define TTY_LINE       DEVICE   /* message parameter: terminal line */
03657   #define TTY_REQUEST    COUNT    /* message parameter: ioctl request code */
03658   #define TTY_SPEK       POSITION /* message parameter: ioctl speed, erasing */
03659   #define TTY_FLAGS      m2_l2    /* message parameter: ioctl tty mode */
03660   #define TTY_PGRP       m2_i3    /* message parameter: process group */
03661
03662   /* Names of the message fields for QIC 02 status reply from tape driver */
03663   #define TAPE_STAT0      m2_l1
03664   #define TAPE_STAT1      m2_l2
03665
03666   /* Names of messages fields used in reply messages from tasks. */
03667   #define REP_PROC_NR    m2_i1    /* # of proc on whose behalf I/O was done */
03668   #define REP_STATUS     m2_i2    /* bytes transferred or error number */
03669
03670   /* Names of fields for copy message to SYSTASK. */
03671   #define SRC_SPACE      m5_c1    /* T or D space (stack is also D) */
03672   #define SRC_PROC_NR    m5_i1    /* process to copy from */
03673   #define SRC_BUFFER     m5_l1    /* virtual address where data come from */
03674   #define DST_SPACE      m5_c2    /* T or D space (stack is also D) */
03675   #define DST_PROC_NR    m5_i2    /* process to copy to */
03676   #define DST_BUFFER     m5_l2    /* virtual address where data go to */
03677   #define COPY_BYTES     m5_l3    /* number of bytes to copy */
03678
03679   /* Field names for accounting, SYSTASK and miscellaneous. */
03680   #define USER_TIME      m4_l1    /* user time consumed by process */
03681   #define SYSTEM_TIME    m4_l2    /* system time consumed by process */
03682   #define CHILD_UTIME    m4_l3    /* user time consumed by process' children */
03683   #define CHILD_STIME    m4_l4    /* sys time consumed by process' children */
03684   #define BOOT_TICKS     m4_l5    /* number of clock ticks since boot time */
03685
03686   #define PROC1          m1_i1    /* indicates a process */
03687   #define PROC2          m1_i2    /* indicates a process */
03688   #define PID            m1_i3    /* process id passed from MM to kernel */
03689   #define STACK_PTR      m1_p1    /* used for stack ptr in sys_exec, sys_getsp */
03690   #define PR             m6_i1    /* process number for sys_sig */
03691   #define SIGNUM         m6_i2    /* signal number for sys_sig */
03692   #define FUNC           m6_f1    /* function pointer for sys_sig */
03693   #define MEM_PTR        m1_p1    /* tells where memory map is for sys_newmap */
03694   #define NAME_PTR       m1_p2    /* tells where program name is for dmp */
03695   #define IP_PTR         m1_p3    /* initial value for ip after exec */
03696   #define SIG_PROC       m2_i1    /* process number for inform */
03697   #define SIG_MAP        m2_l1    /* used by kernel for passing signal bit map */
03698   #define SIG_MSG_PTR    m1_i1    /* pointer to info to build sig catch stack */
03699   #define SIG_CTXT_PTR   m1_p1    /* pointer to info to restore signal context */
```



```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                include/minix/boot.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

03700   /* boot.h */
03701
03702   #ifndef _BOOT_H
03703   #define _BOOT_H
03704
03705   /* Redefine root and root image devices as variables.
03706    * This keeps the diffs small but may cause future confusion.
03707    */
03708   #define ROOT_DEV   (boot_parameters.bp_rootdev)
03709   #define IMAGE_DEV  (boot_parameters.bp_ramimagedev)
03710
03711   /* Device numbers of RAM, floppy and hard disk devices.
03712    * h/com.h defines RAM_DEV but only as the minor number.
03713    */
03714   #define DEV_FD0   0x200
03715   #define DEV_HD0   0x300
03716   #define DEV_RAM   0x100
03717   #define DEV_SCSI  0x700 /* Atari TT only */
03718
03719   /* Structure to hold boot parameters. */
03720   struct bparam_s
03721   {
03722     dev_t bp_rootdev;
03723     dev_t bp_ramimagedev;
03724     unsigned short bp_ramsize;
03725     unsigned short bp_processor;
03726   };
03727
03728   extern struct bparam_s boot_parameters;
03729   #endif /* _BOOT_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                include/minix/keymap.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

03800   /*      keymap.h - defines for keymapping              Author: Marcus Hampel
03801    */
03802   #ifndef _SYS__KEYMAP_H
03803   #define _SYS__KEYMAP_H
03804
03805   #define C(c)     ((c) & 0x1F)    /* Map to control code          */
03806   #define A(c)     ((c) | 0x80)    /* Set eight bit (ALT)          */
03807   #define CA(c)    A(C(c))         /* Control-Alt                  */
03808   #define L(c)     ((c) | HASCAPS) /* Add "Caps Lock has effect" attribute */
03809
03810   #define EXT     0x0100          /* Normal function keys         */
03811   #define CTRL    0x0200          /* Control key                  */
03812   #define SHIFT   0x0400          /* Shift key                    */
03813   #define ALT     0x0800          /* Alternate key                */
03814   #define EXTKEY  0x1000          /* extended keycode             */
```

```
03815   #define HASCAPS 0x8000          /* Caps Lock has effect         */
03816
03817   /* Numeric keypad */
03818   #define HOME    (0x01 + EXT)
03819   #define END     (0x02 + EXT)
03820   #define UP      (0x03 + EXT)
03821   #define DOWN    (0x04 + EXT)
03822   #define LEFT    (0x05 + EXT)
03823   #define RIGHT   (0x06 + EXT)
03824   #define PGUP    (0x07 + EXT)
03825   #define PGDN    (0x08 + EXT)
03826   #define MID     (0x09 + EXT)
03827   #define NMIN    (0x0A + EXT)
03828   #define PLUS    (0x0B + EXT)
03829   #define INSRT   (0x0C + EXT)
03830
03831   /* Alt + Numeric keypad */
03832   #define AHOME   (0x01 + ALT)
03833   #define AEND    (0x02 + ALT)
03834   #define AUP     (0x03 + ALT)
03835   #define ADOWN   (0x04 + ALT)
03836   #define ALEFT   (0x05 + ALT)
03837   #define ARIGHT  (0x06 + ALT)
03838   #define APGUP   (0x07 + ALT)
03839   #define APGDN   (0x08 + ALT)
03840   #define AMID    (0x09 + ALT)
03841   #define ANMIN   (0x0A + ALT)
03842   #define APLUS   (0x0B + ALT)
03843   #define AINSRT  (0x0C + ALT)
03844
03845   /* Ctrl + Numeric keypad */
03846   #define CHOME   (0x01 + CTRL)
03847   #define CEND    (0x02 + CTRL)
03848   #define CUP     (0x03 + CTRL)
03849   #define CDOWN   (0x04 + CTRL)
03850   #define CLEFT   (0x05 + CTRL)
03851   #define CRIGHT  (0x06 + CTRL)
03852   #define CPGUP   (0x07 + CTRL)
03853   #define CPGDN   (0x08 + CTRL)
03854   #define CMID    (0x09 + CTRL)
03855   #define CNMIN   (0x0A + CTRL)
03856   #define CPLUS   (0x0B + CTRL)
03857   #define CINSRT  (0x0C + CTRL)
03858
03859   /* Lock keys */
03860   #define CALOCK  (0x0D + EXT)    /* caps lock    */
03861   #define NLOCK   (0x0E + EXT)    /* number lock  */
03862   #define SLOCK   (0x0F + EXT)    /* scroll lock  */
03863
03864   /* Function keys */
03865   #define F1      (0x10 + EXT)
03866   #define F2      (0x11 + EXT)
03867   #define F3      (0x12 + EXT)
03868   #define F4      (0x13 + EXT)
03869   #define F5      (0x14 + EXT)
03870   #define F6      (0x15 + EXT)
03871   #define F7      (0x16 + EXT)
03872   #define F8      (0x17 + EXT)
03873   #define F9      (0x18 + EXT)
03874   #define F10     (0x19 + EXT)
```



```
03875   #define F11     (0x1A + EXT)
03876   #define F12     (0x1B + EXT)
03877
03878   /* Alt+Fn */
03879   #define AF1     (0x10 + ALT)
03880   #define AF2     (0x11 + ALT)
03881   #define AF3     (0x12 + ALT)
03882   #define AF4     (0x13 + ALT)
03883   #define AF5     (0x14 + ALT)
03884   #define AF6     (0x15 + ALT)
03885   #define AF7     (0x16 + ALT)
03886   #define AF8     (0x17 + ALT)
03887   #define AF9     (0x18 + ALT)
03888   #define AF10    (0x19 + ALT)
03889   #define AF11    (0x1A + ALT)
03890   #define AF12    (0x1B + ALT)
03891
03892   /* Ctrl+Fn */
03893   #define CF1     (0x10 + CTRL)
03894   #define CF2     (0x11 + CTRL)
03895   #define CF3     (0x12 + CTRL)
03896   #define CF4     (0x13 + CTRL)
03897   #define CF5     (0x14 + CTRL)
03898   #define CF6     (0x15 + CTRL)
03899   #define CF7     (0x16 + CTRL)
03900   #define CF8     (0x17 + CTRL)
03901   #define CF9     (0x18 + CTRL)
03902   #define CF10    (0x19 + CTRL)
03903   #define CF11    (0x1A + CTRL)
03904   #define CF12    (0x1B + CTRL)
03905
03906   /* Shift+Fn */
03907   #define SF1     (0x10 + SHIFT)
03908   #define SF2     (0x11 + SHIFT)
03909   #define SF3     (0x12 + SHIFT)
03910   #define SF4     (0x13 + SHIFT)
03911   #define SF5     (0x14 + SHIFT)
03912   #define SF6     (0x15 + SHIFT)
03913   #define SF7     (0x16 + SHIFT)
03914   #define SF8     (0x17 + SHIFT)
03915   #define SF9     (0x18 + SHIFT)
03916   #define SF10    (0x19 + SHIFT)
03917   #define SF11    (0x1A + SHIFT)
03918   #define SF12    (0x1B + SHIFT)
03919
03920   /* Alt+Shift+Fn */
03921   #define ASF1    (0x10 + ALT + SHIFT)
03922   #define ASF2    (0x11 + ALT + SHIFT)
03923   #define ASF3    (0x12 + ALT + SHIFT)
03924   #define ASF4    (0x13 + ALT + SHIFT)
03925   #define ASF5    (0x14 + ALT + SHIFT)
03926   #define ASF6    (0x15 + ALT + SHIFT)
03927   #define ASF7    (0x16 + ALT + SHIFT)
03928   #define ASF8    (0x17 + ALT + SHIFT)
03929   #define ASF9    (0x18 + ALT + SHIFT)
03930   #define ASF10   (0x19 + ALT + SHIFT)
03931   #define ASF11   (0x1A + ALT + SHIFT)
03932   #define ASF12   (0x1B + ALT + SHIFT)
03933
03934   #define MAP_COLS        6       /* Number of columns in keymap */
```

```
03935    #define NR_SCAN_CODES   0x80    /* Number of scan codes (rows in keymap) */
03936
03937    typedef unsigned short keymap_t[NR_SCAN_CODES * MAP_COLS];
03938
03939    #define KEY_MAGIC       "KMAZ"  /* Magic number of keymap file */
03940
03941    #endif /* _SYS__KEYMAP_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                include/minix/partition.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

04000    /*      minix/partition.h                               Author: Kees J. Bot
04001     *                                                             7 Dec 1995
04002     * Place of a partition on disk and the disk geometry,
04003     * for use with the DIOCGETP and DIOCSETP ioctl's.
04004     */
04005    #ifndef _MINIX__PARTITION_H
04006    #define _MINIX__PARTITION_H
04007
04008    struct partition {
04009      u32_t base;           /* byte offset to the partition start */
04010      u32_t size;           /* number of bytes in the partition */
04011      unsigned cylinders;   /* disk geometry */
04012      unsigned heads;
04013      unsigned sectors;
04014    };
04015    #endif /* _MINIX__PARTITION_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                include/ibm/partition.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

04100    /* Description of entry in partition table.  */
04101    #ifndef _PARTITION_H
04102    #define _PARTITION_H
04103
04104    struct part_entry {
04105      unsigned char bootind;        /* boot indicator 0/ACTIVE_FLAG  */
04106      unsigned char start_head;     /* head value for first sector   */
04107      unsigned char start_sec;      /* sector value + cyl bits for first sector */
04108      unsigned char start_cyl;      /* track value for first sector  */
04109      unsigned char sysind;         /* system indicator              */
04110      unsigned char last_head;      /* head value for last sector    */
04111      unsigned char last_sec;       /* sector value + cyl bits for last sector */
04112      unsigned char last_cyl;       /* track value for last sector   */
04113      unsigned long lowsec;         /* logical first sector          */
04114      unsigned long size;           /* size of partition in sectors  */
04115    };
04116
04117    #define ACTIVE_FLAG     0x80    /* value for active in bootind field (hd0) */
04118    #define NR_PARTITIONS   4       /* number of entries in partition table */
04119    #define PART_TABLE_OFF  0x1BE   /* offset of partition table in boot sector */
```



```
04120
04121    /* Partition types. */
04122    #define MINIX_PART      0x81    /* Minix partition type */
04123    #define NO_PART         0x00    /* unused entry */
04124    #define OLD_MINIX_PART  0x80    /* created before 1.4b, obsolete */
04125    #define EXT_PART        0x05    /* extended partition */
04126
04127    #endif /* _PARTITION_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/kernel/kernel.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

04200    /* This is the master header for the kernel.  It includes some other files
04201     * and defines the principal constants.
04202     */
04203    #define _POSIX_SOURCE      1    /* tell headers to include POSIX stuff */
04204    #define _MINIX             1    /* tell headers to include MINIX stuff */
04205    #define _SYSTEM            1    /* tell headers that this is the kernel */
04206
04207    /* The following are so basic, all the *.c files get them automatically. */
04208    #include <minix/config.h>       /* MUST be first */
04209    #include <ansi.h>               /* MUST be second */
04210    #include <sys/types.h>
04211    #include <minix/const.h>
04212    #include <minix/type.h>
04213    #include <minix/syslib.h>
04214
04215    #include <string.h>
04216    #include <limits.h>
04217    #include <errno.h>
04218
04219    #include "const.h"
04220    #include "type.h"
04221    #include "proto.h"
04222    #include "glo.h"
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/kernel/const.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

04300    /* General constants used by the kernel. */
04301
04302    #if (CHIP == INTEL)
04303
04304    #define K_STACK_BYTES   1024    /* how many bytes for the kernel stack */
04305
04306    #define INIT_PSW      0x0200    /* initial psw */
04307    #define INIT_TASK_PSW 0x1200    /* initial psw for tasks (with IOPL 1) */
04308    #define TRACEBIT       0x100    /* OR this with psw in proc[] for tracing */
04309    #define SETPSW(rp, new) /* permits only certain bits to be set */ \
04310            ((rp)->p_reg.psw = (rp)->p_reg.psw & ~0xCD5 | (new) & 0xCD5)
04311
04312    /* Initial sp for mm, fs and init.
04313     *      2 bytes for short jump
04314     *      2 bytes unused
```

```
04315    *      3 words for init_org[] used by fs only
04316    *      3 words for real mode debugger trap (actually needs 1 more)
04317    *      3 words for save and restart temporaries
04318    *      3 words for interrupt
04319    * Leave no margin, to flush bugs early.
04320    */
04321   #define INIT_SP (2 + 2 + 3 * 2 + 3 * 2 + 3 * 2 + 3 * 2)
04322
04323   #define HCLICK_SHIFT       4    /* log2 of HCLICK_SIZE */
04324   #define HCLICK_SIZE       16    /* hardware segment conversion magic */
04325   #if CLICK_SIZE >= HCLICK_SIZE
04326   #define click_to_hclick(n) ((n) << (CLICK_SHIFT - HCLICK_SHIFT))
04327   #else
04328   #define click_to_hclick(n) ((n) >> (HCLICK_SHIFT - CLICK_SHIFT))
04329   #endif
04330   #define hclick_to_physb(n) ((phys_bytes) (n) << HCLICK_SHIFT)
04331   #define physb_to_hclick(n) ((n) >> HCLICK_SHIFT)
04332
04333   /* Interrupt vectors defined/reserved by processor. */
04334   #define DIVIDE_VECTOR      0    /* divide error */
04335   #define DEBUG_VECTOR       1    /* single step (trace) */
04336   #define NMI_VECTOR         2    /* non-maskable interrupt */
04337   #define BREAKPOINT_VECTOR  3    /* software breakpoint */
04338   #define OVERFLOW_VECTOR    4    /* from INTO */
04339
04340   /* Fixed system call vector. */
04341   #define SYS_VECTOR        32    /* system calls are made with int SYSVEC */
04342   #define SYS386_VECTOR     33    /* except 386 system calls use this */
04343   #define LEVEL0_VECTOR     34    /* for execution of a function at level 0 */
04344
04345   /* Suitable irq bases for hardware interrupts.  Reprogram the 8259(s) from
04346    * the PC BIOS defaults since the BIOS doesn't respect all the processor's
04347    * reserved vectors (0 to 31).
04348    */
04349   #define BIOS_IRQ0_VEC   0x08    /* base of IRQ0-7 vectors used by BIOS */
04350   #define BIOS_IRQ8_VEC   0x70    /* base of IRQ8-15 vectors used by BIOS */
04351   #define IRQ0_VECTOR     0x28    /* more or less arbitrary, but > SYS_VECTOR */
04352   #define IRQ8_VECTOR     0x30    /* together for simplicity */
04353
04354   /* Hardware interrupt numbers. */
04355   #define NR_IRQ_VECTORS    16
04356   #define CLOCK_IRQ          0
04357   #define KEYBOARD_IRQ       1
04358   #define CASCADE_IRQ        2    /* cascade enable for 2nd AT controller */
04359   #define ETHER_IRQ          3    /* default ethernet interrupt vector */
04360   #define SECONDARY_IRQ      3    /* RS232 interrupt vector for port 2 */
04361   #define RS232_IRQ          4    /* RS232 interrupt vector for port 1 */
04362   #define XT_WINI_IRQ        5    /* xt winchester */
04363   #define FLOPPY_IRQ         6    /* floppy disk */
04364   #define PRINTER_IRQ        7
04365   #define AT_WINI_IRQ       14    /* at winchester */
04366
04367   /* Interrupt number to hardware vector. */
04368   #define BIOS_VECTOR(irq)        \
04369           (((irq) < 8 ? BIOS_IRQ0_VEC : BIOS_IRQ8_VEC) + ((irq) & 0x07))
04370   #define VECTOR(irq)     \
04371           (((irq) < 8 ? IRQ0_VECTOR : IRQ8_VECTOR) + ((irq) & 0x07))
04372
04373   /* BIOS hard disk parameter vectors. */
04374   #define WINI_0_PARM_VEC 0x41
```



```
04375   #define WINI_1_PARM_VEC 0x46
04376
04377   /* 8259A interrupt controller ports. */
04378   #define INT_CTL         0x20    /* I/O port for interrupt controller */
04379   #define INT_CTLMASK     0x21    /* setting bits in this port disables ints */
04380   #define INT2_CTL        0xA0    /* I/O port for second interrupt controller */
04381   #define INT2_CTLMASK    0xA1    /* setting bits in this port disables ints */
04382
04383   /* Magic numbers for interrupt controller. */
04384   #define ENABLE          0x20    /* code used to re-enable after an interrupt */
04385
04386   /* Sizes of memory tables. */
04387   #define NR_MEMS            3    /* number of chunks of memory */
04388
04389   /* Miscellaneous ports. */
04390   #define PCR             0x65    /* Planar Control Register */
04391   #define PORT_B          0x61    /* I/O port for 8255 port B (kbd, beeper...) */
04392   #define TIMER0          0x40    /* I/O port for timer channel 0 */
04393   #define TIMER2          0x42    /* I/O port for timer channel 2 */
04394   #define TIMER_MODE      0x43    /* I/O port for timer mode control */
04395
04396   #endif /* (CHIP == INTEL) */
04397
04398   #if (CHIP == M68000)
04399
04400   #define K_STACK_BYTES   1024    /* how many bytes for the kernel stack */
04401
04402   /* Sizes of memory tables. */
04403   #define NR_MEMS            2    /* number of chunks of memory */
04404
04405   /* p_reg contains: d0-d7, a0-a6,   in that order. */
04406   #define NR_REGS           15    /* number of general regs in each proc slot */
04407
04408   #define TRACEBIT      0x8000    /* or this with psw in proc[] for tracing */
04409   #define SETPSW(rp, new)         /* permits only certain bits to be set */ \
04410           ((rp)->p_reg.psw = (rp)->p_reg.psw & ~0xFF | (new) & 0xFF)
04411
04412   #define MEM_BYTES  0xffffffff   /* memory size for /dev/mem */
04413
04414   #ifdef __ACK__
04415   #define FSTRUCOPY
04416   #endif
04417
04418   #endif /* (CHIP == M68000) */
04419
04420   /* The following items pertain to the scheduling queues. */
04421   #define TASK_Q             0    /* ready tasks are scheduled via queue 0 */
04422   #define SERVER_Q           1    /* ready servers are scheduled via queue 1 */
04423   #define USER_Q             2    /* ready users are scheduled via queue 2 */
04424
04425   #if (MACHINE == ATARI)
04426   #define SHADOW_Q           3    /* runnable, but shadowed processes */
04427   #define NQ                 4    /* # of scheduling queues */
04428   #else
04429   #define NQ                 3    /* # of scheduling queues */
04430   #endif
04431
04432   /* Env_parse() return values. */
04433   #define EP_UNSET        0       /* variable not set */
04434   #define EP_OFF          1       /* var = off */
```

```
04435   #define EP_ON           2       /* var = on (or field left blank) */
04436   #define EP_SET          3       /* var = 1:2:3 (nonblank field) */
04437
04438   /* To translate an address in kernel space to a physical address.  This is
04439    * the same as umap(proc_ptr, D, vir, sizeof(*vir)), but a lot less costly.
04440    */
04441   #define vir2phys(vir)   (data_base + (vir_bytes) (vir))
04442
04443   #define printf        printk    /* the kernel really uses printk, not printf */


++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                              src/kernel/type.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

04500   #ifndef TYPE_H
04501   #define TYPE_H
04502
04503   typedef _PROTOTYPE( void task_t, (void) );
04504   typedef _PROTOTYPE( int (*rdwt_t), (message *m_ptr) );
04505   typedef _PROTOTYPE( void (*watchdog_t), (void) );
04506
04507   struct tasktab {
04508     task_t *initial_pc;
04509     int stksize;
04510     char name[8];
04511   };
04512
04513   struct memory {
04514     phys_clicks base;
04515     phys_clicks size;
04516   };
04517
04518   /* Administration for clock polling. */
04519   struct milli_state {
04520     unsigned long accum_count;    /* accumulated clock ticks */
04521     unsigned prev_count;          /* previous clock value */
04522   };
04523
04524   #if (CHIP == INTEL)
04525   typedef unsigned port_t;
04526   typedef unsigned segm_t;
04527   typedef unsigned reg_t;         /* machine register */
04528
04529   /* The stack frame layout is determined by the software, but for efficiency
04530    * it is laid out so the assembly code to use it is as simple as possible.
04531    * 80286 protected mode and all real modes use the same frame, built with
04532    * 16-bit registers.  Real mode lacks an automatic stack switch, so little
04533    * is lost by using the 286 frame for it.  The 386 frame differs only in
04534    * having 32-bit registers and more segment registers.  The same names are
04535    * used for the larger registers to avoid differences in the code.
04536    */
04537   struct stackframe_s {           /* proc_ptr points here */
04538   #if _WORD_SIZE == 4
04539     u16_t gs;                     /* last item pushed by save */
04540     u16_t fs;                     /*  ^ */
04541   #endif
04542     u16_t es;                     /*  | */
04543     u16_t ds;                     /*  | */
04544     reg_t di;                     /* di through cx are not accessed in C */
```



```
04545     reg_t si;                   /* order is to match pusha/popa */
04546     reg_t fp;                   /* bp */
04547     reg_t st;                   /* hole for another copy of sp */
04548     reg_t bx;                   /*  | */
04549     reg_t dx;                   /*  | */
04550     reg_t cx;                   /*  | */
04551     reg_t retreg;               /* ax and above are all pushed by save */
04552     reg_t retadr;               /* return address for assembly code save() */
04553     reg_t pc;                   /*  ^  last item pushed by interrupt */
04554     reg_t cs;                   /*  | */
04555     reg_t psw;                  /*  | */
04556     reg_t sp;                   /*  | */
04557     reg_t ss;                   /* these are pushed by CPU during interrupt */
04558   };
04559
04560   struct segdesc_s {            /* segment descriptor for protected mode */
04561     u16_t limit_low;
04562     u16_t base_low;
04563     u8_t base_middle;
04564     u8_t access;                /* |P|DL|1|X|E|R|A| */
04565   #if _WORD_SIZE == 4
04566     u8_t granularity;           /* |G|X|0|A|LIMT| */
04567     u8_t base_high;
04568   #else
04569     u16_t reserved;
04570   #endif
04571   };
04572
04573   typedef _PROTOTYPE( int (*irq_handler_t), (int irq) );
04574
04575   #endif /* (CHIP == INTEL) */
04576
04577   #if (CHIP == M68000)
04578   typedef _PROTOTYPE( void (*dmaint_t), (void) );
04579
04580   typedef u32_t reg_t;          /* machine register */
04581
04582   /* The name and fields of this struct were chosen for PC compatibility. */
04583   struct stackframe_s {
04584     reg_t retreg;               /* d0 */
04585     reg_t d1;
04586     reg_t d2;
04587     reg_t d3;
04588     reg_t d4;
04589     reg_t d5;
04590     reg_t d6;
04591     reg_t d7;
04592     reg_t a0;
04593     reg_t a1;
04594     reg_t a2;
04595     reg_t a3;
04596     reg_t a4;
04597     reg_t a5;
04598     reg_t fp;                   /* also known as a6 */
04599     reg_t sp;                   /* also known as a7 */
04600     reg_t pc;
04601     u16_t psw;
04602     u16_t dummy;                /* make size multiple of reg_t for system.c */
04603   };
04604
```

```
04605    struct fsave {
04606      struct cpu_state {
04607            u16_t i_format;
04608            u32_t i_addr;
04609            u16_t i_state[4];
04610      } cpu_state;
04611      struct state_frame {
04612            u8_t frame_type;
04613            u8_t frame_size;
04614            u16_t reserved;
04615            u8_t frame[212];
04616      } state_frame;
04617      struct fpp_model {
04618            u32_t fpcr;
04619            u32_t fpsr;
04620            u32_t fpiar;
04621            struct fpN {
04622                    u32_t high;
04623                    u32_t low;
04624                    u32_t mid;
04625            } fpN[8];
04626      } fpp_model;
04627    };
04628    #endif /* (CHIP == M68000) */
04629
04630    #endif /* TYPE_H */
```

++

src/kernel/proto.h

++

```
04700    /* Function prototypes. */
04701
04702    #ifndef PROTO_H
04703    #define PROTO_H
04704
04705    /* Struct declarations. */
04706    struct proc;
04707    struct tty;
04708
04709    /* at_wini.c, wini.c */
04710    _PROTOTYPE( void winchester_task, (void)                                );
04711    _PROTOTYPE( void at_winchester_task, (void)                             );
04712
04713    /* clock.c */
04714    _PROTOTYPE( void clock_task, (void)                                     );
04715    _PROTOTYPE( void clock_stop, (void)                                     );
04716    _PROTOTYPE( clock_t get_uptime, (void)                                  );
04717    _PROTOTYPE( void syn_alrm_task, (void)                                  );
04718
04719    /* dmp.c */
04720    _PROTOTYPE( void map_dmp, (void)                                        );
04721    _PROTOTYPE( void p_dmp, (void)                                          );
04722    _PROTOTYPE( void reg_dmp, (struct proc *rp)                             );
04723
04724    /* dp8390.c */
```



```
04725    _PROTOTYPE( void dp8390_task, (void)                                    );
04726    _PROTOTYPE( void dp_dump, (void)                                        );
04727    _PROTOTYPE( void dp8390_stop, (void)                                    );
04728
04729    /* floppy.c, stfloppy.c */
04730    _PROTOTYPE( void floppy_task, (void)                                    );
04731    _PROTOTYPE( void floppy_stop, (void)                                    );
04732
04733    /* main.c, stmain.c */
04734    _PROTOTYPE( void main, (void)                                           );
04735    _PROTOTYPE( void panic, (const char *s, int n)                          );
04736
04737    /* memory.c */
04738    _PROTOTYPE( void mem_task, (void)                                       );
04739
04740    /* misc.c */
04741    _PROTOTYPE( int env_parse, (char *env, char *fmt, int field,
04742                            long *param, long min, long max)                );
04743
04744    /* printer.c, stprint.c */
04745    _PROTOTYPE( void printer_task, (void)                                   );
04746
04747    /* proc.c */
04748    _PROTOTYPE( void interrupt, (int task)                                  );
04749    _PROTOTYPE( int lock_mini_send, (struct proc *caller_ptr,
04750                    int dest, message *m_ptr)                               );
04751    _PROTOTYPE( void lock_pick_proc, (void)                                 );
04752    _PROTOTYPE( void lock_ready, (struct proc *rp)                          );
04753    _PROTOTYPE( void lock_sched, (void)                                     );
04754    _PROTOTYPE( void lock_unready, (struct proc *rp)                        );
04755    _PROTOTYPE( int sys_call, (int function, int src_dest, message *m_ptr)  );
04756    _PROTOTYPE( void unhold, (void)                                         );
04757
04758    /* rs232.c */
04759    _PROTOTYPE( void rs_init, (struct tty *tp)                              );
04760
04761    /* system.c */
04762    _PROTOTYPE( void cause_sig, (int proc_nr, int sig_nr)                   );
04763    _PROTOTYPE( void inform, (void)                                         );
04764    _PROTOTYPE( phys_bytes numap, (int proc_nr, vir_bytes vir_addr,
04765                    vir_bytes bytes)                                        );
04766    _PROTOTYPE( void sys_task, (void)                                       );
04767    _PROTOTYPE( phys_bytes umap, (struct proc *rp, int seg, vir_bytes vir_addr,
04768                    vir_bytes bytes)                                        );
04769
04770    /* tty.c */
04771    _PROTOTYPE( void handle_events, (struct tty *tp)                        );
04772    _PROTOTYPE( void sigchar, (struct tty *tp, int sig)                     );
04773    _PROTOTYPE( void tty_task, (void)                                       );
04774    _PROTOTYPE( int in_process, (struct tty *tp, char *buf, int count)      );
04775    _PROTOTYPE( void out_process, (struct tty *tp, char *bstart, char *bpos,
04776                                char *bend, int *icount, int *ocount)       );
04777    _PROTOTYPE( void tty_wakeup, (clock_t now)                              );
04778    _PROTOTYPE( void tty_reply, (int code, int replyee, int proc_nr,
04779                                                    int status)             );
04780    _PROTOTYPE( void tty_devnop, (struct tty *tp)                           );
04781
04782    /* library */
04783    _PROTOTYPE( void *memcpy, (void *_s1, const void *_s2, size_t _n)       );
04784
```

```
04785   #if (CHIP == INTEL)
04786
04787   /* clock.c */
04788   _PROTOTYPE( void milli_start, (struct milli_state *msp)                );
04789   _PROTOTYPE( unsigned milli_elapsed, (struct milli_state *msp)          );
04790   _PROTOTYPE( void milli_delay, (unsigned millisec)                      );
04791
04792   /* console.c */
04793   _PROTOTYPE( void cons_stop, (void)                                     );
04794   _PROTOTYPE( void putk, (int c)                                         );
04795   _PROTOTYPE( void scr_init, (struct tty *tp)                            );
04796   _PROTOTYPE( void toggle_scroll, (void)                                 );
04797   _PROTOTYPE( int con_loadfont, (phys_bytes user_phys)                   );
04798   _PROTOTYPE( void select_console, (int cons_line)                       );
04799
04800   /* cstart.c */
04801   _PROTOTYPE( void cstart, (U16_t cs, U16_t ds, U16_t mcs, U16_t mds,
04802                                 U16_t parmoff, U16_t parmsize)           );
04803   _PROTOTYPE( char *k_getenv, (char *name)                               );
04804
04805   /* exception.c */
04806   _PROTOTYPE( void exception, (unsigned vec_nr)                          );
04807
04808   /* i8259.c */
04809   _PROTOTYPE( irq_handler_t get_irq_handler, (int irq)                   );
04810   _PROTOTYPE( void put_irq_handler, (int irq, irq_handler_t handler)     );
04811   _PROTOTYPE( void intr_init, (int mine)                                 );
04812
04813   /* keyboard.c */
04814   _PROTOTYPE( void kb_init, (struct tty *tp)                             );
04815   _PROTOTYPE( int kbd_loadmap, (phys_bytes user_phys)                    );
04816   _PROTOTYPE( void wreboot, (int how)                                    );
04817
04818   /* klib*.s */
04819   _PROTOTYPE( void bios13, (void)                                        );
04820   _PROTOTYPE( phys_bytes check_mem, (phys_bytes base, phys_bytes size)   );
04821   _PROTOTYPE( void cp_mess, (int src,phys_clicks src_clicks,vir_bytes src_offset,
04822                 phys_clicks dst_clicks, vir_bytes dst_offset)            );
04823   _PROTOTYPE( int in_byte, (port_t port)                                 );
04824   _PROTOTYPE( int in_word, (port_t port)                                 );
04825   _PROTOTYPE( void lock, (void)                                          );
04826   _PROTOTYPE( void unlock, (void)                                        );
04827   _PROTOTYPE( void enable_irq, (unsigned irq)                            );
04828   _PROTOTYPE( int disable_irq, (unsigned irq)                            );
04829   _PROTOTYPE( u16_t mem_rdw, (segm_t segm, vir_bytes offset)             );
04830   _PROTOTYPE( void out_byte, (port_t port, int value)                    );
04831   _PROTOTYPE( void out_word, (port_t port, int value)                    );
04832   _PROTOTYPE( void phys_copy, (phys_bytes source, phys_bytes dest,
04833                 phys_bytes count)                                        );
04834   _PROTOTYPE( void port_read, (unsigned port, phys_bytes destination,
04835                 unsigned bytcount)                                       );
04836   _PROTOTYPE( void port_read_byte, (unsigned port, phys_bytes destination,
04837                 unsigned bytcount)                                       );
04838   _PROTOTYPE( void port_write, (unsigned port, phys_bytes source,
04839                 unsigned bytcount)                                       );
04840   _PROTOTYPE( void port_write_byte, (unsigned port, phys_bytes source,
04841                 unsigned bytcount)                                       );
04842   _PROTOTYPE( void reset, (void)                                         );
04843   _PROTOTYPE( void vid_vid_copy, (unsigned src, unsigned dst, unsigned count));
04844   _PROTOTYPE( void mem_vid_copy, (u16_t *src, unsigned dst, unsigned count));
```



```
04845   _PROTOTYPE( void level0, (void (*func)(void))                          );
04846   _PROTOTYPE( void monitor, (void)                                       );
04847
04848   /* misc.c */
04849   _PROTOTYPE( void mem_init, (void)                                      );
04850
04851   /* mpx*.s */
04852   _PROTOTYPE( void idle_task, (void)                                     );
04853   _PROTOTYPE( void restart, (void)                                       );
04854
04855   /* The following are never called from C (pure asm procs). */
04856
04857   /* Exception handlers (real or protected mode), in numerical order. */
04858   void _PROTOTYPE( int00, (void) ), _PROTOTYPE( divide_error, (void) );
04859   void _PROTOTYPE( int01, (void) ), _PROTOTYPE( single_step_exception, (void) );
04860   void _PROTOTYPE( int02, (void) ), _PROTOTYPE( nmi, (void) );
04861   void _PROTOTYPE( int03, (void) ), _PROTOTYPE( breakpoint_exception, (void) );
04862   void _PROTOTYPE( int04, (void) ), _PROTOTYPE( overflow, (void) );
04863   void _PROTOTYPE( int05, (void) ), _PROTOTYPE( bounds_check, (void) );
04864   void _PROTOTYPE( int06, (void) ), _PROTOTYPE( inval_opcode, (void) );
04865   void _PROTOTYPE( int07, (void) ), _PROTOTYPE( copr_not_available, (void) );
04866   void                              _PROTOTYPE( double_fault, (void) );
04867   void                              _PROTOTYPE( copr_seg_overrun, (void) );
04868   void                              _PROTOTYPE( inval_tss, (void) );
04869   void                              _PROTOTYPE( segment_not_present, (void) );
04870   void                              _PROTOTYPE( stack_exception, (void) );
04871   void                              _PROTOTYPE( general_protection, (void) );
04872   void                              _PROTOTYPE( page_fault, (void) );
04873   void                              _PROTOTYPE( copr_error, (void) );
04874
04875   /* Hardware interrupt handlers. */
04876   _PROTOTYPE( void hwint00, (void) );
04877   _PROTOTYPE( void hwint01, (void) );
04878   _PROTOTYPE( void hwint02, (void) );
04879   _PROTOTYPE( void hwint03, (void) );
04880   _PROTOTYPE( void hwint04, (void) );
04881   _PROTOTYPE( void hwint05, (void) );
04882   _PROTOTYPE( void hwint06, (void) );
04883   _PROTOTYPE( void hwint07, (void) );
04884   _PROTOTYPE( void hwint08, (void) );
04885   _PROTOTYPE( void hwint09, (void) );
04886   _PROTOTYPE( void hwint10, (void) );
04887   _PROTOTYPE( void hwint11, (void) );
04888   _PROTOTYPE( void hwint12, (void) );
04889   _PROTOTYPE( void hwint13, (void) );
04890   _PROTOTYPE( void hwint14, (void) );
04891   _PROTOTYPE( void hwint15, (void) );
04892
04893   /* Software interrupt handlers, in numerical order. */
04894   _PROTOTYPE( void trp, (void) );
04895   _PROTOTYPE( void s_call, (void) ), _PROTOTYPE( p_s_call, (void) );
04896   _PROTOTYPE( void level0_call, (void) );
04897
04898   /* printer.c */
04899   _PROTOTYPE( void pr_restart, (void)                                    );
04900
04901   /* protect.c */
04902   _PROTOTYPE( void prot_init, (void)                                     );
04903   _PROTOTYPE( void init_codeseg, (struct segdesc_s *segdp, phys_bytes base,
04904                 phys_bytes size, int privilege)                          );
```

```
04905   _PROTOTYPE( void init_dataseg, (struct segdesc_s *segdp, phys_bytes base,
04906                 phys_bytes size, int privilege)                       );
04907   _PROTOTYPE( phys_bytes seg2phys, (U16_t seg)                        );
04908   _PROTOTYPE( void enable_iop, (struct proc *pp)                      );
04909
04910   /* pty.c */
04911   _PROTOTYPE( void do_pty, (struct tty *tp, message *m_ptr)           );
04912   _PROTOTYPE( void pty_init, (struct tty *tp)                         );
04913
04914   /* system.c */
04915   _PROTOTYPE( void alloc_segments, (struct proc *rp)                  );
04916
04917   #endif /* (CHIP == INTEL) */
04918
04919   #endif /* PROTO_H */
```

```
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                              src/kernel/glo.h
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

05000   /* Global variables used in the kernel. */
05001
05002   /* EXTERN is defined as extern except in table.c. */
05003   #ifdef _TABLE
05004   #undef EXTERN
05005   #define EXTERN
05006   #endif
05007
05008   /* Kernel memory. */
05009   EXTERN phys_bytes code_base;    /* base of kernel code */
05010   EXTERN phys_bytes data_base;    /* base of kernel data */
05011
05012   /* Low level interrupt communications. */
05013   EXTERN struct proc *held_head;  /* head of queue of held-up interrupts */
05014   EXTERN struct proc *held_tail;  /* tail of queue of held-up interrupts */
05015   EXTERN unsigned char k_reenter; /* kernel reentry count (entry count less 1)*/
05016
05017   /* Process table.  Here to stop too many things having to include proc.h. */
05018   EXTERN struct proc *proc_ptr;   /* pointer to currently running process */
05019
05020   /* Signals. */
05021   EXTERN int sig_procs;           /* number of procs with p_pending != 0 */
05022
05023   /* Memory sizes. */
05024   EXTERN struct memory mem[NR_MEMS];      /* base and size of chunks of memory */
05025   EXTERN phys_clicks tot_mem_size;        /* total system memory size */
05026
05027   /* Miscellaneous. */
05028   extern u16_t sizes[];           /* table filled in by boot monitor */
05029   extern struct tasktab tasktab[];/* initialized in table.c, so extern here */
05030   extern char *t_stack[];         /* initialized in table.c, so extern here */
05031   EXTERN unsigned lost_ticks;     /* clock ticks counted outside the clock task */
05032   EXTERN clock_t tty_timeout;     /* time to wake up the TTY task */
05033   EXTERN int current;             /* currently visible console */
05034
```



```
05035   #if (CHIP == INTEL)
05036
05037   /* Machine type. */
05038   EXTERN int pc_at;               /* PC-AT compatible hardware interface */
05039   EXTERN int ps_mca;              /* PS/2 with Micro Channel */
05040   EXTERN unsigned int processor;  /* 86, 186, 286, 386, ... */
05041   #if _WORD_SIZE == 2
05042   EXTERN int protected_mode;      /* nonzero if running in Intel protected mode*/
05043   #else
05044   #define protected_mode  1       /* 386 mode implies protected mode */
05045   #endif
05046
05047   /* Video card types. */
05048   EXTERN int ega;                 /* nonzero if console is EGA or VGA */
05049   EXTERN int vga;                 /* nonzero if console is VGA */
05050
05051   /* Memory sizes. */
05052   EXTERN unsigned ext_memsize;    /* initialized by assembler startup code */
05053   EXTERN unsigned low_memsize;
05054
05055   /* Miscellaneous. */
05056   EXTERN irq_handler_t irq_table[NR_IRQ_VECTORS];
05057   EXTERN int irq_use;             /* bit map of all in-use irq's */
05058   EXTERN reg_t mon_ss, mon_sp;    /* monitor stack */
05059   EXTERN int mon_return;          /* true if return to the monitor possible */
05060   EXTERN phys_bytes reboot_code;  /* program for the boot monitor */
05061
05062   /* Variables that are initialized elsewhere are just extern here. */
05063   extern struct segdesc_s gdt[];  /* global descriptor table for protected mode*/
05064
05065   EXTERN _PROTOTYPE( void (*level0_func), (void) );
05066   #endif /* (CHIP == INTEL) */
05067
05068   #if (CHIP == M68000)
05069   /* Variables that are initialized elsewhere are just extern here. */
05070   extern int keypad;              /* Flag for keypad mode */
05071   extern int app_mode;            /* Flag for arrow key application mode */
05072   extern int STdebKey;            /* nonzero if ctl-alt-Fx detected */
05073   extern struct tty *cur_cons;    /* virtual cons currently displayed */
05074   extern unsigned char font8[];   /* 8 pixel wide font table (initialized) */
05075   extern unsigned char font12[];  /* 12 pixel wide font table (initialized) */
05076   extern unsigned char font16[];  /* 16 pixel wide font table (initialized) */
05077   extern unsigned short resolution; /* screen res; ST_RES_LOW..TT_RES_HIGH */
05078   #endif
```

```
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                              src/kernel/proc.h
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

05100   #ifndef PROC_H
05101   #define PROC_H
05102
05103   /* Here is the declaration of the process table.  It contains the process'
05104    * registers, memory map, accounting, and message send/receive information.
05105    * Many assembly code routines reference fields in it.  The offsets to these
05106    * fields are defined in the assembler include file sconst.h.  When changing
05107    * 'proc', be sure to change sconst.h to match.
05108    */
05109
```

```
05110   struct proc {
05111     struct stackframe_s p_reg;    /* process' registers saved in stack frame */
05112
05113   #if (CHIP == INTEL)
05114     reg_t p_ldt_sel;              /* selector in gdt giving ldt base and limit*/
05115     struct segdesc_s p_ldt[2];    /* local descriptors for code and data */
05116                                   /* 2 is LDT_SIZE - avoid include protect.h */
05117   #endif /* (CHIP == INTEL) */
05118
05119     reg_t *p_stguard;             /* stack guard word */
05120
05121     int p_nr;                     /* number of this process (for fast access) */
05122
05123     int p_int_blocked;            /* nonzero if int msg blocked by busy task */
05124     int p_int_held;               /* nonzero if int msg held by busy syscall */
05125     struct proc *p_nextheld;      /* next in chain of held-up int processes */
05126
05127     int p_flags;                  /* P_SLOT_FREE, SENDING, RECEIVING, etc. */
05128     struct mem_map p_map[NR_SEGS];/* memory map */
05129     pid_t p_pid;                  /* process id passed in from MM */
05130
05131     clock_t user_time;            /* user time in ticks */
05132     clock_t sys_time;             /* sys time in ticks */
05133     clock_t child_utime;          /* cumulative user time of children */
05134     clock_t child_stime;          /* cumulative sys time of children */
05135     clock_t p_alarm;              /* time of next alarm in ticks, or 0 */
05136
05137     struct proc *p_callerq;       /* head of list of procs wishing to send */
05138     struct proc *p_sendlink;      /* link to next proc wishing to send */
05139     message *p_messbuf;           /* pointer to message buffer */
05140     int p_getfrom;                /* from whom does process want to receive? */
05141     int p_sendto;
05142
05143     struct proc *p_nextready;     /* pointer to next ready process */
05144     sigset_t p_pending;           /* bit map for pending signals */
05145     unsigned p_pendcount;         /* count of pending and unfinished signals */
05146
05147     char p_name[16];              /* name of the process */
05148   };
05149
05150   /* Guard word for task stacks. */
05151   #define STACK_GUARD     ((reg_t) (sizeof(reg_t) == 2 ? 0xBEEF : 0xDEADBEEF))
05152
05153   /* Bits for p_flags in proc[].  A process is runnable iff p_flags == 0. */
05154   #define P_SLOT_FREE      001    /* set when slot is not in use */
05155   #define NO_MAP           002    /* keeps unmapped forked child from running */
05156   #define SENDING          004    /* set when process blocked trying to send */
05157   #define RECEIVING        010    /* set when process blocked trying to recv */
05158   #define PENDING          020    /* set when inform() of signal pending */
05159   #define SIG_PENDING      040    /* keeps to-be-signalled proc from running */
05160   #define P_STOP          0100    /* set when process is being traced */
05161
05162   /* Magic process table addresses. */
05163   #define BEG_PROC_ADDR (&proc[0])
05164   #define END_PROC_ADDR (&proc[NR_TASKS + NR_PROCS])
05165   #define END_TASK_ADDR (&proc[NR_TASKS])
05166   #define BEG_SERV_ADDR (&proc[NR_TASKS])
05167   #define BEG_USER_ADDR (&proc[NR_TASKS + LOW_USER])
05168
05169   #define NIL_PROC          ((struct proc *) 0)
```



```
05170   #define isidlehardware(n) ((n) == IDLE || (n) == HARDWARE)
05171   #define isokprocn(n)      ((unsigned) ((n) + NR_TASKS) < NR_PROCS + NR_TASKS)
05172   #define isoksrc_dest(n)   (isokprocn(n) || (n) == ANY)
05173   #define isoksusern(n)     ((unsigned) (n) < NR_PROCS)
05174   #define isokusern(n)      ((unsigned) ((n) - LOW_USER) < NR_PROCS - LOW_USER)
05175   #define isrxhardware(n)   ((n) == ANY || (n) == HARDWARE)
05176   #define issysentn(n)      ((n) == FS_PROC_NR || (n) == MM_PROC_NR)
05177   #define istaskp(p)        ((p) < END_TASK_ADDR && (p) != proc_addr(IDLE))
05178   #define isuserp(p)        ((p) >= BEG_USER_ADDR)
05179   #define proc_addr(n)      (pproc_addr + NR_TASKS)[(n)]
05180   #define cproc_addr(n)     (&(proc + NR_TASKS)[(n)])
05181   #define proc_number(p)    ((p)->p_nr)
05182   #define proc_vir2phys(p, vir) \
05183                             (((phys_bytes)(p)->p_map[D].mem_phys << CLICK_SHIFT) \
05184                                                     + (vir_bytes) (vir))
05185
05186   EXTERN struct proc proc[NR_TASKS + NR_PROCS];   /* process table */
05187   EXTERN struct proc *pproc_addr[NR_TASKS + NR_PROCS];
05188   /* ptrs to process table slots; fast because now a process entry can be found
05189      by indexing the pproc_addr array, while accessing an element i requires
05190      a multiplication with sizeof(struct proc) to determine the address */
05191   EXTERN struct proc *bill_ptr;   /* ptr to process to bill for clock ticks */
05192   EXTERN struct proc *rdy_head[NQ];       /* pointers to ready list headers */
05193   EXTERN struct proc *rdy_tail[NQ];       /* pointers to ready list tails */
05194
05195   #endif /* PROC_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/kernel/protect.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

05200   /* Constants for protected mode. */
05201
05202   /* Table sizes. */
05203   #define GDT_SIZE (FIRST_LDT_INDEX + NR_TASKS + NR_PROCS) /* spec. and LDT's */
05204   #define IDT_SIZE (IRQ8_VECTOR + 8)      /* only up to the highest vector */
05205   #define LDT_SIZE         2      /* contains CS and DS only */
05206
05207   /* Fixed global descriptors.  1 to 7 are prescribed by the BIOS. */
05208   #define GDT_INDEX        1      /* GDT descriptor */
05209   #define IDT_INDEX        2      /* IDT descriptor */
05210   #define DS_INDEX         3      /* kernel DS */
05211   #define ES_INDEX         4      /* kernel ES (386: flag 4 Gb at startup) */
05212   #define SS_INDEX         5      /* kernel SS (386: monitor SS at startup) */
05213   #define CS_INDEX         6      /* kernel CS */
05214   #define MON_CS_INDEX     7      /* temp for BIOS (386: monitor CS at startup) */
05215   #define TSS_INDEX        8      /* kernel TSS */
05216   #define DS_286_INDEX     9      /* scratch 16-bit source segment */
05217   #define ES_286_INDEX    10      /* scratch 16-bit destination segment */
05218   #define VIDEO_INDEX     11      /* video memory segment */
05219   #define DP_ETH0_INDEX   12      /* Western Digital Etherplus buffer */
05220   #define DP_ETH1_INDEX   13      /* Western Digital Etherplus buffer */
05221   #define FIRST_LDT_INDEX 14      /* rest of descriptors are LDT's */
05222
05223   #define GDT_SELECTOR      0x08  /* (GDT_INDEX * DESC_SIZE) bad for asld */
05224   #define IDT_SELECTOR      0x10  /* (IDT_INDEX * DESC_SIZE) */
```

```
05225   #define DS_SELECTOR       0x18  /* (DS_INDEX * DESC_SIZE) */
05226   #define ES_SELECTOR       0x20  /* (ES_INDEX * DESC_SIZE) */
05227   #define FLAT_DS_SELECTOR  0x21  /* less privileged ES */
05228   #define SS_SELECTOR       0x28  /* (SS_INDEX * DESC_SIZE) */
05229   #define CS_SELECTOR       0x30  /* (CS_INDEX * DESC_SIZE) */
05230   #define MON_CS_SELECTOR   0x38  /* (MON_CS_INDEX * DESC_SIZE) */
05231   #define TSS_SELECTOR      0x40  /* (TSS_INDEX * DESC_SIZE) */
05232   #define DS_286_SELECTOR   0x49  /* (DS_286_INDEX * DESC_SIZE + 1) */
05233   #define ES_286_SELECTOR   0x51  /* (ES_286_INDEX * DESC_SIZE + 1) */
05234   #define VIDEO_SELECTOR    0x59  /* (VIDEO_INDEX * DESC_SIZE + 1) */
05235   #define DP_ETH0_SELECTOR  0x61  /* (DP_ETH0_INDEX * DESC_SIZE) */
05236   #define DP_ETH1_SELECTOR  0x69  /* (DP_ETH1_INDEX * DESC_SIZE) */
05237
05238   /* Fixed local descriptors. */
05239   #define CS_LDT_INDEX     0      /* process CS */
05240   #define DS_LDT_INDEX     1      /* process DS=ES=FS=GS=SS */
05241
05242   /* Privileges. */
05243   #define INTR_PRIVILEGE   0      /* kernel and interrupt handlers */
05244   #define TASK_PRIVILEGE   1
05245   #define USER_PRIVILEGE   3
05246
05247   /* 286 hardware constants. */
05248
05249   /* Exception vector numbers. */
05250   #define BOUNDS_VECTOR       5   /* bounds check failed */
05251   #define INVAL_OP_VECTOR     6   /* invalid opcode */
05252   #define COPROC_NOT_VECTOR   7   /* coprocessor not available */
05253   #define DOUBLE_FAULT_VECTOR 8
05254   #define COPROC_SEG_VECTOR   9   /* coprocessor segment overrun */
05255   #define INVAL_TSS_VECTOR   10   /* invalid TSS */
05256   #define SEG_NOT_VECTOR     11   /* segment not present */
05257   #define STACK_FAULT_VECTOR 12   /* stack exception */
05258   #define PROTECTION_VECTOR  13   /* general protection */
05259
05260   /* Selector bits. */
05261   #define TI            0x04      /* table indicator */
05262   #define RPL           0x03      /* requester privilege level */
05263
05264   /* Descriptor structure offsets. */
05265   #define DESC_BASE        2      /* to base_low */
05266   #define DESC_BASE_MIDDLE 4      /* to base_middle */
05267   #define DESC_ACCESS      5      /* to access byte */
05268   #define DESC_SIZE        8      /* sizeof (struct segdesc_s) */
05269
05270   /* Segment sizes. */
05271   #define MAX_286_SEG_SIZE 0x10000L
05272
05273   /* Base and limit sizes and shifts. */
05274   #define BASE_MIDDLE_SHIFT   16  /* shift for base --> base_middle */
05275
05276   /* Access-byte and type-byte bits. */
05277   #define PRESENT       0x80      /* set for descriptor present */
05278   #define DPL           0x60      /* descriptor privilege level mask */
05279   #define DPL_SHIFT        5
05280   #define SEGMENT       0x10      /* set for segment-type descriptors */
05281
05282   /* Access-byte bits. */
05283   #define EXECUTABLE    0x08      /* set for executable segment */
05284   #define CONFORMING    0x04      /* set for conforming segment if executable */
```



```
05285   #define EXPAND_DOWN   0x04      /* set for expand-down segment if !executable*/
05286   #define READABLE      0x02      /* set for readable segment if executable */
05287   #define WRITEABLE     0x02      /* set for writeable segment if !executable */
05288   #define TSS_BUSY      0x02      /* set if TSS descriptor is busy */
05289   #define ACCESSED      0x01      /* set if segment accessed */
05290
05291   /* Special descriptor types. */
05292   #define AVL_286_TSS      1      /* available 286 TSS */
05293   #define LDT              2      /* local descriptor table */
05294   #define BUSY_286_TSS     3      /* set transparently to the software */
05295   #define CALL_286_GATE    4      /* not used */
05296   #define TASK_GATE        5      /* only used by debugger */
05297   #define INT_286_GATE     6      /* interrupt gate, used for all vectors */
05298   #define TRAP_286_GATE    7      /* not used */
05299
05300   /* Extra 386 hardware constants. */
05301
05302   /* Exception vector numbers. */
05303   #define PAGE_FAULT_VECTOR   14
05304   #define COPROC_ERR_VECTOR   16  /* coprocessor error */
05305
05306   /* Descriptor structure offsets. */
05307   #define DESC_GRANULARITY     6  /* to granularity byte */
05308   #define DESC_BASE_HIGH       7  /* to base_high */
05309
05310   /* Base and limit sizes and shifts. */
05311   #define BASE_HIGH_SHIFT     24  /* shift for base --> base_high */
05312   #define BYTE_GRAN_MAX   0xFFFFFL   /* maximum size for byte granular segment */
05313   #define GRANULARITY_SHIFT   16  /* shift for limit --> granularity */
05314   #define OFFSET_HIGH_SHIFT   16  /* shift for (gate) offset --> offset_high */
05315   #define PAGE_GRAN_SHIFT     12  /* extra shift for page granular limits */
05316
05317   /* Type-byte bits. */
05318   #define DESC_386_BIT      0x08  /* 386 types are obtained by ORing with this */
05319                                   /* LDT's and TASK_GATE's don't need it */
05320
05321   /* Granularity byte. */
05322   #define GRANULAR          0x80  /* set for 4K granularilty */
05323   #define DEFAULT           0x40  /* set for 32-bit defaults (executable seg) */
05324   #define BIG               0x40  /* set for "BIG" (expand-down seg) */
05325   #define AVL               0x10  /* 0 for available */
05326   #define LIMIT_HIGH        0x0F  /* mask for high bits of limit */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/kernel/sconst.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

05400   ! Miscellaneous constants used in assembler code.
05401   W               =       _WORD_SIZE      ! Machine word size.
05402
05403   ! Offsets in struct proc. They MUST match proc.h.
05404   P_STACKBASE     =       0
05405   #if _WORD_SIZE == 2
05406   ESREG           =       P_STACKBASE
05407   #else
05408   GSREG           =       P_STACKBASE
05409   FSREG           =       GSREG + 2       ! 386 introduces FS and GS segments
```

```
05410   ESREG           =       FSREG + 2
05411   #endif
05412   DSREG           =       ESREG + 2
05413   DIREG           =       DSREG + 2
05414   SIREG           =       DIREG + W
05415   BPREG           =       SIREG + W
05416   STREG           =       BPREG + W       ! hole for another SP
05417   BXREG           =       STREG + W
05418   DXREG           =       BXREG + W
05419   CXREG           =       DXREG + W
05420   AXREG           =       CXREG + W
05421   RETADR          =       AXREG + W       ! return address for save() call
05422   PCREG           =       RETADR + W
05423   CSREG           =       PCREG + W
05424   PSWREG          =       CSREG + W
05425   SPREG           =       PSWREG + W
05426   SSREG           =       SPREG + W
05427   P_STACKTOP      =       SSREG + W
05428   P_LDT_SEL       =       P_STACKTOP
05429   P_LDT           =       P_LDT_SEL + W
05430
05431   #if _WORD_SIZE == 2
05432   Msize           =       12              ! size of a message in 16-bit words
05433   #else
05434   Msize           =       9               ! size of a message in 32-bit words
05435   #endif
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                            src/kernel/assert.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

05500   /*
05501   assert.h
05502   */
05503   #ifndef ASSERT_H
05504   #define ASSERT_H
05505
05506   #if DEBUG
05507
05508   #define INIT_ASSERT     static char *assert_file= __FILE__;
05509
05510   void bad_assertion(char *file, int line, char *what);
05511   void bad_compare(char *file, int line, int lhs, char *what, int rhs);
05512
05513   #define assert(x)       (!(x) ? bad_assertion(assert_file, __LINE__, #x) \
05514                                                           : (void) 0)
05515   #define compare(a,t,b)  (!((a) t (b)) ? bad_compare(assert_file, __LINE__, \
05516                                   (a), #a " " #t " " #b, (b)) : (void) 0)
05517   #else /* !DEBUG */
05518
05519   #define INIT_ASSERT     /* nothing */
05520
05521   #define assert(x)       (void)0
05522   #define compare(a,t,b)  (void)0
05523
05524   #endif /* !DEBUG */
```



```
05525
05526   #endif /* ASSERT_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                            src/kernel/table.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

05600   /* The object file of "table.c" contains all the data.  In the *.h files,
05601    * declared variables appear with EXTERN in front of them, as in
05602    *
05603    *    EXTERN int x;
05604    *
05605    * Normally EXTERN is defined as extern, so when they are included in another
05606    * file, no storage is allocated.  If the EXTERN were not present, but just
05607    * say,
05608    *
05609    *    int x;
05610    *
05611    * then including this file in several source files would cause 'x' to be
05612    * declared several times.  While some linkers accept this, others do not,
05613    * so they are declared extern when included normally.  However, it must
05614    * be declared for real somewhere.  That is done here, by redefining
05615    * EXTERN as the null string, so the inclusion of all the *.h files in
05616    * table.c actually generates storage for them.  All the initialized
05617    * variables are also declared here, since
05618    *
05619    * extern int x = 4;
05620    *
05621    * is not allowed.  If such variables are shared, they must also be declared
05622    * in one of the *.h files without the initialization.
05623    */
05624
05625   #define _TABLE
05626
05627   #include "kernel.h"
05628   #include <termios.h>
05629   #include <minix/com.h>
05630   #include "proc.h"
05631   #include "tty.h"
05632
05633   /* The startup routine of each task is given below, from -NR_TASKS upwards.
05634    * The order of the names here MUST agree with the numerical values assigned to
05635    * the tasks in <minix/com.h>.
05636    */
05637   #define SMALL_STACK     (128 * sizeof(char *))
05638
05639   #define TTY_STACK       (3 * SMALL_STACK)
05640   #define SYN_ALRM_STACK  SMALL_STACK
05641
05642   #define DP8390_STACK    (SMALL_STACK * ENABLE_NETWORKING)
05643
05644   #if (CHIP == INTEL)
05645   #define IDLE_STACK      ((3+3+4) * sizeof(char *))  /* 3 intr, 3 temps, 4 db */
05646   #else
05647   #define IDLE_STACK      SMALL_STACK
05648   #endif
05649
```

```
05650   #define PRINTER_STACK   SMALL_STACK
05651
05652   #if (CHIP == INTEL)
05653   #define WINCH_STACK     (2 * SMALL_STACK * ENABLE_WINI)
05654   #else
05655   #define WINCH_STACK     (3 * SMALL_STACK * ENABLE_WINI)
05656   #endif
05657
05658   #if (MACHINE == ATARI)
05659   #define SCSI_STACK      (3 * SMALL_STACK)
05660   #endif
05661
05662   #if (MACHINE == IBM_PC)
05663   #define SCSI_STACK      (2 * SMALL_STACK * ENABLE_SCSI)
05664   #endif
05665
05666   #define CDROM_STACK     (4 * SMALL_STACK * ENABLE_CDROM)
05667   #define AUDIO_STACK     (4 * SMALL_STACK * ENABLE_AUDIO)
05668   #define MIXER_STACK     (4 * SMALL_STACK * ENABLE_AUDIO)
05669
05670   #define FLOP_STACK      (3 * SMALL_STACK)
05671   #define MEM_STACK       SMALL_STACK
05672   #define CLOCK_STACK     SMALL_STACK
05673   #define SYS_STACK       SMALL_STACK
05674   #define HARDWARE_STACK  0               /* dummy task, uses kernel stack */
05675
05676
05677   #define TOT_STACK_SPACE         (TTY_STACK + DP8390_STACK + SCSI_STACK + \
05678           SYN_ALRM_STACK + IDLE_STACK + HARDWARE_STACK + PRINTER_STACK + \
05679           WINCH_STACK + FLOP_STACK + MEM_STACK + CLOCK_STACK + SYS_STACK + \
05680           CDROM_STACK + AUDIO_STACK + MIXER_STACK)
05681
05682
05683   /* SCSI, CDROM and AUDIO may in the future have different choices like
05684    * WINCHESTER, but for now the choice is fixed.
05685    */
05686   #define scsi_task       aha_scsi_task
05687   #define cdrom_task      mcd_task
05688   #define audio_task      dsp_task
05689
05690
05691   /*
05692    * Some notes about the following table:
05693    *  1) The tty_task should always be first so that other tasks can use printf
05694    *     if their initialisation has problems.
05695    *  2) If you add a new kernel task, add it before the printer task.
05696    *  3) The task name is used for the process name (p_name).
05697    */
05698
05699   PUBLIC struct tasktab tasktab[] = {
05700           { tty_task,             TTY_STACK,      "TTY"           },
05701   #if ENABLE_NETWORKING
05702           { dp8390_task,          DP8390_STACK,   "DP8390"        },
05703   #endif
05704   #if ENABLE_CDROM
05705           { cdrom_task,           CDROM_STACK,    "CDROM"         },
05706   #endif
05707   #if ENABLE_AUDIO
05708           { audio_task,           AUDIO_STACK,    "AUDIO"         },
05709           { mixer_task,           MIXER_STACK,    "MIXER"         },
```



```
05710   #endif
05711   #if ENABLE_SCSI
05712           { scsi_task,            SCSI_STACK,     "SCSI"          },
05713   #endif
05714   #if ENABLE_WINI
05715           { winchester_task,      WINCH_STACK,    "WINCH"         },
05716   #endif
05717           { syn_alrm_task,        SYN_ALRM_STACK, "SYN_AL"        },
05718           { idle_task,            IDLE_STACK,     "IDLE"          },
05719           { printer_task,         PRINTER_STACK,  "PRINTER"       },
05720           { floppy_task,          FLOP_STACK,     "FLOPPY"        },
05721           { mem_task,             MEM_STACK,      "MEMORY"        },
05722           { clock_task,           CLOCK_STACK,    "CLOCK"         },
05723           { sys_task,             SYS_STACK,      "SYS"           },
05724           { 0,                    HARDWARE_STACK, "HARDWAR"       },
05725           { 0,                    0,              "MM"            },
05726           { 0,                    0,              "FS"            },
05727   #if ENABLE_NETWORKING
05728           { 0,                    0,              "INET"          },
05729   #endif
05730           { 0,                    0,              "INIT"          },
05731   };
05732
05733   /* Stack space for all the task stacks.  (Declared as (char *) to align it.) */
05734   PUBLIC char *t_stack[TOT_STACK_SPACE / sizeof(char *)];
05735
05736   /*
05737    * The number of kernel tasks must be the same as NR_TASKS.
05738    * If NR_TASKS is not correct then you will get the compile error:
05739    *   "array size is negative"
05740    */
05741
05742   #define NKT (sizeof tasktab / sizeof (struct tasktab) - (INIT_PROC_NR + 1))
05743
05744   extern int dummy_tasktab_check[NR_TASKS == NKT ? 1 : -1];
```

++
src/kernel/mpx.s
++

```
05800   #
05801   ! Chooses between the 8086 and 386 versions of the Minix startup code.
05802
05803   #include <minix/config.h>
05804   #if _WORD_SIZE == 2
05805   #include "mpx88.s"
05806   #else
05807   #include "mpx386.s"
05808   #endif
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                              src/kernel/mpx386.s
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

05900   #
05901   ! This file contains the assembler startup code for Minix and the 32-bit
05902   ! interrupt handlers.  It cooperates with start.c to set up a good
05903   ! environment for main().
05904
05905   ! This file is part of the lowest layer of the MINIX kernel.  The other part
05906   ! is "proc.c".  The lowest layer does process switching and message handling.
05907
05908   ! Every transition to the kernel goes through this file.  Transitions are
05909   ! caused by sending/receiving messages and by most interrupts.  (RS232
05910   ! interrupts may be handled in the file "rs2.s" and then they rarely enter
05911   ! the kernel.)
05912
05913   ! Transitions to the kernel may be nested.  The initial entry may be with a
05914   ! system call, exception or hardware interrupt; reentries may only be made
05915   ! by hardware interrupts.  The count of reentries is kept in "k_reenter".
05916   ! It is important for deciding whether to switch to the kernel stack and
05917   ! for protecting the message passing code in "proc.c".
05918
05919   ! For the message passing trap, most of the machine state is saved in the
05920   ! proc table.  (Some of the registers need not be saved.)  Then the stack is
05921   ! switched to "k_stack", and interrupts are reenabled.  Finally, the system
05922   ! call handler (in C) is called.  When it returns, interrupts are disabled
05923   ! again and the code falls into the restart routine, to finish off held-up
05924   ! interrupts and run the process or task whose pointer is in "proc_ptr".
05925
05926   ! Hardware interrupt handlers do the same, except  (1) The entire state must
05927   ! be saved.  (2) There are too many handlers to do this inline, so the save
05928   ! routine is called.  A few cycles are saved by pushing the address of the
05929   ! appropiate restart routine for a return later.  (3) A stack switch is
05930   ! avoided when the stack is already switched.  (4) The (master) 8259 interrupt
05931   ! controller is reenabled centrally in save().  (5) Each interrupt handler
05932   ! masks its interrupt line using the 8259 before enabling (other unmasked)
05933   ! interrupts, and unmasks it after servicing the interrupt.  This limits the
05934   ! nest level to the number of lines and protects the handler from itself.
05935
05936   ! For communication with the boot monitor at startup time some constant
05937   ! data are compiled into the beginning of the text segment. This facilitates
05938   ! reading the data at the start of the boot process, since only the first
05939   ! sector of the file needs to be read.
05940
05941   ! Some data storage is also allocated at the end of this file. This data
05942   ! will be at the start of the data segment of the kernel and will be read
05943   ! and modified by the boot monitor before the kernel starts.
05944
05945   ! sections
05946
05947   .sect .text
05948   begtext:
05949   .sect .rom
05950   begrom:
05951   .sect .data
05952   begdata:
05953   .sect .bss
05954   begbss:
```



```
05955
05956   #include <minix/config.h>
05957   #include <minix/const.h>
05958   #include <minix/com.h>
05959   #include "const.h"
05960   #include "protect.h"
05961   #include "sconst.h"
05962
05963   /* Selected 386 tss offsets. */
05964   #define TSS3_S_SP0      4
05965
05966   ! Exported functions
05967   ! Note: in assembly language the .define statement applied to a function name
05968   ! is loosely equivalent to a prototype in C code -- it makes it possible to
05969   ! link to an entity declared in the assembly code but does not create
05970   ! the entity.
05971
05972   .define _idle_task
05973   .define _restart
05974   .define save
05975
05976   .define _divide_error
05977   .define _single_step_exception
05978   .define _nmi
05979   .define _breakpoint_exception
05980   .define _overflow
05981   .define _bounds_check
05982   .define _inval_opcode
05983   .define _copr_not_available
05984   .define _double_fault
05985   .define _copr_seg_overrun
05986   .define _inval_tss
05987   .define _segment_not_present
05988   .define _stack_exception
05989   .define _general_protection
05990   .define _page_fault
05991   .define _copr_error
05992
05993   .define _hwint00        ! handlers for hardware interrupts
05994   .define _hwint01
05995   .define _hwint02
05996   .define _hwint03
05997   .define _hwint04
05998   .define _hwint05
05999   .define _hwint06
06000   .define _hwint07
06001   .define _hwint08
06002   .define _hwint09
06003   .define _hwint10
06004   .define _hwint11
06005   .define _hwint12
06006   .define _hwint13
06007   .define _hwint14
06008   .define _hwint15
06009
06010   .define _s_call
06011   .define _p_s_call
06012   .define _level0_call
06013
06014   ! Imported functions.
```

```
06015
06016   .extern _cstart
06017   .extern _main
06018   .extern _exception
06019   .extern _interrupt
06020   .extern _sys_call
06021   .extern _unhold
06022
06023   ! Exported variables.
06024   ! Note: when used with a variable the .define does not reserve storage,
06025   ! it makes the name externally visible so it may be linked to.
06026
06027   .define begbss
06028   .define begdata
06029   .define _sizes
06030
06031   ! Imported variables.
06032
06033   .extern _gdt
06034   .extern _code_base
06035   .extern _data_base
06036   .extern _held_head
06037   .extern _k_reenter
06038   .extern _pc_at
06039   .extern _proc_ptr
06040   .extern _ps_mca
06041   .extern _tss
06042   .extern _level0_func
06043   .extern _mon_sp
06044   .extern _mon_return
06045   .extern _reboot_code
06046
06047   .sect .text
06048   !*===========================================================================*
06049   !*                                MINIX                                      *
06050   !*===========================================================================*
06051   MINIX:                          ! this is the entry point for the MINIX kernel
06052           jmp     over_flags      ! skip over the next few bytes
06053           .data2  CLICK_SHIFT     ! for the monitor: memory granularity
06054   flags:
06055           .data2  0x002D          ! boot monitor flags:
06056                                   !       call in 386 mode, make stack,
06057                                   !       load high, will return
06058           nop                     ! extra byte to sync up disassembler
06059   over_flags:
06060
06061   ! Set up a C stack frame on the monitor stack.  (The monitor sets cs and ds
06062   ! right.  The ss descriptor still references the monitor data segment.)
06063           movzx   esp, sp         ! monitor stack is a 16 bit stack
06064           push    ebp
06065           mov     ebp, esp
06066           push    esi
06067           push    edi
06068           cmp     4(ebp), 0       ! nonzero if return possible
06069           jz      noret
06070           inc     (_mon_return)
06071   noret:  mov     (_mon_sp), esp  ! save stack pointer for later return
06072
06073   ! Copy the monitor global descriptor table to the address space of kernel and
06074   ! switch over to it.  Prot_init() can then update it with immediate effect.
```



```
06075
06076           sgdt    (_gdt+GDT_SELECTOR)             ! get the monitor gdtr
06077           mov     esi, (_gdt+GDT_SELECTOR+2)      ! absolute address of GDT
06078           mov     ebx, _gdt                       ! address of kernel GDT
06079           mov     ecx, 8*8                        ! copying eight descriptors
06080   copygdt:
06081    eseg   movb    al, (esi)
06082           movb    (ebx), al
06083           inc     esi
06084           inc     ebx
06085           loop    copygdt
06086           mov     eax, (_gdt+DS_SELECTOR+2)       ! base of kernel data
06087           and     eax, 0x00FFFFFF                 ! only 24 bits
06088           add     eax, _gdt                       ! eax = vir2phys(gdt)
06089           mov     (_gdt+GDT_SELECTOR+2), eax      ! set base of GDT
06090           lgdt    (_gdt+GDT_SELECTOR)             ! switch over to kernel GDT
06091
06092   ! Locate boot parameters, set up kernel segment registers and stack.
06093           mov     ebx, 8(ebp)     ! boot parameters offset
06094           mov     edx, 12(ebp)    ! boot parameters length
06095           mov     ax, ds          ! kernel data
06096           mov     es, ax
06097           mov     fs, ax
06098           mov     gs, ax
06099           mov     ss, ax
06100           mov     esp, k_stktop   ! set sp to point to the top of kernel stack
06101
06102   ! Call C startup code to set up a proper environment to run main().
06103           push    edx
06104           push    ebx
06105           push    SS_SELECTOR
06106           push    MON_CS_SELECTOR
06107           push    DS_SELECTOR
06108           push    CS_SELECTOR
06109           call    _cstart         ! cstart(cs, ds, mcs, mds, parmoff, parmlen)
06110           add     esp, 6*4
06111
06112   ! Reload gdtr, idtr and the segment registers to global descriptor table set
06113   ! up by prot_init().
06114
06115           lgdt    (_gdt+GDT_SELECTOR)
06116           lidt    (_gdt+IDT_SELECTOR)
06117
06118           jmpf    CS_SELECTOR:csinit
06119   csinit:
06120       o16 mov     ax, DS_SELECTOR
06121           mov     ds, ax
06122           mov     es, ax
06123           mov     fs, ax
06124           mov     gs, ax
06125           mov     ss, ax
06126       o16 mov     ax, TSS_SELECTOR        ! no other TSS is used
06127           ltr     ax
06128           push    0                       ! set flags to known good state
06129           popf                            ! esp, clear nested task and int enable
06130
06131           jmp     _main                   ! main()
06132
06133
06134   !*===========================================================================*
```

```
06135 !*                              interrupt handlers                        *
06136 !*              interrupt handlers for 386 32-bit protected mode           *
06137 !*=========================================================================*
06138
06139 !*=========================================================================*
06140 !*                              hwint00 - 07                               *
06141 !*=========================================================================*
06142 ! Note this is a macro, it looks like a subroutine.
06143 #define hwint_master(irq)       \
06144         call    save                    /* save interrupted process state */;\
06145         inb     INT_CTLMASK                                                 ;\
06146         orb     al, [1<<irq]                                                ;\
06147         outb    INT_CTLMASK             /* disable the irq                */;\
06148         movb    al, ENABLE                                                  ;\
06149         outb    INT_CTL                 /* reenable master 8259           */;\
06150         sti                             /* enable interrupts              */;\
06151         push    irq                     /* irq                            */;\
06152         call    (_irq_table + 4*irq)    /* eax = (*irq_table[irq])(irq)   */;\
06153         pop     ecx                                                         ;\
06154         cli                             /* disable interrupts             */;\
06155         test    eax, eax                /* need to reenable irq?          */;\
06156         jz      0f                                                          ;\
06157         inb     INT_CTLMASK                                                 ;\
06158         andb    al, ~[1<<irq]                                               ;\
06159         outb    INT_CTLMASK             /* enable the irq                 */;\
06160 0:      ret                             /* restart (another) process      */
06161
06162 ! Each of these entry points is an expansion of the hwint_master macro
06163         .align  16
06164 _hwint00:               ! Interrupt routine for irq 0 (the clock).
06165         hwint_master(0)
06166
06167         .align  16
06168 _hwint01:               ! Interrupt routine for irq 1 (keyboard)
06169         hwint_master(1)
06170
06171         .align  16
06172 _hwint02:               ! Interrupt routine for irq 2 (cascade!)
06173         hwint_master(2)
06174
06175         .align  16
06176 _hwint03:               ! Interrupt routine for irq 3 (second serial)
06177         hwint_master(3)
06178
06179         .align  16
06180 _hwint04:               ! Interrupt routine for irq 4 (first serial)
06181         hwint_master(4)
06182
06183         .align  16
06184 _hwint05:               ! Interrupt routine for irq 5 (XT winchester)
06185         hwint_master(5)
06186
06187         .align  16
06188 _hwint06:               ! Interrupt routine for irq 6 (floppy)
06189         hwint_master(6)
06190
06191         .align  16
06192 _hwint07:               ! Interrupt routine for irq 7 (printer)
06193         hwint_master(7)
06194
```



```
06195 !*=========================================================================*
06196 !*                              hwint08 - 15                               *
06197 !*=========================================================================*
06198 ! Note this is a macro, it looks like a subroutine.
06199 #define hwint_slave(irq)        \
06200         call    save                    /* save interrupted process state */;\
06201         inb     INT2_CTLMASK                                                ;\
06202         orb     al, [1<<[irq-8]]                                            ;\
06203         outb    INT2_CTLMASK            /* disable the irq                */;\
06204         movb    al, ENABLE                                                  ;\
06205         outb    INT_CTL                 /* reenable master 8259           */;\
06206         jmp     .+2                     /* delay                          */;\
06207         outb    INT2_CTL                /* reenable slave 8259            */;\
06208         sti                             /* enable interrupts              */;\
06209         push    irq                     /* irq                            */;\
06210         call    (_irq_table + 4*irq)    /* eax = (*irq_table[irq])(irq)   */;\
06211         pop     ecx                                                         ;\
06212         cli                             /* disable interrupts             */;\
06213         test    eax, eax                /* need to reenable irq?          */;\
06214         jz      0f                                                          ;\
06215         inb     INT2_CTLMASK                                                ;\
06216         andb    al, ~[1<<[irq-8]]                                           ;\
06217         outb    INT2_CTLMASK            /* enable the irq                 */;\
06218 0:      ret                             /* restart (another) process      */
06219
06220 ! Each of these entry points is an expansion of the hwint_slave macro
06221         .align  16
06222 _hwint08:               ! Interrupt routine for irq 8 (realtime clock)
06223         hwint_slave(8)
06224
06225         .align  16
06226 _hwint09:               ! Interrupt routine for irq 9 (irq 2 redirected)
06227         hwint_slave(9)
06228
06229         .align  16
06230 _hwint10:               ! Interrupt routine for irq 10
06231         hwint_slave(10)
06232
06233         .align  16
06234 _hwint11:               ! Interrupt routine for irq 11
06235         hwint_slave(11)
06236
06237         .align  16
06238 _hwint12:               ! Interrupt routine for irq 12
06239         hwint_slave(12)
06240
06241         .align  16
06242 _hwint13:               ! Interrupt routine for irq 13 (FPU exception)
06243         hwint_slave(13)
06244
06245         .align  16
06246 _hwint14:               ! Interrupt routine for irq 14 (AT winchester)
06247         hwint_slave(14)
06248
06249         .align  16
06250 _hwint15:               ! Interrupt routine for irq 15
06251         hwint_slave(15)
06252
06253 !*=========================================================================*
06254 !*                              save                                       *
```

```
06255   !*===========================================================================*
06256   ! Save for protected mode.
06257   ! This is much simpler than for 8086 mode, because the stack already points
06258   ! into the process table, or has already been switched to the kernel stack.
06259
06260           .align  16
06261   save:
06262           cld                     ! set direction flag to a known value
06263           pushad                  ! save "general" registers
06264       o16 push    ds              ! save ds
06265       o16 push    es              ! save es
06266       o16 push    fs              ! save fs
06267       o16 push    gs              ! save gs
06268           mov     dx, ss          ! ss is kernel data segment
06269           mov     ds, dx          ! load rest of kernel segments
06270           mov     es, dx          ! kernel does not use fs, gs
06271           mov     eax, esp        ! prepare to return
06272           incb    (_k_reenter)    ! from -1 if not reentering
06273           jnz     set_restart1    ! stack is already kernel stack
06274           mov     esp, k_stktop
06275           push    _restart        ! build return address for int handler
06276           xor     ebp, ebp        ! for stacktrace
06277           jmp     RETADR-P_STACKBASE(eax)
06278
06279           .align  4
06280   set_restart1:
06281           push    restart1
06282           jmp     RETADR-P_STACKBASE(eax)
06283
06284   !*===========================================================================*
06285   !*                              _s_call                                      *
06286   !*===========================================================================*
06287           .align  16
06288   _s_call:
06289   _p_s_call:
06290           cld                     ! set direction flag to a known value
06291           sub     esp, 6*4        ! skip RETADR, eax, ecx, edx, ebx, est
06292           push    ebp             ! stack already points into proc table
06293           push    esi
06294           push    edi
06295       o16 push    ds
06296       o16 push    es
06297       o16 push    fs
06298       o16 push    gs
06299           mov     dx, ss
06300           mov     ds, dx
06301           mov     es, dx
06302           incb    (_k_reenter)
06303           mov     esi, esp        ! assumes P_STACKBASE == 0
06304           mov     esp, k_stktop
06305           xor     ebp, ebp        ! for stacktrace
06306                                   ! end of inline save
06307           sti                     ! allow SWITCHER to be interrupted
06308                                   ! now set up parameters for sys_call()
06309           push    ebx             ! pointer to user message
06310           push    eax             ! src/dest
06311           push    ecx             ! SEND/RECEIVE/BOTH
06312           call    _sys_call       ! sys_call(function, src_dest, m_ptr)
06313                                   ! caller is now explicitly in proc_ptr
06314           mov     AXREG(esi), eax ! sys_call MUST PRESERVE si
```



```
06315           cli                     ! disable interrupts
06316
06317   ! Fall into code to restart proc/task running.
06318
06319   !*===========================================================================*
06320   !*                              restart                                      *
06321   !*===========================================================================*
06322   _restart:
06323
06324   ! Flush any held-up interrupts.
06325   ! This reenables interrupts, so the current interrupt handler may reenter.
06326   ! This does not matter, because the current handler is about to exit and no
06327   ! other handlers can reenter since flushing is only done when k_reenter == 0.
06328
06329           cmp     (_held_head), 0 ! do fast test to usually avoid function call
06330           jz      over_call_unhold
06331           call    _unhold         ! this is rare so overhead acceptable
06332   over_call_unhold:
06333           mov     esp, (_proc_ptr)        ! will assume P_STACKBASE == 0
06334           lldt    P_LDT_SEL(esp)          ! enable segment descriptors for task
06335           lea     eax, P_STACKTOP(esp)    ! arrange for next interrupt
06336           mov     (_tss+TSS3_S_SP0), eax  ! to save state in process table
06337   restart1:
06338           decb    (_k_reenter)
06339       o16 pop     gs
06340       o16 pop     fs
06341       o16 pop     es
06342       o16 pop     ds
06343           popad
06344           add     esp, 4          ! skip return adr
06345           iretd                   ! continue process
06346
06347   !*===========================================================================*
06348   !*                              exception handlers                           *
06349   !*===========================================================================*
06350   _divide_error:
06351           push    DIVIDE_VECTOR
06352           jmp     exception
06353
06354   _single_step_exception:
06355           push    DEBUG_VECTOR
06356           jmp     exception
06357
06358   _nmi:
06359           push    NMI_VECTOR
06360           jmp     exception
06361
06362   _breakpoint_exception:
06363           push    BREAKPOINT_VECTOR
06364           jmp     exception
06365
06366   _overflow:
06367           push    OVERFLOW_VECTOR
06368           jmp     exception
06369
06370   _bounds_check:
06371           push    BOUNDS_VECTOR
06372           jmp     exception
06373
06374   _inval_opcode:
```

```
06375           push    INVAL_OP_VECTOR
06376           jmp     exception
06377
06378   _copr_not_available:
06379           push    COPROC_NOT_VECTOR
06380           jmp     exception
06381
06382   _double_fault:
06383           push    DOUBLE_FAULT_VECTOR
06384           jmp     errexception
06385
06386   _copr_seg_overrun:
06387           push    COPROC_SEG_VECTOR
06388           jmp     exception
06389
06390   _inval_tss:
06391           push    INVAL_TSS_VECTOR
06392           jmp     errexception
06393
06394   _segment_not_present:
06395           push    SEG_NOT_VECTOR
06396           jmp     errexception
06397
06398   _stack_exception:
06399           push    STACK_FAULT_VECTOR
06400           jmp     errexception
06401
06402   _general_protection:
06403           push    PROTECTION_VECTOR
06404           jmp     errexception
06405
06406   _page_fault:
06407           push    PAGE_FAULT_VECTOR
06408           jmp     errexception
06409
06410   _copr_error:
06411           push    COPROC_ERR_VECTOR
06412           jmp     exception
06413
06414   !*===========================================================================*
06415   !*                              exception                                    *
06416   !*===========================================================================*
06417   ! This is called for all exceptions which do not push an error code.
06418
06419           .align  16
06420   exception:
06421    sseg   mov     (trap_errno), 0         ! clear trap_errno
06422    sseg   pop     (ex_number)
06423           jmp     exception1
06424
06425   !*===========================================================================*
06426   !*                              errexception                                 *
06427   !*===========================================================================*
06428   ! This is called for all exceptions which push an error code.
06429
06430           .align  16
06431   errexception:
06432    sseg   pop     (ex_number)
06433    sseg   pop     (trap_errno)
06434   exception1:                             ! Common for all exceptions.
```



```
06435           push    eax                     ! eax is scratch register
06436           mov     eax, 0+4(esp)           ! old eip
06437    sseg   mov     (old_eip), eax
06438           movzx   eax, 4+4(esp)           ! old cs
06439    sseg   mov     (old_cs), eax
06440           mov     eax, 8+4(esp)           ! old eflags
06441    sseg   mov     (old_eflags), eax
06442           pop     eax
06443           call    save
06444           push    (old_eflags)
06445           push    (old_cs)
06446           push    (old_eip)
06447           push    (trap_errno)
06448           push    (ex_number)
06449           call    _exception              ! (ex_number, trap_errno, old_eip,
06450                                           !       old_cs, old_eflags)
06451           add     esp, 5*4
06452           cli
06453           ret
06454
06455   !*===========================================================================*
06456   !*                              level0_call                                  *
06457   !*===========================================================================*
06458   _level0_call:
06459           call    save
06460           jmp     (_level0_func)
06461
06462   !*===========================================================================*
06463   !*                              idle_task                                    *
06464   !*===========================================================================*
06465   _idle_task:                     ! executed when there is no work
06466           jmp     _idle_task      ! a "hlt" before this fails in protected mode
06467
06468   !*===========================================================================*
06469   !*                              data                                         *
06470   !*===========================================================================*
06471   ! These declarations assure that storage will be allocated at the very
06472   ! beginning of the kernel data section, so the boot monitor can be easily
06473   ! told how to patch these locations. Note that the magic number is put
06474   ! here by the compiler, but will be read by, and then overwritten by,
06475   ! the boot monitor. When the kernel starts the sizes array will be
06476   ! found here, as if it had been initialized by the compiler.
06477
06478   .sect .rom      ! Before the string table please
06479   _sizes:                         ! sizes of kernel, mm, fs filled in by boot
06480           .data2  0x526F          ! this must be the first data entry (magic #)
06481           .space  16*2*2-2        ! monitor uses previous word and this space
06482                                   ! extra space allows for additional servers
06483   .sect .bss
06484   k_stack:
06485           .space  K_STACK_BYTES   ! kernel stack
06486   k_stktop:                       ! top of kernel stack
06487           .comm   ex_number, 4
06488           .comm   trap_errno, 4
06489           .comm   old_eip, 4
06490           .comm   old_cs, 4
06491           .comm   old_eflags, 4
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/kernel/start.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
06500   /* This file contains the C startup code for Minix on Intel processors.
06501    * It cooperates with mpx.s to set up a good environment for main().
06502    *
06503    * This code runs in real mode for a 16 bit kernel and may have to switch
06504    * to protected mode for a 286.
06505    *
06506    * For a 32 bit kernel this already runs in protected mode, but the selectors
06507    * are still those given by the BIOS with interrupts disabled, so the
06508    * descriptors need to be reloaded and interrupt descriptors made.
06509    */
06510
06511   #include "kernel.h"
06512   #include <stdlib.h>
06513   #include <minix/boot.h>
06514   #include "protect.h"
06515
06516   PRIVATE char k_environ[256];    /* environment strings passed by loader */
06517
06518   FORWARD _PROTOTYPE( int k_atoi, (char *s) );
06519
06520
06521   /*===========================================================================*
06522    *                              cstart                                       *
06523    *===========================================================================*/
06524   PUBLIC void cstart(cs, ds, mcs, mds, parmoff, parmsize)
06525   U16_t cs, ds;                   /* Kernel code and data segment */
06526   U16_t mcs, mds;                 /* Monitor code and data segment */
06527   U16_t parmoff, parmsize;        /* boot parameters offset and length */
06528   {
06529   /* Perform system initializations prior to calling main(). */
06530
06531     register char *envp;
06532     phys_bytes mcode_base, mdata_base;
06533     unsigned mon_start;
06534
06535     /* Record where the kernel and the monitor are. */
06536     code_base = seg2phys(cs);
06537     data_base = seg2phys(ds);
06538     mcode_base = seg2phys(mcs);
06539     mdata_base = seg2phys(mds);
06540
06541     /* Initialize protected mode descriptors. */
06542     prot_init();
06543
06544     /* Copy the boot parameters to kernel memory. */
06545     if (parmsize > sizeof k_environ - 2) parmsize = sizeof k_environ - 2;
06546     phys_copy(mdata_base + parmoff, vir2phys(k_environ), (phys_bytes) parmsize);
06547
06548     /* Convert important boot environment variables. */
06549     boot_parameters.bp_rootdev = k_atoi(k_getenv("rootdev"));
06550     boot_parameters.bp_ramimagedev = k_atoi(k_getenv("ramimagedev"));
06551     boot_parameters.bp_ramsize = k_atoi(k_getenv("ramsize"));
06552     boot_parameters.bp_processor = k_atoi(k_getenv("processor"));
06553
06554     /* Type of VDU: */
```



```
06555     envp = k_getenv("video");
06556     if (strcmp(envp, "ega") == 0) ega = TRUE;
06557     if (strcmp(envp, "vga") == 0) vga = ega = TRUE;
06558
06559     /* Memory sizes: */
06560     low_memsize = k_atoi(k_getenv("memsize"));
06561     ext_memsize = k_atoi(k_getenv("emssize"));
06562
06563     /* Processor? */
06564     processor = boot_parameters.bp_processor;     /* 86, 186, 286, 386, ... */
06565
06566     /* XT, AT or MCA bus? */
06567     envp = k_getenv("bus");
06568     if (envp == NIL_PTR || strcmp(envp, "at") == 0) {
06569           pc_at = TRUE;
06570     } else
06571     if (strcmp(envp, "mca") == 0) {
06572           pc_at = ps_mca = TRUE;
06573     }
06574
06575     /* Decide if mode is protected. */
06576   #if _WORD_SIZE == 2
06577     protected_mode = processor >= 286;
06578   #endif
06579
06580     /* Is there a monitor to return to?  If so then keep it safe. */
06581     if (!protected_mode) mon_return = 0;
06582     mon_start = mcode_base / 1024;
06583     if (mon_return && low_memsize > mon_start) low_memsize = mon_start;
06584
06585     /* Return to assembler code to switch to protected mode (if 286), reload
06586      * selectors and call main().
06587      */
06588   }


06591   /*===========================================================================*
06592    *                              k_atoi                                       *
06593    *===========================================================================*/
06594   PRIVATE int k_atoi(s)
06595   register char *s;
06596   {
06597   /* Convert string to integer. */
06598
06599     return strtol(s, (char **) NULL, 10);
06600   }


06603   /*===========================================================================*
06604    *                              k_getenv                                     *
06605    *===========================================================================*/
06606   PUBLIC char *k_getenv(name)
06607   char *name;
06608   {
06609   /* Get environment value - kernel version of getenv to avoid setting up the
06610    * usual environment array.
06611    */
06612
06613     register char *namep;
06614     register char *envp;
```

```
06615
06616       for (envp = k_environ; *envp != 0;) {
06617             for (namep = name; *namep != 0 && *namep == *envp; namep++, envp++)
06618                   ;
06619             if (*namep == '\0' && *envp == '=') return(envp + 1);
06620             while (*envp++ != 0)
06621                   ;
06622       }
06623       return(NIL_PTR);
06624   }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                              src/kernel/main.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

06700   /* This file contains the main program of MINIX.  The routine main()
06701    * initializes the system and starts the ball rolling by setting up the proc
06702    * table, interrupt vectors, and scheduling each task to run to initialize
06703    * itself.
06704    *
06705    * The entries into this file are:
06706    *   main:              MINIX main program
06707    *   panic:             abort MINIX due to a fatal error
06708    */
06709
06710   #include "kernel.h"
06711   #include <signal.h>
06712   #include <unistd.h>
06713   #include <minix/callnr.h>
06714   #include <minix/com.h>
06715   #include "proc.h"
06716
06717
06718   /*===========================================================================*
06719    *                                    main                                   *
06720    *===========================================================================*/
06721   PUBLIC void main()
06722   {
06723   /* Start the ball rolling. */
06724
06725     register struct proc *rp;
06726     register int t;
06727     int sizeindex;
06728     phys_clicks text_base;
06729     vir_clicks text_clicks;
06730     vir_clicks data_clicks;
06731     phys_bytes phys_b;
06732     reg_t ktsb;                 /* kernel task stack base */
06733     struct memory *memp;
06734     struct tasktab *ttp;
06735
06736     /* Initialize the interrupt controller. */
06737     intr_init(1);
06738
06739     /* Interpret memory sizes. */
```



```
06740     mem_init();
06741
06742     /* Clear the process table.
06743      * Set up mappings for proc_addr() and proc_number() macros.
06744      */
06745     for (rp = BEG_PROC_ADDR, t = -NR_TASKS; rp < END_PROC_ADDR; ++rp, ++t) {
06746           rp->p_flags = P_SLOT_FREE;
06747           rp->p_nr = t;           /* proc number from ptr */
06748           (pproc_addr + NR_TASKS)[t] = rp;        /* proc ptr from number */
06749     }
06750
06751     /* Set up proc table entries for tasks and servers.  The stacks of the
06752      * kernel tasks are initialized to an array in data space.  The stacks
06753      * of the servers have been added to the data segment by the monitor, so
06754      * the stack pointer is set to the end of the data segment.  All the
06755      * processes are in low memory on the 8086.  On the 386 only the kernel
06756      * is in low memory, the rest if loaded in extended memory.
06757      */
06758
06759     /* Task stacks. */
06760     ktsb = (reg_t) t_stack;
06761
06762     for (t = -NR_TASKS; t <= LOW_USER; ++t) {
06763           rp = proc_addr(t);                      /* t's process slot */
06764           ttp = &tasktab[t + NR_TASKS];           /* t's task attributes */
06765           strcpy(rp->p_name, ttp->name);
06766           if (t < 0) {
06767                   if (ttp->stksize > 0) {
06768                           rp->p_stguard = (reg_t *) ktsb;
06769                           *rp->p_stguard = STACK_GUARD;
06770                   }
06771                   ktsb += ttp->stksize;
06772                   rp->p_reg.sp = ktsb;
06773                   text_base = code_base >> CLICK_SHIFT;
06774                                           /* tasks are all in the kernel */
06775                   sizeindex = 0;          /* and use the full kernel sizes */
06776                   memp = &mem[0];         /* remove from this memory chunk */
06777           } else {
06778                   sizeindex = 2 * t + 2;  /* MM, FS, INIT have their own sizes */
06779           }
06780           rp->p_reg.pc = (reg_t) ttp->initial_pc;
06781           rp->p_reg.psw = istaskp(rp) ? INIT_TASK_PSW : INIT_PSW;
06782
06783           text_clicks = sizes[sizeindex];
06784           data_clicks = sizes[sizeindex + 1];
06785           rp->p_map[T].mem_phys = text_base;
06786           rp->p_map[T].mem_len  = text_clicks;
06787           rp->p_map[D].mem_phys = text_base + text_clicks;
06788           rp->p_map[D].mem_len  = data_clicks;
06789           rp->p_map[S].mem_phys = text_base + text_clicks + data_clicks;
06790           rp->p_map[S].mem_vir  = data_clicks;    /* empty - stack is in data */
06791           text_base += text_clicks + data_clicks; /* ready for next, if server */
06792           memp->size -= (text_base - memp->base);
06793           memp->base = text_base;                 /* memory no longer free */
06794
06795           if (t >= 0) {
06796                   /* Initialize the server stack pointer.  Take it down one word
06797                    * to give crtso.s something to use as "argc".
06798                    */
06799                   rp->p_reg.sp = (rp->p_map[S].mem_vir +
```

```
06800                                   rp->p_map[S].mem_len) << CLICK_SHIFT;
06801                   rp->p_reg.sp -= sizeof(reg_t);
06802           }
06803
06804   #if _WORD_SIZE == 4
06805           /* Servers are loaded in extended memory if in 386 mode. */
06806           if (t < 0) {
06807                   memp = &mem[1];
06808                   text_base = 0x100000 >> CLICK_SHIFT;
06809           }
06810   #endif
06811           if (!isidlehardware(t)) lock_ready(rp); /* IDLE, HARDWARE neveready */
06812           rp->p_flags = 0;
06813
06814           alloc_segments(rp);
06815     }
06816
06817     proc[NR_TASKS+INIT_PROC_NR].p_pid = 1;/* INIT of course has pid 1 */
06818     bill_ptr = proc_addr(IDLE);           /* it has to point somewhere */
06819     lock_pick_proc();
06820
06821     /* Now go to the assembly code to start running the current process. */
06822     restart();
06823   }


06826   /*===========================================================================*
06827    *                                panic                                      *
06828    *===========================================================================*/
06829   PUBLIC void panic(s,n)
06830   _CONST char *s;
06831   int n;
06832   {
06833   /* The system has run aground of a fatal error.  Terminate execution.
06834    * If the panic originated in MM or FS, the string will be empty and the
06835    * file system already syncked.  If the panic originates in the kernel, we are
06836    * kind of stuck.
06837    */
06838
06839     if (*s != 0) {
06840           printf("\nKernel panic: %s",s);
06841           if (n != NO_NUM) printf(" %d", n);
06842           printf("\n");
06843     }
06844     wreboot(RBT_PANIC);
06845   }
```



```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/kernel/proc.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

06900   /* This file contains essentially all of the process and message handling.
06901    * It has two main entry points from the outside:
06902    *
06903    *   sys_call:   called when a process or task does SEND, RECEIVE or SENDREC
06904    *   interrupt: called by interrupt routines to send a message to task
06905    *
06906    * It also has several minor entry points:
06907    *
06908    *   lock_ready:      put a process on one of the ready queues so it can be run
06909    *   lock_unready:    remove a process from the ready queues
06910    *   lock_sched:      a process has run too long; schedule another one
06911    *   lock_mini_send:  send a message (used by interrupt signals, etc.)
06912    *   lock_pick_proc:  pick a process to run (used by system initialization)
06913    *   unhold:          repeat all held-up interrupts
06914    */
06915
06916   #include "kernel.h"
06917   #include <minix/callnr.h>
06918   #include <minix/com.h>
06919   #include "proc.h"
06920
06921   PRIVATE unsigned char switching;        /* nonzero to inhibit interrupt() */
06922
06923   FORWARD _PROTOTYPE( int mini_send, (struct proc *caller_ptr, int dest,
06924                   message *m_ptr) );
06925   FORWARD _PROTOTYPE( int mini_rec, (struct proc *caller_ptr, int src,
06926                   message *m_ptr) );
06927   FORWARD _PROTOTYPE( void ready, (struct proc *rp) );
06928   FORWARD _PROTOTYPE( void sched, (void) );
06929   FORWARD _PROTOTYPE( void unready, (struct proc *rp) );
06930   FORWARD _PROTOTYPE( void pick_proc, (void) );
06931
06932   #define CopyMess(s,sp,sm,dp,dm) \
06933           cp_mess(s, (sp)->p_map[D].mem_phys, (vir_bytes)sm, (dp)->p_map[D].mem_phys, (vir_bytes)dm)
06934
06935   /*===========================================================================*
06936    *                              interrupt                                    *
06937    *===========================================================================*/
06938   PUBLIC void interrupt(task)
06939   int task;                       /* number of task to be started */
06940   {
06941   /* An interrupt has occurred.  Schedule the task that handles it. */
06942
06943     register struct proc *rp;     /* pointer to task's proc entry */
06944
06945     rp = proc_addr(task);
06946
06947     /* If this call would compete with other process-switching functions, put
06948      * it on the 'held' queue to be flushed at the next non-competing restart().
06949      * The competing conditions are:
06950      * (1) k_reenter == (typeof k_reenter) -1:
06951      *     Call from the task level, typically from an output interrupt
06952      *     routine.  An interrupt handler might reenter interrupt().  Rare,
06953      *     so not worth special treatment.
06954      * (2) k_reenter > 0:
```

```
06955       *     Call from a nested interrupt handler.  A previous interrupt handler
06956       *     might be inside interrupt() or sys_call().
06957       * (3) switching != 0:
06958       *     Some process-switching function other than interrupt() is being
06959       *     called from the task level, typically sched() from CLOCK.  An
06960       *     interrupt handler might call interrupt and pass the k_reenter test.
06961       */
06962      if (k_reenter != 0 || switching) {
06963            lock();
06964            if (!rp->p_int_held) {
06965                    rp->p_int_held = TRUE;
06966                    if (held_head != NIL_PROC)
06967                            held_tail->p_nextheld = rp;
06968                    else
06969                            held_head = rp;
06970                    held_tail = rp;
06971                    rp->p_nextheld = NIL_PROC;
06972            }
06973            unlock();
06974            return;
06975      }
06976
06977      /* If task is not waiting for an interrupt, record the blockage. */
06978      if ( (rp->p_flags & (RECEIVING | SENDING)) != RECEIVING ||
06979          !isrxhardware(rp->p_getfrom)) {
06980            rp->p_int_blocked = TRUE;
06981            return;
06982      }
06983
06984      /* Destination is waiting for an interrupt.
06985       * Send it a message with source HARDWARE and type HARD_INT.
06986       * No more information can be reliably provided since interrupt messages
06987       * are not queued.
06988       */
06989      rp->p_messbuf->m_source = HARDWARE;
06990      rp->p_messbuf->m_type = HARD_INT;
06991      rp->p_flags &= ~RECEIVING;
06992      rp->p_int_blocked = FALSE;
06993
06994       /* Make rp ready and run it unless a task is already running.  This is
06995        * ready(rp) in-line for speed.
06996        */
06997      if (rdy_head[TASK_Q] != NIL_PROC)
06998            rdy_tail[TASK_Q]->p_nextready = rp;
06999      else
07000            proc_ptr = rdy_head[TASK_Q] = rp;
07001      rdy_tail[TASK_Q] = rp;
07002      rp->p_nextready = NIL_PROC;
07003  }

07005  /*===========================================================================*
07006   *                                sys_call                                   *
07007   *===========================================================================*/
07008  PUBLIC int sys_call(function, src_dest, m_ptr)
07009  int function;                   /* SEND, RECEIVE, or BOTH */
07010  int src_dest;                   /* source to receive from or dest to send to */
07011  message *m_ptr;                 /* pointer to message */
07012  {
07013  /* The only system calls that exist in MINIX are sending and receiving
07014   * messages.  These are done by trapping to the kernel with an INT instruction.
```



```
07015   * The trap is caught and sys_call() is called to send or receive a message
07016   * (or both). The caller is always given by proc_ptr.
07017   */
07018
07019    register struct proc *rp;
07020    int n;
07021
07022    /* Check for bad system call parameters. */
07023    if (!isoksrc_dest(src_dest)) return(E_BAD_SRC);
07024    rp = proc_ptr;
07025
07026    if (isuserp(rp) && function != BOTH) return(E_NO_PERM);
07027
07028    /* The parameters are ok. Do the call. */
07029    if (function & SEND) {
07030          /* Function = SEND or BOTH. */
07031          n = mini_send(rp, src_dest, m_ptr);
07032          if (function == SEND || n != OK)
07033                  return(n);      /* done, or SEND failed */
07034    }
07035
07036    /* Function = RECEIVE or BOTH.
07037     * We have checked user calls are BOTH, and trust 'function' otherwise.
07038     */
07039    return(mini_rec(rp, src_dest, m_ptr));
07040  }

07042  /*===========================================================================*
07043   *                                mini_send                                  *
07044   *===========================================================================*/
07045  PRIVATE int mini_send(caller_ptr, dest, m_ptr)
07046  register struct proc *caller_ptr;       /* who is trying to send a message? */
07047  int dest;                       /* to whom is message being sent? */
07048  message *m_ptr;                 /* pointer to message buffer */
07049  {
07050  /* Send a message from 'caller_ptr' to 'dest'. If 'dest' is blocked waiting
07051   * for this message, copy the message to it and unblock 'dest'. If 'dest' is
07052   * not waiting at all, or is waiting for another source, queue 'caller_ptr'.
07053   */
07054
07055    register struct proc *dest_ptr, *next_ptr;
07056    vir_bytes vb;                 /* message buffer pointer as vir_bytes */
07057    vir_clicks vlo, vhi;          /* virtual clicks containing message to send */
07058
07059    /* User processes are only allowed to send to FS and MM.  Check for this. */
07060    if (isuserp(caller_ptr) && !issysentn(dest)) return(E_BAD_DEST);
07061    dest_ptr = proc_addr(dest);   /* pointer to destination's proc entry */
07062    if (dest_ptr->p_flags & P_SLOT_FREE) return(E_BAD_DEST);      /* dead dest */
07063
07064    /* This check allows a message to be anywhere in data or stack or gap.
07065     * It will have to be made more elaborate later for machines which
07066     * don't have the gap mapped.
07067     */
07068    vb = (vir_bytes) m_ptr;
07069    vlo = vb >> CLICK_SHIFT;      /* vir click for bottom of message */
07070    vhi = (vb + MESS_SIZE - 1) >> CLICK_SHIFT;    /* vir click for top of msg */
07071    if (vlo < caller_ptr->p_map[D].mem_vir || vlo > vhi ||
07072        vhi >= caller_ptr->p_map[S].mem_vir + caller_ptr->p_map[S].mem_len)
07073          return(EFAULT);
07074
```

```
07075     /* Check for deadlock by 'caller_ptr' and 'dest' sending to each other. */
07076     if (dest_ptr->p_flags & SENDING) {
07077           next_ptr = proc_addr(dest_ptr->p_sendto);
07078           while (TRUE) {
07079                   if (next_ptr == caller_ptr) return(ELOCKED);
07080                   if (next_ptr->p_flags & SENDING)
07081                           next_ptr = proc_addr(next_ptr->p_sendto);
07082                   else
07083                           break;
07084           }
07085     }
07086
07087     /* Check to see if 'dest' is blocked waiting for this message. */
07088     if ( (dest_ptr->p_flags & (RECEIVING | SENDING)) == RECEIVING &&
07089          (dest_ptr->p_getfrom == ANY ||
07090           dest_ptr->p_getfrom == proc_number(caller_ptr))) {
07091           /* Destination is indeed waiting for this message. */
07092           CopyMess(proc_number(caller_ptr), caller_ptr, m_ptr, dest_ptr,
07093                    dest_ptr->p_messbuf);
07094           dest_ptr->p_flags &= ~RECEIVING;        /* deblock destination */
07095           if (dest_ptr->p_flags == 0) ready(dest_ptr);
07096     } else {
07097           /* Destination is not waiting.  Block and queue caller. */
07098           caller_ptr->p_messbuf = m_ptr;
07099           if (caller_ptr->p_flags == 0) unready(caller_ptr);
07100           caller_ptr->p_flags |= SENDING;
07101           caller_ptr->p_sendto= dest;
07102
07103           /* Process is now blocked.  Put in on the destination's queue. */
07104           if ( (next_ptr = dest_ptr->p_callerq) == NIL_PROC)
07105                   dest_ptr->p_callerq = caller_ptr;
07106           else {
07107                   while (next_ptr->p_sendlink != NIL_PROC)
07108                           next_ptr = next_ptr->p_sendlink;
07109                   next_ptr->p_sendlink = caller_ptr;
07110           }
07111           caller_ptr->p_sendlink = NIL_PROC;
07112     }
07113     return(OK);
07114   }

07116   /*===========================================================================*
07117    *                              mini_rec                                     *
07118    *===========================================================================*/
07119   PRIVATE int mini_rec(caller_ptr, src, m_ptr)
07120   register struct proc *caller_ptr;       /* process trying to get message */
07121   int src;                        /* which message source is wanted (or ANY) */
07122   message *m_ptr;                 /* pointer to message buffer */
07123   {
07124   /* A process or task wants to get a message.  If one is already queued,
07125    * acquire it and deblock the sender.  If no message from the desired source
07126    * is available, block the caller.  No need to check parameters for validity.
07127    * Users calls are always sendrec(), and mini_send() has checked already.
07128    * Calls from the tasks, MM, and FS are trusted.
07129    */
07130
07131     register struct proc *sender_ptr;
07132     register struct proc *previous_ptr;
07133
07134     /* Check to see if a message from desired source is already available. */
```



```
07135     if (!(caller_ptr->p_flags & SENDING)) {
07136           /* Check caller queue. */
07137       for (sender_ptr = caller_ptr->p_callerq; sender_ptr != NIL_PROC;
07138            previous_ptr = sender_ptr, sender_ptr = sender_ptr->p_sendlink) {
07139           if (src == ANY || src == proc_number(sender_ptr)) {
07140                   /* An acceptable message has been found. */
07141                   CopyMess(proc_number(sender_ptr), sender_ptr,
07142                            sender_ptr->p_messbuf, caller_ptr, m_ptr);
07143                   if (sender_ptr == caller_ptr->p_callerq)
07144                           caller_ptr->p_callerq = sender_ptr->p_sendlink;
07145                   else
07146                           previous_ptr->p_sendlink = sender_ptr->p_sendlink;
07147                   if ((sender_ptr->p_flags &= ~SENDING) == 0)
07148                           ready(sender_ptr);      /* deblock sender */
07149                   return(OK);
07150           }
07151       }
07152
07153       /* Check for blocked interrupt. */
07154       if (caller_ptr->p_int_blocked && isrxhardware(src)) {
07155           m_ptr->m_source = HARDWARE;
07156           m_ptr->m_type = HARD_INT;
07157           caller_ptr->p_int_blocked = FALSE;
07158           return(OK);
07159       }
07160     }
07161
07162     /* No suitable message is available.  Block the process trying to receive. */
07163     caller_ptr->p_getfrom = src;
07164     caller_ptr->p_messbuf = m_ptr;
07165     if (caller_ptr->p_flags == 0) unready(caller_ptr);
07166     caller_ptr->p_flags |= RECEIVING;
07167
07168     /* If MM has just blocked and there are kernel signals pending, now is the
07169      * time to tell MM about them, since it will be able to accept the message.
07170      */
07171     if (sig_procs > 0 && proc_number(caller_ptr) == MM_PROC_NR && src == ANY)
07172           inform();
07173     return(OK);
07174   }

07176   /*===========================================================================*
07177    *                              pick_proc                                    *
07178    *===========================================================================*/
07179   PRIVATE void pick_proc()
07180   {
07181   /* Decide who to run now.  A new process is selected by setting 'proc_ptr'.
07182    * When a fresh user (or idle) process is selected, record it in 'bill_ptr',
07183    * so the clock task can tell who to bill for system time.
07184    */
07185
07186     register struct proc *rp;     /* process to run */
07187
07188     if ( (rp = rdy_head[TASK_Q]) != NIL_PROC) {
07189           proc_ptr = rp;
07190           return;
07191     }
07192     if ( (rp = rdy_head[SERVER_Q]) != NIL_PROC) {
07193           proc_ptr = rp;
07194           return;
```

```
07195     }
07196     if ( (rp = rdy_head[USER_Q]) != NIL_PROC) {
07197           proc_ptr = rp;
07198           bill_ptr = rp;
07199           return;
07200     }
07201     /* No one is ready.  Run the idle task.  The idle task might be made an
07202      * always-ready user task to avoid this special case.
07203      */
07204     bill_ptr = proc_ptr = proc_addr(IDLE);
07205   }

07207   /*===========================================================================*
07208    *                                ready                                      *
07209    *===========================================================================*/
07210   PRIVATE void ready(rp)
07211   register struct proc *rp;       /* this process is now runnable */
07212   {
07213   /* Add 'rp' to the end of one of the queues of runnable processes.  Three
07214    * queues are maintained:
07215    *   TASK_Q   - (highest priority) for runnable tasks
07216    *   SERVER_Q - (middle priority) for MM and FS only
07217    *   USER_Q   - (lowest priority) for user processes
07218    */
07219
07220     if (istaskp(rp)) {
07221           if (rdy_head[TASK_Q] != NIL_PROC)
07222                   /* Add to tail of nonempty queue. */
07223                   rdy_tail[TASK_Q]->p_nextready = rp;
07224           else {
07225                   proc_ptr =              /* run fresh task next */
07226                   rdy_head[TASK_Q] = rp;  /* add to empty queue */
07227           }
07228           rdy_tail[TASK_Q] = rp;
07229           rp->p_nextready = NIL_PROC;     /* new entry has no successor */
07230           return;
07231     }
07232     if (!isuserp(rp)) {           /* others are similar */
07233           if (rdy_head[SERVER_Q] != NIL_PROC)
07234                   rdy_tail[SERVER_Q]->p_nextready = rp;
07235           else
07236                   rdy_head[SERVER_Q] = rp;
07237           rdy_tail[SERVER_Q] = rp;
07238           rp->p_nextready = NIL_PROC;
07239           return;
07240     }
07241     if (rdy_head[USER_Q] == NIL_PROC)
07242           rdy_tail[USER_Q] = rp;
07243     rp->p_nextready = rdy_head[USER_Q];
07244     rdy_head[USER_Q] = rp;
07245   /*
07246     if (rdy_head[USER_Q] != NIL_PROC)
07247           rdy_tail[USER_Q]->p_nextready = rp;
07248     else
07249           rdy_head[USER_Q] = rp;
07250     rdy_tail[USER_Q] = rp;
07251     rp->p_nextready = NIL_PROC;
07252   */
07253   }
```



```
07255   /*===========================================================================*
07256    *                               unready                                     *
07257    *===========================================================================*/
07258   PRIVATE void unready(rp)
07259   register struct proc *rp;       /* this process is no longer runnable */
07260   {
07261   /* A process has blocked. */
07262
07263     register struct proc *xp;
07264     register struct proc **qtail;  /* TASK_Q, SERVER_Q, or USER_Q rdy_tail */
07265
07266     if (istaskp(rp)) {
07267           /* task stack still ok? */
07268           if (*rp->p_stguard != STACK_GUARD)
07269                   panic("stack overrun by task", proc_number(rp));
07270
07271           if ( (xp = rdy_head[TASK_Q]) == NIL_PROC) return;
07272           if (xp == rp) {
07273                   /* Remove head of queue */
07274                   rdy_head[TASK_Q] = xp->p_nextready;
07275                   if (rp == proc_ptr) pick_proc();
07276                   return;
07277           }
07278           qtail = &rdy_tail[TASK_Q];
07279     }
07280     else if (!isuserp(rp)) {
07281           if ( (xp = rdy_head[SERVER_Q]) == NIL_PROC) return;
07282           if (xp == rp) {
07283                   rdy_head[SERVER_Q] = xp->p_nextready;
07284                   pick_proc();
07285                   return;
07286           }
07287           qtail = &rdy_tail[SERVER_Q];
07288     } else
07289     {
07290           if ( (xp = rdy_head[USER_Q]) == NIL_PROC) return;
07291           if (xp == rp) {
07292                   rdy_head[USER_Q] = xp->p_nextready;
07293                   pick_proc();
07294                   return;
07295           }
07296           qtail = &rdy_tail[USER_Q];
07297     }
07298
07299     /* Search body of queue.  A process can be made unready even if it is
07300      * not running by being sent a signal that kills it.
07301      */
07302     while (xp->p_nextready != rp)
07303           if ( (xp = xp->p_nextready) == NIL_PROC) return;
07304     xp->p_nextready = xp->p_nextready->p_nextready;
07305     if (*qtail == rp) *qtail = xp;
07306   }

07308   /*===========================================================================*
07309    *                                sched                                      *
07310    *===========================================================================*/
07311   PRIVATE void sched()
07312   {
07313   /* The current process has run too long.  If another low priority (user)
07314    * process is runnable, put the current process on the end of the user queue,
```

```
07315     * possibly promoting another user to head of the queue.
07316     */
07317
07318     if (rdy_head[USER_Q] == NIL_PROC) return;
07319
07320     /* One or more user processes queued. */
07321     rdy_tail[USER_Q]->p_nextready = rdy_head[USER_Q];
07322     rdy_tail[USER_Q] = rdy_head[USER_Q];
07323     rdy_head[USER_Q] = rdy_head[USER_Q]->p_nextready;
07324     rdy_tail[USER_Q]->p_nextready = NIL_PROC;
07325     pick_proc();
07326   }

07328   /*===========================================================================*
07329    *                              lock_mini_send                               *
07330    *===========================================================================*/
07331   PUBLIC int lock_mini_send(caller_ptr, dest, m_ptr)
07332   struct proc *caller_ptr;        /* who is trying to send a message? */
07333   int dest;                       /* to whom is message being sent? */
07334   message *m_ptr;                 /* pointer to message buffer */
07335   {
07336   /* Safe gateway to mini_send() for tasks. */
07337
07338     int result;
07339
07340     switching = TRUE;
07341     result = mini_send(caller_ptr, dest, m_ptr);
07342     switching = FALSE;
07343     return(result);
07344   }

07346   /*===========================================================================*
07347    *                              lock_pick_proc                               *
07348    *===========================================================================*/
07349   PUBLIC void lock_pick_proc()
07350   {
07351   /* Safe gateway to pick_proc() for tasks. */
07352
07353     switching = TRUE;
07354     pick_proc();
07355     switching = FALSE;
07356   }

07358   /*===========================================================================*
07359    *                              lock_ready                                   *
07360    *===========================================================================*/
07361   PUBLIC void lock_ready(rp)
07362   struct proc *rp;                /* this process is now runnable */
07363   {
07364   /* Safe gateway to ready() for tasks. */
07365
07366     switching = TRUE;
07367     ready(rp);
07368     switching = FALSE;
07369   }


07372   /*===========================================================================*
07373    *                              lock_unready                                 *
07374    *===========================================================================*/
```



```
07375   PUBLIC void lock_unready(rp)
07376   struct proc *rp;                /* this process is no longer runnable */
07377   {
07378   /* Safe gateway to unready() for tasks. */
07379
07380     switching = TRUE;
07381     unready(rp);
07382     switching = FALSE;
07383   }

07385   /*===========================================================================*
07386    *                              lock_sched                                   *
07387    *===========================================================================*/
07388   PUBLIC void lock_sched()
07389   {
07390   /* Safe gateway to sched() for tasks. */
07391
07392     switching = TRUE;
07393     sched();
07394     switching = FALSE;
07395   }

07397   /*===========================================================================*
07398    *                              unhold                                       *
07399    *===========================================================================*/
07400   PUBLIC void unhold()
07401   {
07402   /* Flush any held-up interrupts.  k_reenter must be 0.  held_head must not
07403    * be NIL_PROC.  Interrupts must be disabled.  They will be enabled but will
07404    * be disabled when this returns.
07405    */
07406
07407     register struct proc *rp;     /* current head of held queue */
07408
07409     if (switching) return;
07410     rp = held_head;
07411     do {
07412           if ( (held_head = rp->p_nextheld) == NIL_PROC) held_tail = NIL_PROC;
07413           rp->p_int_held = FALSE;
07414           unlock();               /* reduce latency; held queue may change! */
07415           interrupt(proc_number(rp));
07416           lock();                 /* protect the held queue again */
07417     }
07418     while ( (rp = held_head) != NIL_PROC);
07419   }
```

++
src/kernel/exception.c
++

```
07500   /* This file contains a simple exception handler.  Exceptions in user
07501    * processes are converted to signals.  Exceptions in the kernel, MM and
07502    * FS cause a panic.
07503    */
07504
07505   #include "kernel.h"
07506   #include <signal.h>
07507   #include "proc.h"
07508
07509   /*===========================================================================*
07510    *                              exception                                    *
07511    *===========================================================================*/
07512   PUBLIC void exception(vec_nr)
07513   unsigned vec_nr;
07514   {
07515   /* An exception or unexpected interrupt has occurred. */
07516
07517     struct ex_s {
07518           char *msg;
07519           int signum;
07520           int minprocessor;
07521     };
07522     static struct ex_s ex_data[] = {
07523           "Divide error", SIGFPE, 86,
07524           "Debug exception", SIGTRAP, 86,
07525           "Nonmaskable interrupt", SIGBUS, 86,
07526           "Breakpoint", SIGEMT, 86,
07527           "Overflow", SIGFPE, 86,
07528           "Bounds check", SIGFPE, 186,
07529           "Invalid opcode", SIGILL, 186,
07530           "Coprocessor not available", SIGFPE, 186,
07531           "Double fault", SIGBUS, 286,
07532           "Copressor segment overrun", SIGSEGV, 286,
07533           "Invalid TSS", SIGSEGV, 286,
07534           "Segment not present", SIGSEGV, 286,
07535           "Stack exception", SIGSEGV, 286,        /* STACK_FAULT already used */
07536           "General protection", SIGSEGV, 286,
07537           "Page fault", SIGSEGV, 386,             /* not close */
07538           NIL_PTR, SIGILL, 0,                     /* probably software trap */
07539           "Coprocessor error", SIGFPE, 386,
07540     };
07541     register struct ex_s *ep;
07542     struct proc *saved_proc;
07543
07544     saved_proc= proc_ptr; /* Save proc_ptr, because it may be changed by debug
07545                            * statements.
07546                            */
07547
07548     ep = &ex_data[vec_nr];
07549
07550     if (vec_nr == 2) {            /* spurious NMI on some machines */
07551           printf("got spurious NMI\n");
07552           return;
07553     }
07554
```



```
07555     if (k_reenter == 0 && isuserp(saved_proc)) {
07556           unlock();               /* this is protected like sys_call() */
07557           cause_sig(proc_number(saved_proc), ep->signum);
07558           return;
07559     }
07560
07561     /* This is not supposed to happen. */
07562     if (ep->msg == NIL_PTR || processor < ep->minprocessor)
07563           printf("\nIntel-reserved exception %d\n", vec_nr);
07564     else
07565           printf("\n%s\n", ep->msg);
07566     printf("process number %d, pc = 0x%04x:0x%08x\n",
07567           proc_number(saved_proc),
07568           (unsigned) saved_proc->p_reg.cs,
07569           (unsigned) saved_proc->p_reg.pc);
07570     panic("exception in system code", NO_NUM);
07571   }
```

++
src/kernel/i8259.c
++

```
07600   /* This file contains routines for initializing the 8259 interrupt controller:
07601    *      get_irq_handler: address of handler for a given interrupt
07602    *      put_irq_handler: register an interrupt handler
07603    *      intr_init:      initialize the interrupt controller(s)
07604    */
07605
07606   #include "kernel.h"
07607
07608   #define ICW1_AT         0x11    /* edge triggered, cascade, need ICW4 */
07609   #define ICW1_PC         0x13    /* edge triggered, no cascade, need ICW4 */
07610   #define ICW1_PS         0x19    /* level triggered, cascade, need ICW4 */
07611   #define ICW4_AT         0x01    /* not SFNM, not buffered, normal EOI, 8086 */
07612   #define ICW4_PC         0x09    /* not SFNM, buffered, normal EOI, 8086 */
07613
07614   FORWARD _PROTOTYPE( int spurious_irq, (int irq) );
07615
07616   #define set_vec(nr, addr)       ((void)0) /* kluge for protected mode */
07617
07618   /*===========================================================================*
07619    *                              intr_init                                    *
07620    *===========================================================================*/
07621   PUBLIC void intr_init(mine)
07622   int mine;
07623   {
07624   /* Initialize the 8259s, finishing with all interrupts disabled.  This is
07625    * only done in protected mode, in real mode we don't touch the 8259s, but
07626    * use the BIOS locations instead.  The flag "mine" is set if the 8259s are
07627    * to be programmed for Minix, or to be reset to what the BIOS expects.
07628    */
07629
07630     int i;
07631
07632     lock();
07633     /* The AT and newer PS/2 have two interrupt controllers, one master,
07634      * one slaved at IRQ 2.  (We don't have to deal with the PC that
```

```
07635       * has just one controller, because it must run in real mode.)
07636       */
07637      out_byte(INT_CTL, ps_mca ? ICW1_PS : ICW1_AT);
07638      out_byte(INT_CTLMASK, mine ? IRQ0_VECTOR : BIOS_IRQ0_VEC);
07639                                              /* ICW2 for master */
07640      out_byte(INT_CTLMASK, (1 << CASCADE_IRQ));    /* ICW3 tells slaves */
07641      out_byte(INT_CTLMASK, ICW4_AT);
07642      out_byte(INT_CTLMASK, ~(1 << CASCADE_IRQ));   /* IRQ 0-7 mask */
07643      out_byte(INT2_CTL, ps_mca ? ICW1_PS : ICW1_AT);
07644      out_byte(INT2_CTLMASK, mine ? IRQ8_VECTOR : BIOS_IRQ8_VEC);
07645                                              /* ICW2 for slave */
07646      out_byte(INT2_CTLMASK, CASCADE_IRQ);          /* ICW3 is slave nr */
07647      out_byte(INT2_CTLMASK, ICW4_AT);
07648      out_byte(INT2_CTLMASK, ~0);                   /* IRQ 8-15 mask */
07649
07650      /* Initialize the table of interrupt handlers. */
07651      for (i = 0; i < NR_IRQ_VECTORS; i++) irq_table[i] = spurious_irq;
07652  }

07654  /*===========================================================================*
07655   *                                spurious_irq                               *
07656   *===========================================================================*/
07657  PRIVATE int spurious_irq(irq)
07658  int irq;
07659  {
07660  /* Default interrupt handler.  It complains a lot. */
07661
07662    if (irq < 0 || irq >= NR_IRQ_VECTORS)
07663          panic("invalid call to spurious_irq", irq);
07664
07665    printf("spurious irq %d\n", irq);
07666
07667    return 1;     /* Reenable interrupt */
07668  }

07670  /*===========================================================================*
07671   *                              put_irq_handler                              *
07672   *===========================================================================*/
07673  PUBLIC void put_irq_handler(irq, handler)
07674  int irq;
07675  irq_handler_t handler;
07676  {
07677  /* Register an interrupt handler. */
07678
07679    if (irq < 0 || irq >= NR_IRQ_VECTORS)
07680          panic("invalid call to put_irq_handler", irq);
07681
07682    if (irq_table[irq] == handler)
07683          return;         /* extra initialization */
07684
07685    if (irq_table[irq] != spurious_irq)
07686          panic("attempt to register second irq handler for irq", irq);
07687
07688    disable_irq(irq);
07689    if (!protected_mode) set_vec(BIOS_VECTOR(irq), irq_vec[irq]);
07690    irq_table[irq]= handler;
07691    irq_use |= 1 << irq;
07692  }
```



```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/kernel/protect.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

07700  /* This file contains code for initialization of protected mode, to initialize
07701   * code and data segment descriptors, and to initialize global descriptors
07702   * for local descriptors in the process table.
07703   */
07704
07705  #include "kernel.h"
07706  #include "proc.h"
07707  #include "protect.h"
07708
07709  #define INT_GATE_TYPE   (INT_286_GATE | DESC_386_BIT)
07710  #define TSS_TYPE        (AVL_286_TSS  | DESC_386_BIT)
07711
07712  struct desctableptr_s {
07713    char limit[sizeof(u16_t)];
07714    char base[sizeof(u32_t)];             /* really u24_t + pad for 286 */
07715  };
07716
07717  struct gatedesc_s {
07718    u16_t offset_low;
07719    u16_t selector;
07720    u8_t pad;                     /* |000|XXXXX| ig & trpg, |XXXXXXXX| task g */
07721    u8_t p_dpl_type;              /* |P|DL|0|TYPE| */
07722    u16_t offset_high;
07723  };
07724
07725  struct tss_s {
07726    reg_t backlink;
07727    reg_t sp0;                    /* stack pointer to use during interrupt */
07728    reg_t ss0;                    /*   "   segment  "  "    "        "     */
07729    reg_t sp1;
07730    reg_t ss1;
07731    reg_t sp2;
07732    reg_t ss2;
07733    reg_t cr3;
07734    reg_t ip;
07735    reg_t flags;
07736    reg_t ax;
07737    reg_t cx;
07738    reg_t dx;
07739    reg_t bx;
07740    reg_t sp;
07741    reg_t bp;
07742    reg_t si;
07743    reg_t di;
07744    reg_t es;
07745    reg_t cs;
07746    reg_t ss;
07747    reg_t ds;
07748    reg_t fs;
07749    reg_t gs;
07750    reg_t ldt;
07751    u16_t trap;
07752    u16_t iobase;
07753  };
07754
```

```
07755   PUBLIC struct segdesc_s gdt[GDT_SIZE];
07756   PRIVATE struct gatedesc_s idt[IDT_SIZE];        /* zero-init so none present */
07757   PUBLIC struct tss_s tss;        /* zero init */
07758
07759   FORWARD _PROTOTYPE( void int_gate, (unsigned vec_nr, phys_bytes base,
07760                   unsigned dpl_type) );
07761   FORWARD _PROTOTYPE( void sdesc, (struct segdesc_s *segdp, phys_bytes base,
07762                   phys_bytes size) );
07763
07764   /*==========================================================================*
07765    *                              prot_init                                   *
07766    *==========================================================================*/
07767   PUBLIC void prot_init()
07768   {
07769   /* Set up tables for protected mode.
07770    * All GDT slots are allocated at compile time.
07771    */
07772
07773     phys_bytes code_bytes;
07774     phys_bytes data_bytes;
07775     struct gate_table_s *gtp;
07776     struct desctableptr_s *dtp;
07777     unsigned ldt_selector;
07778     register struct proc *rp;
07779
07780     static struct gate_table_s {
07781           _PROTOTYPE( void (*gate), (void) );
07782           unsigned char vec_nr;
07783           unsigned char privilege;
07784     }
07785     gate_table[] = {
07786           divide_error, DIVIDE_VECTOR, INTR_PRIVILEGE,
07787           single_step_exception, DEBUG_VECTOR, INTR_PRIVILEGE,
07788           nmi, NMI_VECTOR, INTR_PRIVILEGE,
07789           breakpoint_exception, BREAKPOINT_VECTOR, USER_PRIVILEGE,
07790           overflow, OVERFLOW_VECTOR, USER_PRIVILEGE,
07791           bounds_check, BOUNDS_VECTOR, INTR_PRIVILEGE,
07792           inval_opcode, INVAL_OP_VECTOR, INTR_PRIVILEGE,
07793           copr_not_available, COPROC_NOT_VECTOR, INTR_PRIVILEGE,
07794           double_fault, DOUBLE_FAULT_VECTOR, INTR_PRIVILEGE,
07795           copr_seg_overrun, COPROC_SEG_VECTOR, INTR_PRIVILEGE,
07796           inval_tss, INVAL_TSS_VECTOR, INTR_PRIVILEGE,
07797           segment_not_present, SEG_NOT_VECTOR, INTR_PRIVILEGE,
07798           stack_exception, STACK_FAULT_VECTOR, INTR_PRIVILEGE,
07799           general_protection, PROTECTION_VECTOR, INTR_PRIVILEGE,
07800           page_fault, PAGE_FAULT_VECTOR, INTR_PRIVILEGE,
07801           copr_error, COPROC_ERR_VECTOR, INTR_PRIVILEGE,
07802           { hwint00, VECTOR( 0), INTR_PRIVILEGE },
07803           { hwint01, VECTOR( 1), INTR_PRIVILEGE },
07804           { hwint02, VECTOR( 2), INTR_PRIVILEGE },
07805           { hwint03, VECTOR( 3), INTR_PRIVILEGE },
07806           { hwint04, VECTOR( 4), INTR_PRIVILEGE },
07807           { hwint05, VECTOR( 5), INTR_PRIVILEGE },
07808           { hwint06, VECTOR( 6), INTR_PRIVILEGE },
07809           { hwint07, VECTOR( 7), INTR_PRIVILEGE },
07810           { hwint08, VECTOR( 8), INTR_PRIVILEGE },
07811           { hwint09, VECTOR( 9), INTR_PRIVILEGE },
07812           { hwint10, VECTOR(10), INTR_PRIVILEGE },
07813           { hwint11, VECTOR(11), INTR_PRIVILEGE },
07814           { hwint12, VECTOR(12), INTR_PRIVILEGE },
```



```
07815           { hwint13, VECTOR(13), INTR_PRIVILEGE },
07816           { hwint14, VECTOR(14), INTR_PRIVILEGE },
07817           { hwint15, VECTOR(15), INTR_PRIVILEGE },
07818     };
07819
07820     /* This is called early and can't use tables set up by main(). */
07821     data_bytes = (phys_bytes) sizes[1] << CLICK_SHIFT;
07822     if (sizes[0] == 0)
07823           code_bytes = data_bytes;        /* common I&D */
07824     else
07825           code_bytes = (phys_bytes) sizes[0] << CLICK_SHIFT;
07826
07827     /* Build gdt and idt pointers in GDT where the BIOS expects them. */
07828     dtp= (struct desctableptr_s *) &gdt[GDT_INDEX];
07829     * (u16_t *) dtp->limit = (sizeof gdt) - 1;
07830     * (u32_t *) dtp->base = vir2phys(gdt);
07831
07832     dtp= (struct desctableptr_s *) &gdt[IDT_INDEX];
07833     * (u16_t *) dtp->limit = (sizeof idt) - 1;
07834     * (u32_t *) dtp->base = vir2phys(idt);
07835
07836     /* Build segment descriptors for tasks and interrupt handlers. */
07837     init_codeseg(&gdt[CS_INDEX], code_base, code_bytes, INTR_PRIVILEGE);
07838     init_dataseg(&gdt[DS_INDEX], data_base, data_bytes, INTR_PRIVILEGE);
07839     init_dataseg(&gdt[ES_INDEX], 0L, 0L, TASK_PRIVILEGE);
07840
07841     /* Build scratch descriptors for functions in klib88. */
07842     init_dataseg(&gdt[DS_286_INDEX], (phys_bytes) 0,
07843                   (phys_bytes) MAX_286_SEG_SIZE, TASK_PRIVILEGE);
07844     init_dataseg(&gdt[ES_286_INDEX], (phys_bytes) 0,
07845                   (phys_bytes) MAX_286_SEG_SIZE, TASK_PRIVILEGE);
07846
07847     /* Build local descriptors in GDT for LDT's in process table.
07848      * The LDT's are allocated at compile time in the process table, and
07849      * initialized whenever a process' map is initialized or changed.
07850      */
07851     for (rp = BEG_PROC_ADDR, ldt_selector = FIRST_LDT_INDEX * DESC_SIZE;
07852          rp < END_PROC_ADDR; ++rp, ldt_selector += DESC_SIZE) {
07853           init_dataseg(&gdt[ldt_selector / DESC_SIZE], vir2phys(rp->p_ldt),
07854                         (phys_bytes) sizeof rp->p_ldt, INTR_PRIVILEGE);
07855           gdt[ldt_selector / DESC_SIZE].access = PRESENT | LDT;
07856           rp->p_ldt_sel = ldt_selector;
07857     }
07858
07859     /* Build main TSS.
07860      * This is used only to record the stack pointer to be used after an
07861      * interrupt.
07862      * The pointer is set up so that an interrupt automatically saves the
07863      * current process's registers ip:cs:f:sp:ss in the correct slots in the
07864      * process table.
07865      */
07866     tss.ss0 = DS_SELECTOR;
07867     init_dataseg(&gdt[TSS_INDEX], vir2phys(&tss), (phys_bytes) sizeof tss,
07868                                                               INTR_PRIVILEGE);
07869     gdt[TSS_INDEX].access = PRESENT | (INTR_PRIVILEGE << DPL_SHIFT) | TSS_TYPE;
07870     tss.iobase = sizeof tss;      /* empty i/o permissions map */
07871
07872     /* Build descriptors for interrupt gates in IDT. */
07873     for (gtp = &gate_table[0];
07874          gtp < &gate_table[sizeof gate_table / sizeof gate_table[0]]; ++gtp) {
```

```
07875           int_gate(gtp->vec_nr, (phys_bytes) (vir_bytes) gtp->gate,
07876                    PRESENT | INT_GATE_TYPE | (gtp->privilege << DPL_SHIFT));
07877     }
07878     int_gate(SYS_VECTOR, (phys_bytes) (vir_bytes) p_s_call,
07879              PRESENT | (USER_PRIVILEGE << DPL_SHIFT) | INT_GATE_TYPE);
07880     int_gate(LEVEL0_VECTOR, (phys_bytes) (vir_bytes) level0_call,
07881              PRESENT | (TASK_PRIVILEGE << DPL_SHIFT) | INT_GATE_TYPE);
07882     int_gate(SYS386_VECTOR, (phys_bytes) (vir_bytes) s_call,
07883              PRESENT | (USER_PRIVILEGE << DPL_SHIFT) | INT_GATE_TYPE);
07884   }

07886   /*===========================================================================*
07887    *                              init_codeseg                                 *
07888    *===========================================================================*/
07889   PUBLIC void init_codeseg(segdp, base, size, privilege)
07890   register struct segdesc_s *segdp;
07891   phys_bytes base;
07892   phys_bytes size;
07893   int privilege;
07894   {
07895   /* Build descriptor for a code segment. */
07896
07897     sdesc(segdp, base, size);
07898     segdp->access = (privilege << DPL_SHIFT)
07899                     | (PRESENT | SEGMENT | EXECUTABLE | READABLE);
07900                     /* CONFORMING = 0, ACCESSED = 0 */
07901   }

07903   /*===========================================================================*
07904    *                              init_dataseg                                 *
07905    *===========================================================================*/
07906   PUBLIC void init_dataseg(segdp, base, size, privilege)
07907   register struct segdesc_s *segdp;
07908   phys_bytes base;
07909   phys_bytes size;
07910   int privilege;
07911   {
07912   /* Build descriptor for a data segment. */
07913
07914     sdesc(segdp, base, size);
07915     segdp->access = (privilege << DPL_SHIFT) | (PRESENT | SEGMENT | WRITEABLE);
07916                     /* EXECUTABLE = 0, EXPAND_DOWN = 0, ACCESSED = 0 */
07917   }

07919   /*===========================================================================*
07920    *                              sdesc                                        *
07921    *===========================================================================*/
07922   PRIVATE void sdesc(segdp, base, size)
07923   register struct segdesc_s *segdp;
07924   phys_bytes base;
07925   phys_bytes size;
07926   {
07927   /* Fill in the size fields (base, limit and granularity) of a descriptor. */
07928
07929     segdp->base_low = base;
07930     segdp->base_middle = base >> BASE_MIDDLE_SHIFT;
07931     segdp->base_high = base >> BASE_HIGH_SHIFT;
07932     --size;                       /* convert to a limit, 0 size means 4G */
07933     if (size > BYTE_GRAN_MAX) {
07934           segdp->limit_low = size >> PAGE_GRAN_SHIFT;
```



```
07935           segdp->granularity = GRANULAR | (size >>
07936                                       (PAGE_GRAN_SHIFT + GRANULARITY_SHIFT));
07937     } else {
07938           segdp->limit_low = size;
07939           segdp->granularity = size >> GRANULARITY_SHIFT;
07940     }
07941     segdp->granularity |= DEFAULT;        /* means BIG for data seg */
07942   }

07944   /*===========================================================================*
07945    *                              seg2phys                                     *
07946    *===========================================================================*/
07947   PUBLIC phys_bytes seg2phys(seg)
07948   U16_t seg;
07949   {
07950   /* Return the base address of a segment, with seg being either a 8086 segment
07951    * register, or a 286/386 segment selector.
07952    */
07953     phys_bytes base;
07954     struct segdesc_s *segdp;
07955
07956     if (!protected_mode) {
07957           base = hclick_to_physb(seg);
07958     } else {
07959           segdp = &gdt[seg >> 3];
07960           base = segdp->base_low | ((u32_t) segdp->base_middle << 16);
07961           base |= ((u32_t) segdp->base_high << 24);
07962     }
07963     return base;
07964   }

07966   /*===========================================================================*
07967    *                              int_gate                                     *
07968    *===========================================================================*/
07969   PRIVATE void int_gate(vec_nr, base, dpl_type)
07970   unsigned vec_nr;
07971   phys_bytes base;
07972   unsigned dpl_type;
07973   {
07974   /* Build descriptor for an interrupt gate. */
07975
07976     register struct gatedesc_s *idp;
07977
07978     idp = &idt[vec_nr];
07979     idp->offset_low = base;
07980     idp->selector = CS_SELECTOR;
07981     idp->p_dpl_type = dpl_type;
07982     idp->offset_high = base >> OFFSET_HIGH_SHIFT;
07983   }

07985   /*===========================================================================*
07986    *                              enable_iop                                   *
07987    *===========================================================================*/
07988   PUBLIC void enable_iop(pp)
07989   struct proc *pp;
07990   {
07991   /* Allow a user process to use I/O instructions.  Change the I/O Permission
07992    * Level bits in the psw. These specify least-privileged Current Permission
07993    * Level allowed to execute I/O instructions. Users and servers have CPL 3.
07994    * You can't have less privilege than that. Kernel has CPL 0, tasks CPL 1.
```

```
07995     */
07996      pp->p_reg.psw |= 0x3000;
07997    }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                              src/kernel/klib.s
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

08000    #
08001    ! Chooses between the 8086 and 386 versions of the low level kernel code.
08002
08003    #include <minix/config.h>
08004    #if _WORD_SIZE == 2
08005    #include "klib88.s"
08006    #else
08007    #include "klib386.s"
08008    #endif
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                              src/kernel/klib386.s
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

08100    #
08101    ! sections
08102
08103    .sect .text; .sect .rom; .sect .data; .sect .bss
08104
08105    #include <minix/config.h>
08106    #include <minix/const.h>
08107    #include "const.h"
08108    #include "sconst.h"
08109    #include "protect.h"
08110
08111    ! This file contains a number of assembly code utility routines needed by the
08112    ! kernel.  They are:
08113
08114    .define _monitor        ! exit Minix and return to the monitor
08115    .define _check_mem      ! check a block of memory, return the valid size
08116    .define _cp_mess        ! copies messages from source to destination
08117    .define _exit           ! dummy for library routines
08118    .define __exit          ! dummy for library routines
08119    .define ___exit         ! dummy for library routines
08120    .define ___main         ! dummy for GCC
08121    .define _in_byte        ! read a byte from a port and return it
08122    .define _in_word        ! read a word from a port and return it
08123    .define _out_byte       ! write a byte to a port
08124    .define _out_word       ! write a word to a port
08125    .define _port_read      ! transfer data from (disk controller) port to memory
08126    .define _port_read_byte ! likewise byte by byte
08127    .define _port_write     ! transfer data from memory to (disk controller) port
08128    .define _port_write_byte ! likewise byte by byte
08129    .define _lock           ! disable interrupts
08130    .define _unlock         ! enable interrupts
08131    .define _enable_irq     ! enable an irq at the 8259 controller
08132    .define _disable_irq    ! disable an irq
08133    .define _phys_copy      ! copy data from anywhere to anywhere in memory
08134    .define _mem_rdw        ! copy one word from [segment:offset]
```



```
08135    .define _reset          ! reset the system
08136    .define _mem_vid_copy   ! copy data to video ram
08137    .define _vid_vid_copy   ! move data in video ram
08138    .define _level0         ! call a function at level 0
08139
08140    ! The routines only guarantee to preserve the registers the C compiler
08141    ! expects to be preserved (ebx, esi, edi, ebp, esp, segment registers, and
08142    ! direction bit in the flags).
08143
08144    ! imported variables
08145
08146    .sect .bss
08147    .extern _mon_return, _mon_sp
08148    .extern _irq_use
08149    .extern _blank_color
08150    .extern _ext_memsize
08151    .extern _gdt
08152    .extern _low_memsize
08153    .extern _sizes
08154    .extern _vid_seg
08155    .extern _vid_size
08156    .extern _vid_mask
08157    .extern _level0_func
08158
08159    .sect .text
08160    !*===========================================================================*
08161    !*                              monitor                                      *
08162    !*===========================================================================*
08163    ! PUBLIC void monitor();
08164    ! Return to the monitor.
08165
08166    _monitor:
08167            mov     eax, (_reboot_code)     ! address of new parameters
08168            mov     esp, (_mon_sp)          ! restore monitor stack pointer
08169        o16 mov     dx, SS_SELECTOR         ! monitor data segment
08170            mov     ds, dx
08171            mov     es, dx
08172            mov     fs, dx
08173            mov     gs, dx
08174            mov     ss, dx
08175            pop     edi
08176            pop     esi
08177            pop     ebp
08178        o16 retf                            ! return to the monitor
08179
08180
08181    !*===========================================================================*
08182    !*                              check_mem                                    *
08183    !*===========================================================================*
08184    ! PUBLIC phys_bytes check_mem(phys_bytes base, phys_bytes size);
08185    ! Check a block of memory, return the amount valid.
08186    ! Only every 16th byte is checked.
08187    ! An initial size of 0 means everything.
08188    ! This really should do some alias checks.
08189
08190    CM_DENSITY      =       16
08191    CM_LOG_DENSITY  =       4
08192    TEST1PATTERN    =       0x55            ! memory test pattern 1
08193    TEST2PATTERN    =       0xAA            ! memory test pattern 2
08194
```

```
08195 CHKM_ARGS       =       4 + 4 + 4       ! 4 + 4
08196 !                       ds ebx eip      base size
08197
08198 _check_mem:
08199           push    ebx
08200           push    ds
08201       o16 mov     ax, FLAT_DS_SELECTOR
08202           mov     ds, ax
08203           mov     eax, CHKM_ARGS(esp)
08204           mov     ebx, eax
08205           mov     ecx, CHKM_ARGS+4(esp)
08206           shr     ecx, CM_LOG_DENSITY
08207 cm_loop:
08208           movb    dl, TEST1PATTERN
08209           xchgb   dl, (eax)               ! write test pattern, remember original
08210           xchgb   dl, (eax)               ! restore original, read test pattern
08211           cmpb    dl, TEST1PATTERN        ! must agree if good real memory
08212           jnz     cm_exit                 ! if different, memory is unusable
08213           movb    dl, TEST2PATTERN
08214           xchgb   dl, (eax)
08215           xchgb   dl, (eax)
08216           add     eax, CM_DENSITY
08217           cmpb    dl, TEST2PATTERN
08218           loopz   cm_loop
08219 cm_exit:
08220           sub     eax, ebx
08221           pop     ds
08222           pop     ebx
08223           ret
08224
08225
08226 !*===========================================================================*
08227 !*                              cp_mess                                      *
08228 !*===========================================================================*
08229 ! PUBLIC void cp_mess(int src, phys_clicks src_clicks, vir_bytes src_offset,
08230 !                    phys_clicks dst_clicks, vir_bytes dst_offset);
08231 ! This routine makes a fast copy of a message from anywhere in the address
08232 ! space to anywhere else.  It also copies the source address provided as a
08233 ! parameter to the call into the first word of the destination message.
08234 !
08235 ! Note that the message size, "Msize" is in DWORDS (not bytes) and must be set
08236 ! correctly.  Changing the definition of message in the type file and not
08237 ! changing it here will lead to total disaster.
08238
08239 CM_ARGS =       4 + 4 + 4 + 4 + 4       ! 4 + 4 + 4 + 4 + 4 + 4
08240 !               es  ds edi esi eip      proc scl sof dcl dof
08241
08242           .align  16
08243 _cp_mess:
08244           cld
08245           push    esi
08246           push    edi
08247           push    ds
08248           push    es
08249
08250           mov     eax, FLAT_DS_SELECTOR
08251           mov     ds, ax
08252           mov     es, ax
08253
08254           mov     esi, CM_ARGS+4(esp)             ! src clicks
```



```
08255           shl     esi, CLICK_SHIFT
08256           add     esi, CM_ARGS+4+4(esp)           ! src offset
08257           mov     edi, CM_ARGS+4+4+4(esp)         ! dst clicks
08258           shl     edi, CLICK_SHIFT
08259           add     edi, CM_ARGS+4+4+4+4(esp)       ! dst offset
08260
08261           mov     eax, CM_ARGS(esp)       ! process number of sender
08262           stos                            ! copy number of sender to dest message
08263           add     esi, 4                  ! do not copy first word
08264           mov     ecx, Msize - 1          ! remember, first word does not count
08265           rep
08266           movs                            ! copy the message
08267
08268           pop     es
08269           pop     ds
08270           pop     edi
08271           pop     esi
08272           ret                             ! that is all folks!
08273
08274
08275 !*===========================================================================*
08276 !*                              exit                                         *
08277 !*===========================================================================*
08278 ! PUBLIC void exit();
08279 ! Some library routines use exit, so provide a dummy version.
08280 ! Actual calls to exit cannot occur in the kernel.
08281 ! GNU CC likes to call ___main from main() for nonobvious reasons.
08282
08283 _exit:
08284 __exit:
08285 ___exit:
08286           sti
08287           jmp     ___exit
08288
08289 ___main:
08290           ret
08291
08292
08293 !*===========================================================================*
08294 !*                              in_byte                                      *
08295 !*===========================================================================*
08296 ! PUBLIC unsigned in_byte(port_t port);
08297 ! Read an (unsigned) byte from the i/o port  port  and return it.
08298
08299           .align  16
08300 _in_byte:
08301           mov     edx, 4(esp)             ! port
08302           sub     eax, eax
08303           inb     dx                      ! read 1 byte
08304           ret
08305
08306
08307 !*===========================================================================*
08308 !*                              in_word                                      *
08309 !*===========================================================================*
08310 ! PUBLIC unsigned in_word(port_t port);
08311 ! Read an (unsigned) word from the i/o port  port  and return it.
08312
08313           .align  16
08314 _in_word:
```

```
08315           mov     edx, 4(esp)             ! port
08316           sub     eax, eax
08317       o16 in      dx                      ! read 1 word
08318           ret
08319
08320
08321   !*===========================================================================*
08322   !*                              out_byte                                     *
08323   !*===========================================================================*
08324   ! PUBLIC void out_byte(port_t port, u8_t value);
08325   ! Write  value  (cast to a byte)  to the I/O port  port.
08326
08327           .align  16
08328   _out_byte:
08329           mov     edx, 4(esp)             ! port
08330           movb    al, 4+4(esp)            ! value
08331           outb    dx                      ! output 1 byte
08332           ret
08333
08334
08335   !*===========================================================================*
08336   !*                              out_word                                     *
08337   !*===========================================================================*
08338   ! PUBLIC void out_word(Port_t port, U16_t value);
08339   ! Write  value  (cast to a word)  to the I/O port  port.
08340
08341           .align  16
08342   _out_word:
08343           mov     edx, 4(esp)             ! port
08344           mov     eax, 4+4(esp)           ! value
08345       o16 out     dx                      ! output 1 word
08346           ret
08347
08348
08349   !*===========================================================================*
08350   !*                              port_read                                    *
08351   !*===========================================================================*
08352   ! PUBLIC void port_read(port_t port, phys_bytes destination, unsigned bytcount);
08353   ! Transfer data from (hard disk controller) port to memory.
08354
08355   PR_ARGS =       4 + 4 + 4               ! 4 + 4 + 4
08356   !               es edi eip              port dst len
08357
08358           .align  16
08359   _port_read:
08360           cld
08361           push    edi
08362           push    es
08363           mov     ecx, FLAT_DS_SELECTOR
08364           mov     es, cx
08365           mov     edx, PR_ARGS(esp)       ! port to read from
08366           mov     edi, PR_ARGS+4(esp)     ! destination addr
08367           mov     ecx, PR_ARGS+4+4(esp)   ! byte count
08368           shr     ecx, 1                  ! word count
08369           rep                             ! (hardware cannot handle dwords)
08370       o16 ins                             ! read everything
08371           pop     es
08372           pop     edi
08373           ret
08374
```



```
08375
08376   !*===========================================================================*
08377   !*                              port_read_byte                               *
08378   !*===========================================================================*
08379   ! PUBLIC void port_read_byte(port_t port, phys_bytes destination,
08380   !                                               unsigned bytcount);
08381   ! Transfer data from port to memory.
08382
08383   PR_ARGS_B =     4 + 4 + 4               ! 4 + 4 + 4
08384   !               es edi eip              port dst len
08385
08386   _port_read_byte:
08387           cld
08388           push    edi
08389           push    es
08390           mov     ecx, FLAT_DS_SELECTOR
08391           mov     es, cx
08392           mov     edx, PR_ARGS_B(esp)
08393           mov     edi, PR_ARGS_B+4(esp)
08394           mov     ecx, PR_ARGS_B+4+4(esp)
08395           rep
08396           insb
08397           pop     es
08398           pop     edi
08399           ret
08400
08401
08402   !*===========================================================================*
08403   !*                              port_write                                   *
08404   !*===========================================================================*
08405   ! PUBLIC void port_write(port_t port, phys_bytes source, unsigned bytcount);
08406   ! Transfer data from memory to (hard disk controller) port.
08407
08408   PW_ARGS =       4 + 4 + 4               ! 4 + 4 + 4
08409   !               es edi eip              port src len
08410
08411           .align  16
08412   _port_write:
08413           cld
08414           push    esi
08415           push    ds
08416           mov     ecx, FLAT_DS_SELECTOR
08417           mov     ds, cx
08418           mov     edx, PW_ARGS(esp)       ! port to write to
08419           mov     esi, PW_ARGS+4(esp)     ! source addr
08420           mov     ecx, PW_ARGS+4+4(esp)   ! byte count
08421           shr     ecx, 1                  ! word count
08422           rep                             ! (hardware cannot handle dwords)
08423       o16 outs                            ! write everything
08424           pop     ds
08425           pop     esi
08426           ret
08427
08428
08429   !*===========================================================================*
08430   !*                              port_write_byte                              *
08431   !*===========================================================================*
08432   ! PUBLIC void port_write_byte(port_t port, phys_bytes source,
08433   !                                               unsigned bytcount);
08434   ! Transfer data from memory to port.
```

```
08435
08436   PW_ARGS_B =     4 + 4 + 4               ! 4 + 4 + 4
08437   !               es edi eip              port src len
08438
08439   _port_write_byte:
08440           cld
08441           push    esi
08442           push    ds
08443           mov     ecx, FLAT_DS_SELECTOR
08444           mov     ds, cx
08445           mov     edx, PW_ARGS_B(esp)
08446           mov     esi, PW_ARGS_B+4(esp)
08447           mov     ecx, PW_ARGS_B+4+4(esp)
08448           rep
08449           outsb
08450           pop     ds
08451           pop     esi
08452           ret
08453
08454
08455   !*===========================================================================*
08456   !*                              lock                                         *
08457   !*===========================================================================*
08458   ! PUBLIC void lock();
08459   ! Disable CPU interrupts.
08460
08461           .align  16
08462   _lock:
08463           cli                             ! disable interrupts
08464           ret
08465
08466
08467   !*===========================================================================*
08468   !*                              unlock                                       *
08469   !*===========================================================================*
08470   ! PUBLIC void unlock();
08471   ! Enable CPU interrupts.
08472
08473           .align  16
08474   _unlock:
08475           sti
08476           ret
08477
08478
08479   !*===========================================================================*
08480   !*                              enable_irq                                   *
08481   !*===========================================================================*/
08482   ! PUBLIC void enable_irq(unsigned irq)
08483   ! Enable an interrupt request line by clearing an 8259 bit.
08484   ! Equivalent code for irq < 8:
08485   !       out_byte(INT_CTLMASK, in_byte(INT_CTLMASK) & ~(1 << irq));
08486
08487           .align  16
08488   _enable_irq:
08489           mov     ecx, 4(esp)             ! irq
08490           pushf
08491           cli
08492           movb    ah, ~1
08493           rolb    ah, cl                  ! ah = ~(1 << (irq % 8))
08494           cmpb    cl, 8
```



```
08495           jae     enable_8                ! enable irq >= 8 at the slave 8259
08496   enable_0:
08497           inb     INT_CTLMASK
08498           andb    al, ah
08499           outb    INT_CTLMASK             ! clear bit at master 8259
08500           popf
08501           ret
08502           .align  4
08503   enable_8:
08504           inb     INT2_CTLMASK
08505           andb    al, ah
08506           outb    INT2_CTLMASK            ! clear bit at slave 8259
08507           popf
08508           ret
08509
08510
08511   !*===========================================================================*
08512   !*                              disable_irq                                  *
08513   !*===========================================================================*/
08514   ! PUBLIC int disable_irq(unsigned irq)
08515   ! Disable an interrupt request line by setting an 8259 bit.
08516   ! Equivalent code for irq < 8:
08517   !       out_byte(INT_CTLMASK, in_byte(INT_CTLMASK) | (1 << irq));
08518   ! Returns true iff the interrupt was not already disabled.
08519
08520           .align  16
08521   _disable_irq:
08522           mov     ecx, 4(esp)             ! irq
08523           pushf
08524           cli
08525           movb    ah, 1
08526           rolb    ah, cl                  ! ah = (1 << (irq % 8))
08527           cmpb    cl, 8
08528           jae     disable_8               ! disable irq >= 8 at the slave 8259
08529   disable_0:
08530           inb     INT_CTLMASK
08531           testb   al, ah
08532           jnz     dis_already             ! already disabled?
08533           orb     al, ah
08534           outb    INT_CTLMASK             ! set bit at master 8259
08535           popf
08536           mov     eax, 1                  ! disabled by this function
08537           ret
08538   disable_8:
08539           inb     INT2_CTLMASK
08540           testb   al, ah
08541           jnz     dis_already             ! already disabled?
08542           orb     al, ah
08543           outb    INT2_CTLMASK            ! set bit at slave 8259
08544           popf
08545           mov     eax, 1                  ! disabled by this function
08546           ret
08547   dis_already:
08548           popf
08549           xor     eax, eax                ! already disabled
08550           ret
08551
08552
08553   !*===========================================================================*
08554   !*                              phys_copy                                    *
```

```
08555 !*===========================================================================*
08556 ! PUBLIC void phys_copy(phys_bytes source, phys_bytes destination,
08557 !                       phys_bytes bytecount);
08558 ! Copy a block of physical memory.
08559
08560 PC_ARGS =       4 + 4 + 4 + 4 + 4   ! 4 + 4 + 4
08561 !               es edi esi eip   src dst len
08562
08563         .align  16
08564 _phys_copy:
08565         cld
08566         push    esi
08567         push    edi
08568         push    es
08569
08570         mov     eax, FLAT_DS_SELECTOR
08571         mov     es, ax
08572
08573         mov     esi, PC_ARGS(esp)
08574         mov     edi, PC_ARGS+4(esp)
08575         mov     eax, PC_ARGS+4+4(esp)
08576
08577         cmp     eax, 10                 ! avoid align overhead for small counts
08578         jb      pc_small
08579         mov     ecx, esi                ! align source, hope target is too
08580         neg     ecx
08581         and     ecx, 3                  ! count for alignment
08582         sub     eax, ecx
08583         rep
08584    eseg movsb
08585         mov     ecx, eax
08586         shr     ecx, 2                  ! count of dwords
08587         rep
08588    eseg movs
08589         and     eax, 3
08590 pc_small:
08591         xchg    ecx, eax                ! remainder
08592         rep
08593    eseg movsb
08594
08595         pop     es
08596         pop     edi
08597         pop     esi
08598         ret
08599
08600
08601 !*===========================================================================*
08602 !*                              mem_rdw                                      *
08603 !*===========================================================================*
08604 ! PUBLIC u16_t mem_rdw(U16_t segment, u16_t *offset);
08605 ! Load and return word at far pointer segment:offset.
08606
08607         .align  16
08608 _mem_rdw:
08609         mov     cx, ds
08610         mov     ds, 4(esp)              ! segment
08611         mov     eax, 4+4(esp)           ! offset
08612         movzx   eax, (eax)              ! word to return
08613         mov     ds, cx
08614         ret
```



```
08615
08616
08617 !*===========================================================================*
08618 !*                              reset                                        *
08619 !*===========================================================================*
08620 ! PUBLIC void reset();
08621 ! Reset the system by loading IDT with offset 0 and interrupting.
08622
08623 _reset:
08624         lidt    (idt_zero)
08625         int     3               ! anything goes, the 386 will not like it
08626 .sect .data
08627 idt_zero:       .data4  0, 0
08628 .sect .text
08629
08630
08631 !*===========================================================================*
08632 !*                              mem_vid_copy                                 *
08633 !*===========================================================================*
08634 ! PUBLIC void mem_vid_copy(u16 *src, unsigned dst, unsigned count);
08635 !
08636 ! Copy count characters from kernel memory to video memory.  Src, dst and
08637 ! count are character (word) based video offsets and counts.  If src is null
08638 ! then screen memory is blanked by filling it with blank_color.
08639
08640 MVC_ARGS        =       4 + 4 + 4 + 4 + 4   ! 4 + 4 + 4
08641 !                       es edi esi eip   src dst ct
08642
08643 _mem_vid_copy:
08644         push    esi
08645         push    edi
08646         push    es
08647         mov     esi, MVC_ARGS(esp)      ! source
08648         mov     edi, MVC_ARGS+4(esp)    ! destination
08649         mov     edx, MVC_ARGS+4+4(esp)  ! count
08650         mov     es, (_vid_seg)          ! destination is video segment
08651         cld                             ! make sure direction is up
08652 mvc_loop:
08653         and     edi, (_vid_mask)        ! wrap address
08654         mov     ecx, edx                ! one chunk to copy
08655         mov     eax, (_vid_size)
08656         sub     eax, edi
08657         cmp     ecx, eax
08658         jbe     0f
08659         mov     ecx, eax                ! ecx = min(ecx, vid_size - edi)
08660 0:      sub     edx, ecx                ! count -= ecx
08661         shl     edi, 1                  ! byte address
08662         test    esi, esi                ! source == 0 means blank the screen
08663         jz      mvc_blank
08664 mvc_copy:
08665         rep                             ! copy words to video memory
08666     o16 movs
08667         jmp     mvc_test
08668 mvc_blank:
08669         mov     eax, (_blank_color)     ! ax = blanking character
08670         rep
08671     o16 stos                            ! copy blanks to video memory
08672         !jmp    mvc_test
08673 mvc_test:
08674         shr     edi, 1                  ! word addresses
```

```
08675           test    edx, edx
08676           jnz     mvc_loop
08677   mvc_done:
08678           pop     es
08679           pop     edi
08680           pop     esi
08681           ret
08682
08683
08684   !*===========================================================================*
08685   !*                              vid_vid_copy                                 *
08686   !*===========================================================================*
08687   ! PUBLIC void vid_vid_copy(unsigned src, unsigned dst, unsigned count);
08688   !
08689   ! Copy count characters from video memory to video memory.  Handle overlap.
08690   ! Used for scrolling, line or character insertion and deletion.  Src, dst
08691   ! and count are character (word) based video offsets and counts.
08692
08693   VVC_ARGS        =       4 + 4 + 4 + 4 + 4   ! 4 + 4 + 4
08694   !                       es edi esi eip   src dst ct
08695
08696   _vid_vid_copy:
08697           push    esi
08698           push    edi
08699           push    es
08700           mov     esi, VVC_ARGS(esp)      ! source
08701           mov     edi, VVC_ARGS+4(esp)    ! destination
08702           mov     edx, VVC_ARGS+4+4(esp)  ! count
08703           mov     es, (_vid_seg)          ! use video segment
08704           cmp     esi, edi                ! copy up or down?
08705           jb      vvc_down
08706   vvc_up:
08707           cld                             ! direction is up
08708   vvc_uploop:
08709           and     esi, (_vid_mask)        ! wrap addresses
08710           and     edi, (_vid_mask)
08711           mov     ecx, edx                ! one chunk to copy
08712           mov     eax, (_vid_size)
08713           sub     eax, esi
08714           cmp     ecx, eax
08715           jbe     0f
08716           mov     ecx, eax                ! ecx = min(ecx, vid_size - esi)
08717   0:      mov     eax, (_vid_size)
08718           sub     eax, edi
08719           cmp     ecx, eax
08720           jbe     0f
08721           mov     ecx, eax                ! ecx = min(ecx, vid_size - edi)
08722   0:      sub     edx, ecx                ! count -= ecx
08723           shl     esi, 1
08724           shl     edi, 1                  ! byte addresses
08725           rep
08726   eseg o16 movs                           ! copy video words
08727           shr     esi, 1
08728           shr     edi, 1                  ! word addresses
08729           test    edx, edx
08730           jnz     vvc_uploop              ! again?
08731           jmp     vvc_done
08732   vvc_down:
08733           std                             ! direction is down
08734           lea     esi, -1(esi)(edx*1)     ! start copying at the top
```



```
08735           lea     edi, -1(edi)(edx*1)
08736   vvc_downloop:
08737           and     esi, (_vid_mask)        ! wrap addresses
08738           and     edi, (_vid_mask)
08739           mov     ecx, edx                ! one chunk to copy
08740           lea     eax, 1(esi)
08741           cmp     ecx, eax
08742           jbe     0f
08743           mov     ecx, eax                ! ecx = min(ecx, esi + 1)
08744   0:      lea     eax, 1(edi)
08745           cmp     ecx, eax
08746           jbe     0f
08747           mov     ecx, eax                ! ecx = min(ecx, edi + 1)
08748   0:      sub     edx, ecx                ! count -= ecx
08749           shl     esi, 1
08750           shl     edi, 1                  ! byte addresses
08751           rep
08752   eseg o16 movs                           ! copy video words
08753           shr     esi, 1
08754           shr     edi, 1                  ! word addresses
08755           test    edx, edx
08756           jnz     vvc_downloop            ! again?
08757           cld                             ! C compiler expect up
08758           !jmp    vvc_done
08759   vvc_done:
08760           pop     es
08761           pop     edi
08762           pop     esi
08763           ret
08764
08765
08766   !*===========================================================================*
08767   !*                              level0                                       *
08768   !*===========================================================================*
08769   ! PUBLIC void level0(void (*func)(void))
08770   ! Call a function at permission level 0.  This allows kernel tasks to do
08771   ! things that are only possible at the most privileged CPU level.
08772   !
08773   _level0:
08774           mov     eax, 4(esp)
08775           mov     (_level0_func), eax
08776           int     LEVEL0_VECTOR
08777           ret



++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/kernel/misc.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

08800   /* This file contains a collection of miscellaneous procedures:
08801    *      mem_init:       initialize memory tables.  Some memory is reported
08802    *                      by the BIOS, some is guesstimated and checked later
08803    *      env_parse       parse environment variable.
08804    *      bad_assertion   for debugging
08805    *      bad_compare     for debugging
08806    */
08807
08808   #include "kernel.h"
08809   #include "assert.h"
```

```
08810   #include <stdlib.h>
08811   #include <minix/com.h>
08812
08813   #define EM_BASE     0x100000L   /* base of extended memory on AT's */
08814   #define SHADOW_BASE 0xFA0000L   /* base of RAM shadowing ROM on some AT's */
08815   #define SHADOW_MAX  0x060000L   /* maximum usable shadow memory (16M limit) */
08816
08817   /*==========================================================================*
08818    *                              mem_init                                    *
08819    *==========================================================================*/
08820   PUBLIC void mem_init()
08821   {
08822   /* Initialize the memory size tables.  This is complicated by fragmentation
08823    * and different access strategies for protected mode.  There must be a
08824    * chunk at 0 big enough to hold Minix proper.  For 286 and 386 processors,
08825    * there can be extended memory (memory above 1MB).  This usually starts at
08826    * 1MB, but there may be another chunk just below 16MB, reserved under DOS
08827    * for shadowing ROM, but available to Minix if the hardware can be re-mapped.
08828    * In protected mode, extended memory is accessible assuming CLICK_SIZE is
08829    * large enough, and is treated as ordinary memory.
08830    */
08831
08832     u32_t ext_clicks;
08833     phys_clicks max_clicks;
08834
08835     /* Get the size of ordinary memory from the BIOS. */
08836     mem[0].size = k_to_click(low_memsize);        /* base = 0 */
08837
08838     if (pc_at && protected_mode) {
08839           /* Get the size of extended memory from the BIOS.  This is special
08840            * except in protected mode, but protected mode is now normal.
08841            * Note that no more than 16M can be addressed in 286 mode, so make
08842            * sure that the highest memory address fits in a short when counted
08843            * in clicks.
08844            */
08845           ext_clicks = k_to_click((u32_t) ext_memsize);
08846           max_clicks = USHRT_MAX - (EM_BASE >> CLICK_SHIFT);
08847           mem[1].size = MIN(ext_clicks, max_clicks);
08848           mem[1].base = EM_BASE >> CLICK_SHIFT;
08849
08850           if (ext_memsize <= (unsigned) ((SHADOW_BASE - EM_BASE) / 1024)
08851                           && check_mem(SHADOW_BASE, SHADOW_MAX) == SHADOW_MAX) {
08852                   /* Shadow ROM memory. */
08853                   mem[2].size = SHADOW_MAX >> CLICK_SHIFT;
08854                   mem[2].base = SHADOW_BASE >> CLICK_SHIFT;
08855           }
08856     }
08857
08858     /* Total system memory. */
08859     tot_mem_size = mem[0].size + mem[1].size + mem[2].size;
08860   }

08862   /*==========================================================================*
08863    *                              env_parse                                   *
08864    *==========================================================================*/
08865   PUBLIC int env_parse(env, fmt, field, param, min, max)
08866   char *env;              /* environment variable to inspect */
08867   char *fmt;              /* template to parse it with */
08868   int field;              /* field number of value to return */
08869   long *param;            /* address of parameter to get */
```



```
08870   long min, max;          /* minimum and maximum values for the parameter */
08871   {
08872   /* Parse an environment variable setting, something like "DPETH0=300:3".
08873    * Panic if the parsing fails.  Return EP_UNSET if the environment variable
08874    * is not set, EP_OFF if it is set to "off", EP_ON if set to "on" or a
08875    * field is left blank, or EP_SET if a field is given (return value through
08876    * *param).  Commas and colons may be used in the environment and format
08877    * string, fields in the environment string may be empty, and punctuation
08878    * may be missing to skip fields.  The format string contains characters
08879    * 'd', 'o', 'x' and 'c' to indicate that 10, 8, 16, or 0 is used as the
08880    * last argument to strtol.
08881    */
08882
08883     char *val, *end;
08884     long newpar;
08885     int i = 0, radix, r;
08886
08887     if ((val = k_getenv(env)) == NIL_PTR) return(EP_UNSET);
08888     if (strcmp(val, "off") == 0) return(EP_OFF);
08889     if (strcmp(val, "on") == 0) return(EP_ON);
08890
08891     r = EP_ON;
08892     for (;;) {
08893           while (*val == ' ') val++;
08894
08895           if (*val == 0) return(r);       /* the proper exit point */
08896
08897           if (*fmt == 0) break;           /* too many values */
08898
08899           if (*val == ',' || *val == ':') {
08900                   /* Time to go to the next field. */
08901                   if (*fmt == ',' || *fmt == ':') i++;
08902                   if (*fmt++ == *val) val++;
08903           } else {
08904                   /* Environment contains a value, get it. */
08905                   switch (*fmt) {
08906                   case 'd':       radix =   10;   break;
08907                   case 'o':       radix =  010;   break;
08908                   case 'x':       radix = 0x10;   break;
08909                   case 'c':       radix =    0;   break;
08910                   default:        goto badenv;
08911                   }
08912                   newpar = strtol(val, &end, radix);
08913
08914                   if (end == val) break;  /* not a number */
08915                   val = end;
08916
08917                   if (i == field) {
08918                           /* The field requested. */
08919                           if (newpar < min || newpar > max) break;
08920                           *param = newpar;
08921                           r = EP_SET;
08922                   }
08923           }
08924     }
08925   badenv:
08926     printf("Bad environment setting: '%s = %s'\n", env, k_getenv(env));
08927     panic("", NO_NUM);
08928     /*NOTREACHED*/
08929   }
```

```
08931    #if DEBUG
08932    /*===========================================================================*
08933     *                               bad_assertion                               *
08934     *===========================================================================*/
08935    PUBLIC void bad_assertion(file, line, what)
08936    char *file;
08937    int line;
08938    char *what;
08939    {
08940      printf("panic at %s(%d): assertion \"%s\" failed\n", file, line, what);
08941      panic(NULL, NO_NUM);
08942    }

08944    /*===========================================================================*
08945     *                               bad_compare                                 *
08946     *===========================================================================*/
08947    PUBLIC void bad_compare(file, line, lhs, what, rhs)
08948    char *file;
08949    int line;
08950    int lhs;
08951    char *what;
08952    int rhs;
08953    {
08954      printf("panic at %s(%d): compare (%d) %s (%d) failed\n",
08955            file, line, lhs, what, rhs);
08956      panic(NULL, NO_NUM);
08957    }
08958    #endif /* DEBUG */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/kernel/driver.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

09000    /* Types and constants shared between the generic and device dependent
09001     * device driver code.
09002     */
09003
09004    #include <minix/callnr.h>
09005    #include <minix/com.h>
09006    #include "proc.h"
09007    #include <minix/partition.h>
09008
09009    /* Info about and entry points into the device dependent code. */
09010    struct driver {
09011      _PROTOTYPE( char *(*dr_name), (void) );
09012      _PROTOTYPE( int (*dr_open), (struct driver *dp, message *m_ptr) );
09013      _PROTOTYPE( int (*dr_close), (struct driver *dp, message *m_ptr) );
09014      _PROTOTYPE( int (*dr_ioctl), (struct driver *dp, message *m_ptr) );
09015      _PROTOTYPE( struct device *(*dr_prepare), (int device) );
09016      _PROTOTYPE( int (*dr_schedule), (int proc_nr, struct iorequest_s *request) );
09017      _PROTOTYPE( int (*dr_finish), (void) );
09018      _PROTOTYPE( void (*dr_cleanup), (void) );
09019      _PROTOTYPE( void (*dr_geometry), (struct partition *entry) );
09020    };
09021
09022    #if (CHIP == INTEL)
09023
09024    /* Number of bytes you can DMA before hitting a 64K boundary: */
```



```
09025    #define dma_bytes_left(phys)    \
09026       ((unsigned) (sizeof(int) == 2 ? 0 : 0x10000) - (unsigned) ((phys) & 0xFFFF))
09027
09028    #endif /* CHIP == INTEL */
09029
09030    /* Base and size of a partition in bytes. */
09031    struct device {
09032      unsigned long dv_base;
09033      unsigned long dv_size;
09034    };
09035
09036    #define NIL_DEV         ((struct device *) 0)
09037
09038    /* Functions defined by driver.c: */
09039    _PROTOTYPE( void driver_task, (struct driver *dr) );
09040    _PROTOTYPE( int do_rdwt, (struct driver *dr, message *m_ptr) );
09041    _PROTOTYPE( int do_vrdwt, (struct driver *dr, message *m_ptr) );
09042    _PROTOTYPE( char *no_name, (void) );
09043    _PROTOTYPE( int do_nop, (struct driver *dp, message *m_ptr) );
09044    _PROTOTYPE( int nop_finish, (void) );
09045    _PROTOTYPE( void nop_cleanup, (void) );
09046    _PROTOTYPE( void clock_mess, (int ticks, watchdog_t func) );
09047    _PROTOTYPE( int do_diocntl, (struct driver *dr, message *m_ptr) );
09048
09049    /* Parameters for the disk drive. */
09050    #define SECTOR_SIZE      512    /* physical sector size in bytes */
09051    #define SECTOR_SHIFT       9    /* for division */
09052    #define SECTOR_MASK      511    /* and remainder */
09053
09054    /* Size of the DMA buffer buffer in bytes. */
09055    #define DMA_BUF_SIZE    (DMA_SECTORS * SECTOR_SIZE)
09056
09057    #if (CHIP == INTEL)
09058    extern u8_t *tmp_buf;                   /* the DMA buffer */
09059    #else
09060    extern u8_t tmp_buf[];                  /* the DMA buffer */
09061    #endif
09062    extern phys_bytes tmp_phys;             /* phys address of DMA buffer */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/kernel/driver.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

09100    /* This file contains device independent device driver interface.
09101     *                                                     Author: Kees J. Bot.
09102     *
09103     * The drivers support the following operations (using message format m2):
09104     *
09105     *    m_type      DEVICE    PROC_NR     COUNT    POSITION  ADRRESS
09106     * ----------------------------------------------------------------
09107     * |  DEV_OPEN  | device  | proc nr |         |         |         |
09108     * |------------+---------+---------+---------+---------+---------|
09109     * |  DEV_CLOSE | device  | proc nr |         |         |         |
09110     * |------------+---------+---------+---------+---------+---------|
09111     * |  DEV_READ  | device  | proc nr |  bytes  |  offset | buf ptr |
09112     * |------------+---------+---------+---------+---------+---------|
09113     * |  DEV_WRITE | device  | proc nr |  bytes  |  offset | buf ptr |
09114     * |------------+---------+---------+---------+---------+---------|
```

```
09115     * |SCATTERED_IO| device  | proc nr | requests|         | iov ptr |
09116     * ------------------------------------------------------------
09117     * |  DEV_IOCTL | device  | proc nr |func code|         | buf ptr |
09118     * ------------------------------------------------------------
09119     *
09120     * The file contains one entry point:
09121     *
09122     *   driver_task:       called by the device dependent task entry
09123     *
09124     *
09125     * Constructed 92/04/02 by Kees J. Bot from the old AT wini and floppy driver.
09126     */
09127
09128    #include "kernel.h"
09129    #include <sys/ioctl.h>
09130    #include "driver.h"
09131
09132    #define BUF_EXTRA       0
09133
09134    /* Claim space for variables. */
09135    PRIVATE u8_t buffer[(unsigned) 2 * DMA_BUF_SIZE + BUF_EXTRA];
09136    u8_t *tmp_buf;                  /* the DMA buffer eventually */
09137    phys_bytes tmp_phys;            /* phys address of DMA buffer */
09138
09139    FORWARD _PROTOTYPE( void init_buffer, (void) );
09140
09141    /*===========================================================================*
09142     *                              driver_task                                  *
09143     *===========================================================================*/
09144    PUBLIC void driver_task(dp)
09145    struct driver *dp;      /* Device dependent entry points. */
09146    {
09147    /* Main program of any device driver task. */
09148
09149      int r, caller, proc_nr;
09150      message mess;
09151
09152      init_buffer();        /* Get a DMA buffer. */
09153
09154      /* Here is the main loop of the disk task.  It waits for a message, carries
09155       * it out, and sends a reply.
09156       */
09157
09158      while (TRUE) {
09159            /* First wait for a request to read or write a disk block. */
09160            receive(ANY, &mess);
09161
09162            caller = mess.m_source;
09163            proc_nr = mess.PROC_NR;
09164
09165            switch (caller) {
09166            case HARDWARE:
09167                    /* Leftover interrupt. */
09168                    continue;
09169            case FS_PROC_NR:
09170                    /* The only legitimate caller. */
09171                    break;
09172            default:
09173                    printf("%s: got message from %d\n", (*dp->dr_name)(), caller);
09174                    continue;
```



```
09175           }
09176
09177           /* Now carry out the work. */
09178           switch(mess.m_type) {
09179               case DEV_OPEN:      r = (*dp->dr_open)(dp, &mess);  break;
09180               case DEV_CLOSE:     r = (*dp->dr_close)(dp, &mess); break;
09181               case DEV_IOCTL:     r = (*dp->dr_ioctl)(dp, &mess); break;
09182
09183               case DEV_READ:
09184               case DEV_WRITE:     r = do_rdwt(dp, &mess);         break;
09185
09186               case SCATTERED_IO:  r = do_vrdwt(dp, &mess);        break;
09187               default:            r = EINVAL;                     break;
09188           }
09189
09190           /* Clean up leftover state. */
09191           (*dp->dr_cleanup)();
09192
09193           /* Finally, prepare and send the reply message. */
09194           mess.m_type = TASK_REPLY;
09195           mess.REP_PROC_NR = proc_nr;
09196
09197           mess.REP_STATUS = r;    /* # of bytes transferred or error code */
09198           send(caller, &mess);    /* send reply to caller */
09199       }
09200   }

09202   /*===========================================================================*
09203    *                              init_buffer                                  *
09204    *===========================================================================*/
09205   PRIVATE void init_buffer()
09206   {
09207   /* Select a buffer that can safely be used for dma transfers.  It may also
09208    * be used to read partition tables and such.  Its absolute address is
09209    * 'tmp_phys', the normal address is 'tmp_buf'.
09210    */
09211
09212     tmp_buf = buffer;
09213     tmp_phys = vir2phys(buffer);
09214
09215     if (tmp_phys == 0) panic("no DMA buffer", NO_NUM);
09216
09217     if (dma_bytes_left(tmp_phys) < DMA_BUF_SIZE) {
09218           /* First half of buffer crosses a 64K boundary, can't DMA into that */
09219           tmp_buf += DMA_BUF_SIZE;
09220           tmp_phys += DMA_BUF_SIZE;
09221     }
09222   }

09224   /*===========================================================================*
09225    *                              do_rdwt                                      *
09226    *===========================================================================*/
09227   PUBLIC int do_rdwt(dp, m_ptr)
09228   struct driver *dp;              /* device dependent entry points */
09229   message *m_ptr;                 /* pointer to read or write message */
09230   {
09231   /* Carry out a single read or write request. */
09232     struct iorequest_s ioreq;
09233     int r;
09234
```

```
09235     if (m_ptr->COUNT <= 0) return(EINVAL);
09236
09237     if ((*dp->dr_prepare)(m_ptr->DEVICE) == NIL_DEV) return(ENXIO);
09238
09239     ioreq.io_request = m_ptr->m_type;
09240     ioreq.io_buf = m_ptr->ADDRESS;
09241     ioreq.io_position = m_ptr->POSITION;
09242     ioreq.io_nbytes = m_ptr->COUNT;
09243
09244     r = (*dp->dr_schedule)(m_ptr->PROC_NR, &ioreq);
09245
09246     if (r == OK) (void) (*dp->dr_finish)();
09247
09248     r = ioreq.io_nbytes;
09249     return(r < 0 ? r : m_ptr->COUNT - r);
09250   }

09252   /*===========================================================================*
09253    *                                do_vrdwt                                   *
09254    *===========================================================================*/
09255   PUBLIC int do_vrdwt(dp, m_ptr)
09256   struct driver *dp;      /* device dependent entry points */
09257   message *m_ptr;         /* pointer to read or write message */
09258   {
09259   /* Fetch a vector of i/o requests.  Handle requests one at a time.  Return
09260    * status in the vector.
09261    */
09262
09263     struct iorequest_s *iop;
09264     static struct iorequest_s iovec[NR_IOREQS];
09265     phys_bytes iovec_phys;
09266     unsigned nr_requests;
09267     int request;
09268     int r;
09269     phys_bytes user_iovec_phys;
09270
09271     nr_requests = m_ptr->COUNT;
09272
09273     if (nr_requests > sizeof iovec / sizeof iovec[0])
09274           panic("FS passed too big an I/O vector", nr_requests);
09275
09276     iovec_phys = vir2phys(iovec);
09277     user_iovec_phys = numap(m_ptr->PROC_NR, (vir_bytes) m_ptr->ADDRESS,
09278                             (vir_bytes) (nr_requests * sizeof iovec[0]));
09279
09280     if (user_iovec_phys == 0)
09281           panic("FS passed a bad I/O vector", (int) m_ptr->ADDRESS);
09282
09283     phys_copy(user_iovec_phys, iovec_phys,
09284                               (phys_bytes) nr_requests * sizeof iovec[0]);
09285
09286     if ((*dp->dr_prepare)(m_ptr->DEVICE) == NIL_DEV) return(ENXIO);
09287
09288     for (request = 0, iop = iovec; request < nr_requests; request++, iop++) {
09289           if ((r = (*dp->dr_schedule)(m_ptr->PROC_NR, iop)) != OK) break;
09290     }
09291
09292     if (r == OK) (void) (*dp->dr_finish)();
09293
09294     phys_copy(iovec_phys, user_iovec_phys,
```



```
09295                               (phys_bytes) nr_requests * sizeof iovec[0]);
09296     return(OK);
09297   }

09299   /*===========================================================================*
09300    *                                no_name                                    *
09301    *===========================================================================*/
09302   PUBLIC char *no_name()
09303   {
09304   /* If no specific name for the device. */
09305
09306     return(tasktab[proc_number(proc_ptr) + NR_TASKS].name);
09307   }

09309   /*===========================================================================*
09310    *                                do_nop                                     *
09311    *===========================================================================*/
09312   PUBLIC int do_nop(dp, m_ptr)
09313   struct driver *dp;
09314   message *m_ptr;
09315   {
09316   /* Nothing there, or nothing to do. */
09317
09318     switch (m_ptr->m_type) {
09319     case DEV_OPEN:        return(ENODEV);
09320     case DEV_CLOSE:       return(OK);
09321     case DEV_IOCTL:       return(ENOTTY);
09322     default:              return(EIO);
09323     }
09324   }

09326   /*===========================================================================*
09327    *                                nop_finish                                 *
09328    *===========================================================================*/
09329   PUBLIC int nop_finish()
09330   {
09331   /* Nothing to finish, all the work has been done by dp->dr_schedule. */
09332     return(OK);
09333   }

09335   /*===========================================================================*
09336    *                                nop_cleanup                                *
09337    *===========================================================================*/
09338   PUBLIC void nop_cleanup()
09339   {
09340   /* Nothing to clean up. */
09341   }

09343   /*===========================================================================*
09344    *                                clock_mess                                 *
09345    *===========================================================================*/
09346   PUBLIC void clock_mess(ticks, func)
09347   int ticks;                      /* how many clock ticks to wait */
09348   watchdog_t func;                /* function to call upon time out */
09349   {
09350   /* Send the clock task a message. */
09351
09352     message mess;
09353
09354     mess.m_type = SET_ALARM;
```

```
09355     mess.CLOCK_PROC_NR = proc_number(proc_ptr);
09356     mess.DELTA_TICKS = (long) ticks;
09357     mess.FUNC_TO_CALL = (sighandler_t) func;
09358     sendrec(CLOCK, &mess);
09359   }

09361   /*===========================================================================*
09362    *                              do_diocntl                                   *
09363    *===========================================================================*/
09364   PUBLIC int do_diocntl(dp, m_ptr)
09365   struct driver *dp;
09366   message *m_ptr;                 /* pointer to ioctl request */
09367   {
09368   /* Carry out a partition setting/getting request. */
09369     struct device *dv;
09370     phys_bytes user_phys, entry_phys;
09371     struct partition entry;
09372
09373     if (m_ptr->REQUEST != DIOCSETP && m_ptr->REQUEST != DIOCGETP) return(ENOTTY);
09374
09375     /* Decode the message parameters. */
09376     if ((dv = (*dp->dr_prepare)(m_ptr->DEVICE)) == NIL_DEV) return(ENXIO);
09377
09378     user_phys = numap(m_ptr->PROC_NR, (vir_bytes) m_ptr->ADDRESS, sizeof(entry));
09379     if (user_phys == 0) return(EFAULT);
09380
09381     entry_phys = vir2phys(&entry);
09382
09383     if (m_ptr->REQUEST == DIOCSETP) {
09384           /* Copy just this one partition table entry. */
09385           phys_copy(user_phys, entry_phys, (phys_bytes) sizeof(entry));
09386           dv->dv_base = entry.base;
09387           dv->dv_size = entry.size;
09388     } else {
09389           /* Return a partition table entry and the geometry of the drive. */
09390           entry.base = dv->dv_base;
09391           entry.size = dv->dv_size;
09392           (*dp->dr_geometry)(&entry);
09393           phys_copy(entry_phys, user_phys, (phys_bytes) sizeof(entry));
09394     }
09395     return(OK);
09396   }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                              src/kernel/drvlib.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

09400   /* IBM device driver definitions                        Author: Kees J. Bot
09401    *                                                             7 Dec 1995
09402    */
09403
09404   #include <ibm/partition.h>
09405
09406   _PROTOTYPE( void partition, (struct driver *dr, int device, int style) );
09407
09408   /* BIOS parameter table layout. */
09409   #define bp_cylinders(t)         (* (u16_t *) (&(t)[0]))
```



```
09410   #define bp_heads(t)             (* (u8_t *)  (&(t)[2]))
09411   #define bp_reduced_wr(t)        (* (u16_t *) (&(t)[3]))
09412   #define bp_precomp(t)           (* (u16_t *) (&(t)[5]))
09413   #define bp_max_ecc(t)           (* (u8_t *)  (&(t)[7]))
09414   #define bp_ctlbyte(t)           (* (u8_t *)  (&(t)[8]))
09415   #define bp_landingzone(t)       (* (u16_t *) (&(t)[12]))
09416   #define bp_sectors(t)           (* (u8_t *)  (&(t)[14]))
09417
09418   /* Miscellaneous. */
09419   #define DEV_PER_DRIVE   (1 + NR_PARTITIONS)
09420   #define MINOR_hd1a      128
09421   #define MINOR_fd0a      (28<<2)
09422   #define P_FLOPPY        0
09423   #define P_PRIMARY       1
09424   #define P_SUB           2
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                              src/kernel/drvlib.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

09500   /* IBM device driver utility functions.                 Author: Kees J. Bot
09501    *                                                             7 Dec 1995
09502    * Entry point:
09503    *   partition: partition a disk to the partition table(s) on it.
09504    */
09505
09506   #include "kernel.h"
09507   #include "driver.h"
09508   #include "drvlib.h"
09509
09510
09511   FORWARD _PROTOTYPE( void extpartition, (struct driver *dp, int extdev,
09512                                                   unsigned long extbase) );
09513   FORWARD _PROTOTYPE( int get_part_table, (struct driver *dp, int device,
09514                           unsigned long offset, struct part_entry *table) );
09515   FORWARD _PROTOTYPE( void sort, (struct part_entry *table) );
09516
09517
09518   /*===========================================================================*
09519    *                              partition                                    *
09520    *===========================================================================*/
09521   PUBLIC void partition(dp, device, style)
09522   struct driver *dp;      /* device dependent entry points */
09523   int device;             /* device to partition */
09524   int style;              /* partitioning style: floppy, primary, sub. */
09525   {
09526   /* This routine is called on first open to initialize the partition tables
09527    * of a device.  It makes sure that each partition falls safely within the
09528    * device's limits.  Depending on the partition style we are either making
09529    * floppy partitions, primary partitions or subpartitions.  Only primary
09530    * partitions are sorted, because they are shared with other operating
09531    * systems that expect this.
09532    */
09533     struct part_entry table[NR_PARTITIONS], *pe;
09534     int disk, par;
```

```
09535     struct device *dv;
09536     unsigned long base, limit, part_limit;
09537
09538     /* Get the geometry of the device to partition */
09539     if ((dv = (*dp->dr_prepare)(device)) == NIL_DEV || dv->dv_size == 0) return;
09540     base = dv->dv_base >> SECTOR_SHIFT;
09541     limit = base + (dv->dv_size >> SECTOR_SHIFT);
09542
09543     /* Read the partition table for the device. */
09544     if (!get_part_table(dp, device, 0L, table)) return;
09545
09546     /* Compute the device number of the first partition. */
09547     switch (style) {
09548     case P_FLOPPY:
09549           device += MINOR_fd0a;
09550           break;
09551     case P_PRIMARY:
09552           sort(table);            /* sort a primary partition table */
09553           device += 1;
09554           break;
09555     case P_SUB:
09556           disk = device / DEV_PER_DRIVE;
09557           par = device % DEV_PER_DRIVE - 1;
09558           device = MINOR_hdla + (disk * NR_PARTITIONS + par) * NR_PARTITIONS;
09559     }
09560
09561     /* Find an array of devices. */
09562     if ((dv = (*dp->dr_prepare)(device)) == NIL_DEV) return;
09563
09564     /* Set the geometry of the partitions from the partition table. */
09565     for (par = 0; par < NR_PARTITIONS; par++, dv++) {
09566           /* Shrink the partition to fit within the device. */
09567           pe = &table[par];
09568           part_limit = pe->lowsec + pe->size;
09569           if (part_limit < pe->lowsec) part_limit = limit;
09570           if (part_limit > limit) part_limit = limit;
09571           if (pe->lowsec < base) pe->lowsec = base;
09572           if (part_limit < pe->lowsec) part_limit = pe->lowsec;
09573
09574           dv->dv_base = pe->lowsec << SECTOR_SHIFT;
09575           dv->dv_size = (part_limit - pe->lowsec) << SECTOR_SHIFT;
09576
09577           if (style == P_PRIMARY) {
09578                   /* Each Minix primary partition can be subpartitioned. */
09579                   if (pe->sysind == MINIX_PART)
09580                           partition(dp, device + par, P_SUB);
09581
09582                   /* An extended partition has logical partitions. */
09583                   if (pe->sysind == EXT_PART)
09584                           extpartition(dp, device + par, pe->lowsec);
09585           }
09586     }
09587   }


09590   /*============================================================================*
09591    *                              extpartition                                  *
09592    *============================================================================*/
09593   PRIVATE void extpartition(dp, extdev, extbase)
09594   struct driver *dp;      /* device dependent entry points */
```



```
09595   int extdev;             /* extended partition to scan */
09596   unsigned long extbase;  /* sector offset of the base extended partition */
09597   {
09598   /* Extended partitions cannot be ignored alas, because people like to move
09599    * files to and from DOS partitions.  Avoid reading this code, it's no fun.
09600    */
09601     struct part_entry table[NR_PARTITIONS], *pe;
09602     int subdev, disk, par;
09603     struct device *dv;
09604     unsigned long offset, nextoffset;
09605
09606     disk = extdev / DEV_PER_DRIVE;
09607     par = extdev % DEV_PER_DRIVE - 1;
09608     subdev = MINOR_hdla + (disk * NR_PARTITIONS + par) * NR_PARTITIONS;
09609
09610     offset = 0;
09611     do {
09612           if (!get_part_table(dp, extdev, offset, table)) return;
09613           sort(table);
09614
09615           /* The table should contain one logical partition and optionally
09616            * another extended partition.  (It's a linked list.)
09617            */
09618           nextoffset = 0;
09619           for (par = 0; par < NR_PARTITIONS; par++) {
09620                   pe = &table[par];
09621                   if (pe->sysind == EXT_PART) {
09622                           nextoffset = pe->lowsec;
09623                   } else
09624                   if (pe->sysind != NO_PART) {
09625                           if ((dv = (*dp->dr_prepare)(subdev)) == NIL_DEV) return;
09626
09627                           dv->dv_base = (extbase + offset
09628                                           + pe->lowsec) << SECTOR_SHIFT;
09629                           dv->dv_size = pe->size << SECTOR_SHIFT;
09630
09631                           /* Out of devices? */
09632                           if (++subdev % NR_PARTITIONS == 0) return;
09633                   }
09634           }
09635     } while ((offset = nextoffset) != 0);
09636   }


09639   /*============================================================================*
09640    *                              get_part_table                                *
09641    *============================================================================*/
09642   PRIVATE int get_part_table(dp, device, offset, table)
09643   struct driver *dp;
09644   int device;
09645   unsigned long offset;           /* sector offset to the table */
09646   struct part_entry *table;       /* four entries */
09647   {
09648   /* Read the partition table for the device, return true iff there were no
09649    * errors.
09650    */
09651     message mess;
09652
09653     mess.DEVICE = device;
09654     mess.POSITION = offset << SECTOR_SHIFT;
```

```
09655       mess.COUNT = SECTOR_SIZE;
09656       mess.ADDRESS = (char *) tmp_buf;
09657       mess.PROC_NR = proc_number(proc_ptr);
09658       mess.m_type = DEV_READ;
09659
09660       if (do_rdwt(dp, &mess) != SECTOR_SIZE) {
09661             printf("%s: can't read partition table\n", (*dp->dr_name)());
09662             return 0;
09663       }
09664       if (tmp_buf[510] != 0x55 || tmp_buf[511] != 0xAA) {
09665             /* Invalid partition table. */
09666             return 0;
09667       }
09668       memcpy(table, (tmp_buf + PART_TABLE_OFF), NR_PARTITIONS * sizeof(table[0]));
09669       return 1;
09670   }


09673   /*============================================================================*
09674    *                              sort                                          *
09675    *============================================================================*/
09676   PRIVATE void sort(table)
09677   struct part_entry *table;
09678   {
09679   /* Sort a partition table. */
09680     struct part_entry *pe, tmp;
09681     int n = NR_PARTITIONS;
09682
09683     do {
09684           for (pe = table; pe < table + NR_PARTITIONS-1; pe++) {
09685                 if (pe[0].sysind == NO_PART
09686                       || (pe[0].lowsec > pe[1].lowsec
09687                                   && pe[1].sysind != NO_PART)) {
09688                       tmp = pe[0]; pe[0] = pe[1]; pe[1] = tmp;
09689                 }
09690           }
09691     } while (--n > 0);
09692   }


++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                              src/kernel/memory.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

09700   /* This file contains the device dependent part of the drivers for the
09701    * following special files:
09702    *     /dev/null        - null device (data sink)
09703    *     /dev/mem         - absolute memory
09704    *     /dev/kmem        - kernel virtual memory
09705    *     /dev/ram         - RAM disk
09706    *
09707    * The file contains one entry point:
09708    *
09709    *   mem_task:  main entry when system is brought up
09710    *
09711    *  Changes:
09712    *      20 Apr  1992 by Kees J. Bot: device dependent/independent split
09713    */
09714
```



```
09715   #include "kernel.h"
09716   #include "driver.h"
09717   #include <sys/ioctl.h>
09718
09719   #define NR_RAMS            4    /* number of RAM-type devices */
09720
09721   PRIVATE struct device m_geom[NR_RAMS];  /* Base and size of each RAM disk */
09722   PRIVATE int m_device;           /* current device */
09723
09724   FORWARD _PROTOTYPE( struct device *m_prepare, (int device) );
09725   FORWARD _PROTOTYPE( int m_schedule, (int proc_nr, struct iorequest_s *iop) );
09726   FORWARD _PROTOTYPE( int m_do_open, (struct driver *dp, message *m_ptr) );
09727   FORWARD _PROTOTYPE( void m_init, (void) );
09728   FORWARD _PROTOTYPE( int m_ioctl, (struct driver *dp, message *m_ptr) );
09729   FORWARD _PROTOTYPE( void m_geometry, (struct partition *entry) );
09730
09731
09732   /* Entry points to this driver. */
09733   PRIVATE struct driver m_dtab = {
09734     no_name,      /* current device's name */
09735     m_do_open,    /* open or mount */
09736     do_nop,       /* nothing on a close */
09737     m_ioctl,      /* specify ram disk geometry */
09738     m_prepare,    /* prepare for I/O on a given minor device */
09739     m_schedule,   /* do the I/O */
09740     nop_finish,   /* schedule does the work, no need to be smart */
09741     nop_cleanup,  /* nothing's dirty */
09742     m_geometry,   /* memory device "geometry" */
09743   };
09744
09745
09746   /*============================================================================*
09747    *                              mem_task                                      *
09748    *============================================================================*/
09749   PUBLIC void mem_task()
09750   {
09751     m_init();
09752     driver_task(&m_dtab);
09753   }


09756   /*============================================================================*
09757    *                              m_prepare                                     *
09758    *============================================================================*/
09759   PRIVATE struct device *m_prepare(device)
09760   int device;
09761   {
09762   /* Prepare for I/O on a device. */
09763
09764     if (device < 0 || device >= NR_RAMS) return(NIL_DEV);
09765     m_device = device;
09766
09767     return(&m_geom[device]);
09768   }


09771   /*============================================================================*
09772    *                              m_schedule                                    *
09773    *============================================================================*/
09774   PRIVATE int m_schedule(proc_nr, iop)
```

```
09775   int proc_nr;                    /* process doing the request */
09776   struct iorequest_s *iop;        /* pointer to read or write request */
09777   {
09778   /* Read or write /dev/null, /dev/mem, /dev/kmem, or /dev/ram. */
09779
09780     int device, count, opcode;
09781     phys_bytes mem_phys, user_phys;
09782     struct device *dv;
09783
09784     /* Type of request */
09785     opcode = iop->io_request & ~OPTIONAL_IO;
09786
09787     /* Get minor device number and check for /dev/null. */
09788     device = m_device;
09789     dv = &m_geom[device];
09790
09791     /* Determine address where data is to go or to come from. */
09792     user_phys = numap(proc_nr, (vir_bytes) iop->io_buf,
09793                                             (vir_bytes) iop->io_nbytes);
09794     if (user_phys == 0) return(iop->io_nbytes = EINVAL);
09795
09796     if (device == NULL_DEV) {
09797           /* /dev/null: Black hole. */
09798           if (opcode == DEV_WRITE) iop->io_nbytes = 0;
09799           count = 0;
09800     } else {
09801           /* /dev/mem, /dev/kmem, or /dev/ram: Check for EOF */
09802           if (iop->io_position >= dv->dv_size) return(OK);
09803           count = iop->io_nbytes;
09804           if (iop->io_position + count > dv->dv_size)
09805                   count = dv->dv_size - iop->io_position;
09806     }
09807
09808     /* Set up 'mem_phys' for /dev/mem, /dev/kmem, or /dev/ram */
09809     mem_phys = dv->dv_base + iop->io_position;
09810
09811     /* Book the number of bytes to be transferred in advance. */
09812     iop->io_nbytes -= count;
09813
09814     if (count == 0) return(OK);
09815
09816     /* Copy the data. */
09817     if (opcode == DEV_READ)
09818           phys_copy(mem_phys, user_phys, (phys_bytes) count);
09819     else
09820           phys_copy(user_phys, mem_phys, (phys_bytes) count);
09821
09822     return(OK);
09823   }


09826   /*===========================================================================*
09827    *                              m_do_open                                    *
09828    *===========================================================================*/
09829   PRIVATE int m_do_open(dp, m_ptr)
09830   struct driver *dp;
09831   message *m_ptr;
09832   {
09833   /* Check device number on open.  Give I/O privileges to a process opening
09834    * /dev/mem or /dev/kmem.
```



```
09835    */
09836
09837     if (m_prepare(m_ptr->DEVICE) == NIL_DEV) return(ENXIO);
09838
09839     if (m_device == MEM_DEV || m_device == KMEM_DEV)
09840           enable_iop(proc_addr(m_ptr->PROC_NR));
09841
09842     return(OK);
09843   }


09846   /*===========================================================================*
09847    *                              m_init                                       *
09848    *===========================================================================*/
09849   PRIVATE void m_init()
09850   {
09851     /* Initialize this task. */
09852     extern int _end;
09853
09854     m_geom[KMEM_DEV].dv_base = vir2phys(0);
09855     m_geom[KMEM_DEV].dv_size = vir2phys(&_end);
09856
09857   #if (CHIP == INTEL)
09858     if (!protected_mode) {
09859           m_geom[MEM_DEV].dv_size =   0x100000;   /* 1M for 8086 systems */
09860     } else {
09861   #if _WORD_SIZE == 2
09862           m_geom[MEM_DEV].dv_size =  0x1000000;   /* 16M for 286 systems */
09863   #else
09864           m_geom[MEM_DEV].dv_size = 0xFFFFFFFF;   /* 4G-1 for 386 systems */
09865   #endif
09866     }
09867   #endif
09868   }


09871   /*===========================================================================*
09872    *                              m_ioctl                                      *
09873    *===========================================================================*/
09874   PRIVATE int m_ioctl(dp, m_ptr)
09875   struct driver *dp;
09876   message *m_ptr;                 /* pointer to read or write message */
09877   {
09878   /* Set parameters for one of the RAM disks. */
09879
09880     unsigned long bytesize;
09881     unsigned base, size;
09882     struct memory *memp;
09883     static struct psinfo psinfo = { NR_TASKS, NR_PROCS, (vir_bytes) proc, 0, 0 };
09884     phys_bytes psinfo_phys;
09885
09886     switch (m_ptr->REQUEST) {
09887     case MIOCRAMSIZE:
09888           /* FS sets the RAM disk size. */
09889           if (m_ptr->PROC_NR != FS_PROC_NR) return(EPERM);
09890
09891           bytesize = m_ptr->POSITION * BLOCK_SIZE;
09892           size = (bytesize + CLICK_SHIFT-1) >> CLICK_SHIFT;
09893
09894           /* Find a memory chunk big enough for the RAM disk. */
```

```
09895           memp= &mem[NR_MEMS];
09896           while ((--memp)->size < size) {
09897                   if (memp == mem) panic("RAM disk is too big", NO_NUM);
09898           }
09899           base = memp->base;
09900           memp->base += size;
09901           memp->size -= size;
09902
09903           m_geom[RAM_DEV].dv_base = (unsigned long) base << CLICK_SHIFT;
09904           m_geom[RAM_DEV].dv_size = bytesize;
09905           break;
09906     case MIOCSPSINFO:
09907           /* MM or FS set the address of their process table. */
09908           if (m_ptr->PROC_NR == MM_PROC_NR) {
09909                   psinfo.mproc = (vir_bytes) m_ptr->ADDRESS;
09910           } else
09911           if (m_ptr->PROC_NR == FS_PROC_NR) {
09912                   psinfo.fproc = (vir_bytes) m_ptr->ADDRESS;
09913           } else {
09914                   return(EPERM);
09915           }
09916           break;
09917     case MIOCGPSINFO:
09918           /* The ps program wants the process table addresses. */
09919           psinfo_phys = numap(m_ptr->PROC_NR, (vir_bytes) m_ptr->ADDRESS,
09920                                                   sizeof(psinfo));
09921           if (psinfo_phys == 0) return(EFAULT);
09922           phys_copy(vir2phys(&psinfo), psinfo_phys, (phys_bytes) sizeof(psinfo));
09923           break;
09924     default:
09925           return(do_diocntl(&m_dtab, m_ptr));
09926     }
09927     return(OK);
09928   }


09931   /*===========================================================================*
09932    *                              m_geometry                                   *
09933    *===========================================================================*/
09934   PRIVATE void m_geometry(entry)
09935   struct partition *entry;
09936   {
09937     /* Memory devices don't have a geometry, but the outside world insists. */
09938     entry->cylinders = (m_geom[m_device].dv_size >> SECTOR_SHIFT) / (64 * 32);
09939     entry->heads = 64;
09940     entry->sectors = 32;
09941   }
```



```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/kernel/wini.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

10000   /*      wini.c - choose a winchester driver.            Author: Kees J. Bot
10001    *                                                             28 May 1994
10002    * Several different winchester drivers may be compiled
10003    * into the kernel, but only one may run.  That one is chosen here using
10004    * the boot variable 'hd'.
10005    */
10006
10007   #include "kernel.h"
10008   #include "driver.h"
10009
10010   #if ENABLE_WINI
10011
10012   /* Map driver name to task function. */
10013   struct hdmap {
10014     char          *name;
10015     task_t        *task;
10016   } hdmap[] = {
10017
10018   #if ENABLE_AT_WINI
10019     { "at",       at_winchester_task      },
10020   #endif
10021
10022   #if ENABLE_BIOS_WINI
10023     { "bios",     bios_winchester_task    },
10024   #endif
10025
10026   #if ENABLE_ESDI_WINI
10027     { "esdi",     esdi_winchester_task    },
10028   #endif
10029
10030   #if ENABLE_XT_WINI
10031     { "xt",       xt_winchester_task      },
10032   #endif
10033
10034   };
10035
10036
10037   /*===========================================================================*
10038    *                              winchester_task                              *
10039    *===========================================================================*/
10040   PUBLIC void winchester_task()
10041   {
10042     /* Call the default or selected winchester task. */
10043     char *hd;
10044     struct hdmap *map;
10045
10046     hd = k_getenv("hd");
10047
10048     for (map = hdmap; map < hdmap + sizeof(hdmap)/sizeof(hdmap[0]); map++) {
10049           if (hd == NULL || strcmp(hd, map->name) == 0) {
10050                   /* Run the selected winchester task. */
10051                   (*map->task)();
10052           }
10053     }
10054     panic("no hd driver", NO_NUM);
```

```
10055    }
10056    #endif /* ENABLE_WINI */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/kernel/at_wini.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
10100    /* This file contains the device dependent part of a driver for the IBM-AT
10101     * winchester controller.
10102     * It was written by Adri Koppes.
10103     *
10104     * The file contains one entry point:
10105     *
10106     *   at_winchester_task:        main entry when system is brought up
10107     *
10108     *
10109     * Changes:
10110     *      13 Apr 1992 by Kees J. Bot: device dependent/independent split.
10111     */
10112
10113    #include "kernel.h"
10114    #include "driver.h"
10115    #include "drvlib.h"
10116
10117    #if ENABLE_AT_WINI
10118
10119    /* I/O Ports used by winchester disk controllers. */
10120
10121    /* Read and write registers */
10122    #define REG_BASE0       0x1F0   /* base register of controller 0 */
10123    #define REG_BASE1       0x170   /* base register of controller 1 */
10124    #define REG_DATA             0  /* data register (offset from the base reg.) */
10125    #define REG_PRECOMP          1  /* start of write precompensation */
10126    #define REG_COUNT            2  /* sectors to transfer */
10127    #define REG_SECTOR           3  /* sector number */
10128    #define REG_CYL_LO           4  /* low byte of cylinder number */
10129    #define REG_CYL_HI           5  /* high byte of cylinder number */
10130    #define REG_LDH              6  /* lba, drive and head */
10131    #define   LDH_DEFAULT           0xA0    /* ECC enable, 512 bytes per sector */
10132    #define   LDH_LBA               0x40    /* Use LBA addressing */
10133    #define   ldh_init(drive)       (LDH_DEFAULT | ((drive) << 4))
10134
10135    /* Read only registers */
10136    #define REG_STATUS           7  /* status */
10137    #define   STATUS_BSY            0x80    /* controller busy */
10138    #define   STATUS_RDY            0x40    /* drive ready */
10139    #define   STATUS_WF             0x20    /* write fault */
10140    #define   STATUS_SC             0x10    /* seek complete (obsolete) */
10141    #define   STATUS_DRQ            0x08    /* data transfer request */
10142    #define   STATUS_CRD            0x04    /* corrected data */
10143    #define   STATUS_IDX            0x02    /* index pulse */
10144    #define   STATUS_ERR            0x01    /* error */
10145    #define REG_ERROR            1  /* error code */
10146    #define   ERROR_BB              0x80    /* bad block */
10147    #define   ERROR_ECC             0x40    /* bad ecc bytes */
10148    #define   ERROR_ID              0x10    /* id not found */
10149    #define   ERROR_AC              0x04    /* aborted command */
```



```
10150    #define   ERROR_TK              0x02    /* track zero error */
10151    #define   ERROR_DM              0x01    /* no data address mark */
10152
10153    /* Write only registers */
10154    #define REG_COMMAND          7  /* command */
10155    #define   CMD_IDLE              0x00    /* for w_command: drive idle */
10156    #define   CMD_RECALIBRATE       0x10    /* recalibrate drive */
10157    #define   CMD_READ              0x20    /* read data */
10158    #define   CMD_WRITE             0x30    /* write data */
10159    #define   CMD_READVERIFY        0x40    /* read verify */
10160    #define   CMD_FORMAT            0x50    /* format track */
10161    #define   CMD_SEEK              0x70    /* seek cylinder */
10162    #define   CMD_DIAG              0x90    /* execute device diagnostics */
10163    #define   CMD_SPECIFY           0x91    /* specify parameters */
10164    #define   ATA_IDENTIFY          0xEC    /* identify drive */
10165    #define REG_CTL         0x206   /* control register */
10166    #define   CTL_NORETRY           0x80    /* disable access retry */
10167    #define   CTL_NOECC             0x40    /* disable ecc retry */
10168    #define   CTL_EIGHTHEADS        0x08    /* more than eight heads */
10169    #define   CTL_RESET             0x04    /* reset controller */
10170    #define   CTL_INTDISABLE        0x02    /* disable interrupts */
10171
10172    /* Interrupt request lines. */
10173    #define AT_IRQ0         14      /* interrupt number for controller 0 */
10174    #define AT_IRQ1         15      /* interrupt number for controller 1 */
10175
10176    /* Common command block */
10177    struct command {
10178      u8_t  precomp;        /* REG_PRECOMP, etc. */
10179      u8_t  count;
10180      u8_t  sector;
10181      u8_t  cyl_lo;
10182      u8_t  cyl_hi;
10183      u8_t  ldh;
10184      u8_t  command;
10185    };
10186
10187
10188    /* Error codes */
10189    #define ERR              (-1)   /* general error */
10190    #define ERR_BAD_SECTOR   (-2)   /* block marked bad detected */
10191
10192    /* Some controllers don't interrupt, the clock will wake us up. */
10193    #define WAKEUP          (32*HZ) /* drive may be out for 31 seconds max */
10194
10195    /* Miscellaneous. */
10196    #define MAX_DRIVES         4    /* this driver supports 4 drives (hd0 - hd19) */
10197    #if _WORD_SIZE > 2
10198    #define MAX_SECS         256    /* controller can transfer this many sectors */
10199    #else
10200    #define MAX_SECS         127    /* but not to a 16 bit process */
10201    #endif
10202    #define MAX_ERRORS         4    /* how often to try rd/wt before quitting */
10203    #define NR_DEVICES      (MAX_DRIVES * DEV_PER_DRIVE)
10204    #define SUB_PER_DRIVE   (NR_PARTITIONS * NR_PARTITIONS)
10205    #define NR_SUBDEVS      (MAX_DRIVES * SUB_PER_DRIVE)
10206    #define TIMEOUT        32000    /* controller timeout in ms */
10207    #define RECOVERYTIME     500    /* controller recovery time in ms */
10208    #define INITIALIZED     0x01    /* drive is initialized */
10209    #define DEAF            0x02    /* controller must be reset */
```

```
10210 #define SMART           0x04    /* drive supports ATA commands */
10211
10212
10213 /* Variables. */
10214 PRIVATE struct wini {           /* main drive struct, one entry per drive */
10215   unsigned state;               /* drive state: deaf, initialized, dead */
10216   unsigned base;                /* base register of the register file */
10217   unsigned irq;                 /* interrupt request line */
10218   unsigned lcylinders;          /* logical number of cylinders (BIOS) */
10219   unsigned lheads;              /* logical number of heads */
10220   unsigned lsectors;            /* logical number of sectors per track */
10221   unsigned pcylinders;          /* physical number of cylinders (translated) */
10222   unsigned pheads;              /* physical number of heads */
10223   unsigned psectors;            /* physical number of sectors per track */
10224   unsigned ldhpref;             /* top four bytes of the LDH (head) register */
10225   unsigned precomp;             /* write precompensation cylinder / 4 */
10226   unsigned max_count;           /* max request for this drive */
10227   unsigned open_ct;             /* in-use count */
10228   struct device part[DEV_PER_DRIVE];    /* primary partitions: hd[0-4] */
10229   struct device subpart[SUB_PER_DRIVE]; /* subpartitions: hd[1-4][a-d] */
10230 } wini[MAX_DRIVES], *w_wn;
10231
10232 PRIVATE struct trans {
10233   struct iorequest_s *iop;      /* belongs to this I/O request */
10234   unsigned long block;          /* first sector to transfer */
10235   unsigned count;               /* byte count */
10236   phys_bytes phys;              /* user physical address */
10237 } wtrans[NR_IOREQS];
10238
10239 PRIVATE struct trans *w_tp;             /* to add transfer requests */
10240 PRIVATE unsigned w_count;               /* number of bytes to transfer */
10241 PRIVATE unsigned long w_nextblock;      /* next block on disk to transfer */
10242 PRIVATE int w_opcode;                   /* DEV_READ or DEV_WRITE */
10243 PRIVATE int w_command;                  /* current command in execution */
10244 PRIVATE int w_status;                   /* status after interrupt */
10245 PRIVATE int w_drive;                    /* selected drive */
10246 PRIVATE struct device *w_dv;            /* device's base and size */
10247
10248 FORWARD _PROTOTYPE( void init_params, (void) );
10249 FORWARD _PROTOTYPE( int w_do_open, (struct driver *dp, message *m_ptr) );
10250 FORWARD _PROTOTYPE( struct device *w_prepare, (int device) );
10251 FORWARD _PROTOTYPE( int w_identify, (void) );
10252 FORWARD _PROTOTYPE( char *w_name, (void) );
10253 FORWARD _PROTOTYPE( int w_specify, (void) );
10254 FORWARD _PROTOTYPE( int w_schedule, (int proc_nr, struct iorequest_s *iop) );
10255 FORWARD _PROTOTYPE( int w_finish, (void) );
10256 FORWARD _PROTOTYPE( int com_out, (struct command *cmd) );
10257 FORWARD _PROTOTYPE( void w_need_reset, (void) );
10258 FORWARD _PROTOTYPE( int w_do_close, (struct driver *dp, message *m_ptr) );
10259 FORWARD _PROTOTYPE( int com_simple, (struct command *cmd) );
10260 FORWARD _PROTOTYPE( void w_timeout, (void) );
10261 FORWARD _PROTOTYPE( int w_reset, (void) );
10262 FORWARD _PROTOTYPE( int w_intr_wait, (void) );
10263 FORWARD _PROTOTYPE( int w_waitfor, (int mask, int value) );
10264 FORWARD _PROTOTYPE( int w_handler, (int irq) );
10265 FORWARD _PROTOTYPE( void w_geometry, (struct partition *entry) );
10266
10267 /* w_waitfor loop unrolled once for speed. */
10268 #define waitfor(mask, value)    \
10269         ((in_byte(w_wn->base + REG_STATUS) & mask) == value \
```



```
10270                 || w_waitfor(mask, value))
10271
10272
10273 /* Entry points to this driver. */
10274 PRIVATE struct driver w_dtab = {
10275   w_name,               /* current device's name */
10276   w_do_open,            /* open or mount request, initialize device */
10277   w_do_close,           /* release device */
10278   do_diocntl,           /* get or set a partition's geometry */
10279   w_prepare,            /* prepare for I/O on a given minor device */
10280   w_schedule,           /* precompute cylinder, head, sector, etc. */
10281   w_finish,             /* do the I/O */
10282   nop_cleanup,          /* nothing to clean up */
10283   w_geometry,           /* tell the geometry of the disk */
10284 };
10285
10286 #if ENABLE_ATAPI
10287 #include "atapi.c"      /* extra code for ATAPI CD-ROM */
10288 #endif
10289
10290
10291 /*===========================================================================*
10292  *                              at_winchester_task                           *
10293  *===========================================================================*/
10294 PUBLIC void at_winchester_task()
10295 {
10296 /* Set special disk parameters then call the generic main loop. */
10297
10298   init_params();
10299
10300   driver_task(&w_dtab);
10301 }
10302
10303
10304 /*===========================================================================*
10305  *                              init_params                                  *
10306  *===========================================================================*/
10307 PRIVATE void init_params()
10308 {
10309 /* This routine is called at startup to initialize the drive parameters. */
10310
10311   u16_t parv[2];
10312   unsigned int vector;
10313   int drive, nr_drives, i;
10314   struct wini *wn;
10315   u8_t params[16];
10316   phys_bytes param_phys = vir2phys(params);
10317
10318   /* Get the number of drives from the BIOS data area */
10319   phys_copy(0x475L, param_phys, 1L);
10320   if ((nr_drives = params[0]) > 2) nr_drives = 2;
10321
10322   for (drive = 0, wn = wini; drive < MAX_DRIVES; drive++, wn++) {
10323         if (drive < nr_drives) {
10324                 /* Copy the BIOS parameter vector */
10325                 vector = drive == 0 ? WINI_0_PARM_VEC : WINI_1_PARM_VEC;
10326                 phys_copy(vector * 4L, vir2phys(parv), 4L);
10327
10328                 /* Calculate the address of the parameters and copy them */
10329                 phys_copy(hclick_to_physb(parv[1]) + parv[0], param_phys, 16L);
```

```
10330
10331                   /* Copy the parameters to the structures of the drive */
10332                   wn->lcylinders = bp_cylinders(params);
10333                   wn->lheads = bp_heads(params);
10334                   wn->lsectors = bp_sectors(params);
10335                   wn->precomp = bp_precomp(params) >> 2;
10336           }
10337           wn->ldhpref = ldh_init(drive);
10338           wn->max_count = MAX_SECS << SECTOR_SHIFT;
10339           if (drive < 2) {
10340                   /* Controller 0. */
10341                   wn->base = REG_BASE0;
10342                   wn->irq = AT_IRQ0;
10343           } else {
10344                   /* Controller 1. */
10345                   wn->base = REG_BASE1;
10346                   wn->irq = AT_IRQ1;
10347           }
10348     }
10349   }


10352   /*============================================================================*
10353    *                              w_do_open                                     *
10354    *============================================================================*/
10355   PRIVATE int w_do_open(dp, m_ptr)
10356   struct driver *dp;
10357   message *m_ptr;
10358   {
10359   /* Device open: Initialize the controller and read the partition table. */
10360
10361     int r;
10362     struct wini *wn;
10363     struct command cmd;
10364
10365     if (w_prepare(m_ptr->DEVICE) == NIL_DEV) return(ENXIO);
10366     wn = w_wn;
10367
10368     if (wn->state == 0) {
10369           /* Try to identify the device. */
10370           if (w_identify() != OK) {
10371                   printf("%s: probe failed\n", w_name());
10372                   if (wn->state & DEAF) w_reset();
10373                   wn->state = 0;
10374                   return(ENXIO);
10375           }
10376     }
10377     if (wn->open_ct++ == 0) {
10378           /* Partition the disk. */
10379           partition(&w_dtab, w_drive * DEV_PER_DRIVE, P_PRIMARY);
10380     }
10381     return(OK);
10382   }


10385   /*============================================================================*
10386    *                              w_prepare                                     *
10387    *============================================================================*/
10388   PRIVATE struct device *w_prepare(device)
10389   int device;
```



```
10390   {
10391   /* Prepare for I/O on a device. */
10392
10393     /* Nothing to transfer as yet. */
10394     w_count = 0;
10395
10396     if (device < NR_DEVICES) {                  /* hd0, hd1, ... */
10397           w_drive = device / DEV_PER_DRIVE;     /* save drive number */
10398           w_wn = &wini[w_drive];
10399           w_dv = &w_wn->part[device % DEV_PER_DRIVE];
10400     } else
10401     if ((unsigned) (device -= MINOR_hd1a) < NR_SUBDEVS) { /* hd1a, hd1b, ... */
10402           w_drive = device / SUB_PER_DRIVE;
10403           w_wn = &wini[w_drive];
10404           w_dv = &w_wn->subpart[device % SUB_PER_DRIVE];
10405     } else {
10406           return(NIL_DEV);
10407     }
10408     return(w_dv);
10409   }


10412   /*============================================================================*
10413    *                              w_identify                                    *
10414    *============================================================================*/
10415   PRIVATE int w_identify()
10416   {
10417   /* Find out if a device exists, if it is an old AT disk, or a newer ATA
10418    * drive, a removable media device, etc.
10419    */
10420
10421     struct wini *wn = w_wn;
10422     struct command cmd;
10423     char id_string[40];
10424     int i, r;
10425     unsigned long size;
10426   #define id_byte(n)      (&tmp_buf[2 * (n)])
10427   #define id_word(n)      (((u16_t) id_byte(n)[0] <<  0) \
10428                           |((u16_t) id_byte(n)[1] <<  8))
10429   #define id_longword(n)  (((u32_t) id_byte(n)[0] <<  0) \
10430                           |((u32_t) id_byte(n)[1] <<  8) \
10431                           |((u32_t) id_byte(n)[2] << 16) \
10432                           |((u32_t) id_byte(n)[3] << 24))
10433
10434     /* Check if the one of the registers exists. */
10435     r = in_byte(wn->base + REG_CYL_LO);
10436     out_byte(wn->base + REG_CYL_LO, ~r);
10437     if (in_byte(wn->base + REG_CYL_LO) == r) return(ERR);
10438
10439     /* Looks OK; register IRQ and try an ATA identify command. */
10440     put_irq_handler(wn->irq, w_handler);
10441     enable_irq(wn->irq);
10442
10443     cmd.ldh     = wn->ldhpref;
10444     cmd.command = ATA_IDENTIFY;
10445     if (com_simple(&cmd) == OK) {
10446           /* This is an ATA device. */
10447           wn->state |= SMART;
10448
10449           /* Device information. */
```

```
10450           port_read(wn->base + REG_DATA, tmp_phys, SECTOR_SIZE);
10451
10452           /* Why are the strings byte swapped??? */
10453           for (i = 0; i < 40; i++) id_string[i] = id_byte(27)[i^1];
10454
10455           /* Preferred CHS translation mode. */
10456           wn->pcylinders = id_word(1);
10457           wn->pheads = id_word(3);
10458           wn->psectors = id_word(6);
10459           size = (u32_t) wn->pcylinders * wn->pheads * wn->psectors;
10460
10461           if ((id_byte(49)[1] & 0x02) && size > 512L*1024*2) {
10462                   /* Drive is LBA capable and is big enough to trust it to
10463                    * not make a mess of it.
10464                    */
10465                   wn->ldhpref |= LDH_LBA;
10466                   size = id_longword(60);
10467           }
10468
10469           if (wn->lcylinders == 0) {
10470                   /* No BIOS parameters?  Then make some up. */
10471                   wn->lcylinders = wn->pcylinders;
10472                   wn->lheads = wn->pheads;
10473                   wn->lsectors = wn->psectors;
10474                   while (wn->lcylinders > 1024) {
10475                           wn->lheads *= 2;
10476                           wn->lcylinders /= 2;
10477                   }
10478           }
10479   } else {
10480           /* Not an ATA device; no translations, no special features.  Don't
10481            * touch it unless the BIOS knows about it.
10482            */
10483           if (wn->lcylinders == 0) return(ERR);   /* no BIOS parameters */
10484           wn->pcylinders = wn->lcylinders;
10485           wn->pheads = wn->lheads;
10486           wn->psectors = wn->lsectors;
10487           size = (u32_t) wn->pcylinders * wn->pheads * wn->psectors;
10488   }
10489   /* The fun ends at 4 GB. */
10490   if (size > ((u32_t) -1) / SECTOR_SIZE) size = ((u32_t) -1) / SECTOR_SIZE;
10491
10492   /* Base and size of the whole drive */
10493   wn->part[0].dv_base = 0;
10494   wn->part[0].dv_size = size << SECTOR_SHIFT;
10495
10496   if (w_specify() != OK && w_specify() != OK) return(ERR);
10497
10498   printf("%s: ", w_name());
10499   if (wn->state & SMART) {
10500         printf("%.40s\n", id_string);
10501   } else {
10502         printf("%ux%ux%u\n", wn->pcylinders, wn->pheads, wn->psectors);
10503   }
10504   return(OK);
10505 }


10508 /*============================================================================*
10509  *                              w_name                                        *
```



```
10510  *============================================================================*/
10511 PRIVATE char *w_name()
10512 {
10513 /* Return a name for the current device. */
10514   static char name[] = "at-hd15";
10515   unsigned device = w_drive * DEV_PER_DRIVE;
10516
10517   if (device < 10) {
10518         name[5] = '0' + device;
10519         name[6] = 0;
10520   } else {
10521         name[5] = '0' + device / 10;
10522         name[6] = '0' + device % 10;
10523   }
10524   return name;
10525 }


10528 /*============================================================================*
10529  *                              w_specify                                     *
10530  *============================================================================*/
10531 PRIVATE int w_specify()
10532 {
10533 /* Routine to initialize the drive after boot or when a reset is needed. */
10534
10535   struct wini *wn = w_wn;
10536   struct command cmd;
10537
10538   if ((wn->state & DEAF) && w_reset() != OK) return(ERR);
10539
10540   /* Specify parameters: precompensation, number of heads and sectors. */
10541   cmd.precomp = wn->precomp;
10542   cmd.count   = wn->psectors;
10543   cmd.ldh     = w_wn->ldhpref | (wn->pheads - 1);
10544   cmd.command = CMD_SPECIFY;            /* Specify some parameters */
10545
10546   if (com_simple(&cmd) != OK) return(ERR);
10547
10548   if (!(wn->state & SMART)) {
10549         /* Calibrate an old disk. */
10550         cmd.sector  = 0;
10551         cmd.cyl_lo  = 0;
10552         cmd.cyl_hi  = 0;
10553         cmd.ldh     = w_wn->ldhpref;
10554         cmd.command = CMD_RECALIBRATE;
10555
10556         if (com_simple(&cmd) != OK) return(ERR);
10557   }
10558
10559   wn->state |= INITIALIZED;
10560   return(OK);
10561 }


10564 /*============================================================================*
10565  *                              w_schedule                                    *
10566  *============================================================================*/
10567 PRIVATE int w_schedule(proc_nr, iop)
10568 int proc_nr;                    /* process doing the request */
10569 struct iorequest_s *iop;        /* pointer to read or write request */
```

```
10570   {
10571   /* Gather I/O requests on consecutive blocks so they may be read/written
10572    * in one controller command.  (There is enough time to compute the next
10573    * consecutive request while an unwanted block passes by.)
10574    */
10575     struct wini *wn = w_wn;
10576     int r, opcode;
10577     unsigned long pos;
10578     unsigned nbytes, count;
10579     unsigned long block;
10580     phys_bytes user_phys;
10581
10582     /* This many bytes to read/write */
10583     nbytes = iop->io_nbytes;
10584     if ((nbytes & SECTOR_MASK) != 0) return(iop->io_nbytes = EINVAL);
10585
10586     /* From/to this position on the device */
10587     pos = iop->io_position;
10588     if ((pos & SECTOR_MASK) != 0) return(iop->io_nbytes = EINVAL);
10589
10590     /* To/from this user address */
10591     user_phys = numap(proc_nr, (vir_bytes) iop->io_buf, nbytes);
10592     if (user_phys == 0) return(iop->io_nbytes = EINVAL);
10593
10594     /* Read or write? */
10595     opcode = iop->io_request & ~OPTIONAL_IO;
10596
10597     /* Which block on disk and how close to EOF? */
10598     if (pos >= w_dv->dv_size) return(OK);          /* At EOF */
10599     if (pos + nbytes > w_dv->dv_size) nbytes = w_dv->dv_size - pos;
10600     block = (w_dv->dv_base + pos) >> SECTOR_SHIFT;
10601
10602     if (w_count > 0 && block != w_nextblock) {
10603           /* This new request can't be chained to the job being built */
10604           if ((r = w_finish()) != OK) return(r);
10605     }
10606
10607     /* The next consecutive block */
10608     w_nextblock = block + (nbytes >> SECTOR_SHIFT);
10609
10610     /* While there are "unscheduled" bytes in the request: */
10611     do {
10612           count = nbytes;
10613
10614           if (w_count == wn->max_count) {
10615                   /* The drive can't do more then max_count at once */
10616                   if ((r = w_finish()) != OK) return(r);
10617           }
10618
10619           if (w_count + count > wn->max_count)
10620                   count = wn->max_count - w_count;
10621
10622           if (w_count == 0) {
10623                   /* The first request in a row, initialize. */
10624                   w_opcode = opcode;
10625                   w_tp = wtrans;
10626           }
10627
10628           /* Store I/O parameters */
10629           w_tp->iop = iop;
```



```
10630           w_tp->block = block;
10631           w_tp->count = count;
10632           w_tp->phys = user_phys;
10633
10634           /* Update counters */
10635           w_tp++;
10636           w_count += count;
10637           block += count >> SECTOR_SHIFT;
10638           user_phys += count;
10639           nbytes -= count;
10640     } while (nbytes > 0);
10641
10642     return(OK);
10643   }


10646   /*============================================================================*
10647    *                              w_finish                                      *
10648    *============================================================================*/
10649   PRIVATE int w_finish()
10650   {
10651   /* Carry out the I/O requests gathered in wtrans[]. */
10652
10653     struct trans *tp = wtrans;
10654     struct wini *wn = w_wn;
10655     int r, errors;
10656     struct command cmd;
10657     unsigned cylinder, head, sector, secspcyl;
10658
10659     if (w_count == 0) return(OK); /* Spurious finish. */
10660
10661     r = ERR;      /* Trigger the first com_out */
10662     errors = 0;
10663
10664     do {
10665           if (r != OK) {
10666                   /* The controller must be (re)programmed. */
10667
10668                   /* First check to see if a reinitialization is needed. */
10669                   if (!(wn->state & INITIALIZED) && w_specify() != OK)
10670                           return(tp->iop->io_nbytes = EIO);
10671
10672                   /* Tell the controller to transfer w_count bytes */
10673                   cmd.precomp = wn->precomp;
10674                   cmd.count   = (w_count >> SECTOR_SHIFT) & BYTE;
10675                   if (wn->ldhpref & LDH_LBA) {
10676                           cmd.sector  = (tp->block >>  0) & 0xFF;
10677                           cmd.cyl_lo  = (tp->block >>  8) & 0xFF;
10678                           cmd.cyl_hi  = (tp->block >> 16) & 0xFF;
10679                           cmd.ldh     = wn->ldhpref | ((tp->block >> 24) & 0xF);
10680                   } else {
10681                           secspcyl = wn->pheads * wn->psectors;
10682                           cylinder = tp->block / secspcyl;
10683                           head = (tp->block % secspcyl) / wn->psectors;
10684                           sector = tp->block % wn->psectors;
10685                           cmd.sector  = sector + 1;
10686                           cmd.cyl_lo  = cylinder & BYTE;
10687                           cmd.cyl_hi  = (cylinder >> 8) & BYTE;
10688                           cmd.ldh     = wn->ldhpref | head;
10689                   }
```

```
10690                   cmd.command = w_opcode == DEV_WRITE ? CMD_WRITE : CMD_READ;
10691
10692                   if ((r = com_out(&cmd)) != OK) {
10693                           if (++errors == MAX_ERRORS) {
10694                                   w_command = CMD_IDLE;
10695                                   return(tp->iop->io_nbytes = EIO);
10696                           }
10697                           continue;       /* Retry */
10698                   }
10699           }
10700
10701           /* For each sector, wait for an interrupt and fetch the data (read),
10702            * or supply data to the controller and wait for an interrupt (write).
10703            */
10704
10705           if (w_opcode == DEV_READ) {
10706                   if ((r = w_intr_wait()) == OK) {
10707                           /* Copy data from the device's buffer to user space. */
10708
10709                           port_read(wn->base + REG_DATA, tp->phys, SECTOR_SIZE);
10710
10711                           tp->phys += SECTOR_SIZE;
10712                           tp->iop->io_nbytes -= SECTOR_SIZE;
10713                           w_count -= SECTOR_SIZE;
10714                           if ((tp->count -= SECTOR_SIZE) == 0) tp++;
10715                   } else {
10716                           /* Any faulty data? */
10717                           if (w_status & STATUS_DRQ) {
10718                                   port_read(wn->base + REG_DATA, tmp_phys,
10719                                                                   SECTOR_SIZE);
10720                           }
10721                   }
10722           } else {
10723                   /* Wait for data requested. */
10724                   if (!waitfor(STATUS_DRQ, STATUS_DRQ)) {
10725                           r = ERR;
10726                   } else {
10727                           /* Fill the buffer of the drive. */
10728
10729                           port_write(wn->base + REG_DATA, tp->phys, SECTOR_SIZE);
10730                           r = w_intr_wait();
10731                   }
10732
10733                   if (r == OK) {
10734                           /* Book the bytes successfully written. */
10735
10736                           tp->phys += SECTOR_SIZE;
10737                           tp->iop->io_nbytes -= SECTOR_SIZE;
10738                           w_count -= SECTOR_SIZE;
10739                           if ((tp->count -= SECTOR_SIZE) == 0) tp++;
10740                   }
10741           }
10742
10743           if (r != OK) {
10744                   /* Don't retry if sector marked bad or too many errors */
10745                   if (r == ERR_BAD_SECTOR || ++errors == MAX_ERRORS) {
10746                           w_command = CMD_IDLE;
10747                           return(tp->iop->io_nbytes = EIO);
10748                   }
10749
```



```
10750                   /* Reset if halfway, but bail out if optional I/O. */
10751                   if (errors == MAX_ERRORS / 2) {
10752                           w_need_reset();
10753                           if (tp->iop->io_request & OPTIONAL_IO) {
10754                                   w_command = CMD_IDLE;
10755                                   return(tp->iop->io_nbytes = EIO);
10756                           }
10757                   }
10758                   continue;       /* Retry */
10759           }
10760           errors = 0;
10761     } while (w_count > 0);
10762
10763     w_command = CMD_IDLE;
10764     return(OK);
10765   }


10768   /*===========================================================================*
10769    *                              com_out                                      *
10770    *===========================================================================*/
10771   PRIVATE int com_out(cmd)
10772   struct command *cmd;            /* Command block */
10773   {
10774   /* Output the command block to the winchester controller and return status */
10775
10776     struct wini *wn = w_wn;
10777     unsigned base = wn->base;
10778
10779     if (!waitfor(STATUS_BSY, 0)) {
10780           printf("%s: controller not ready\n", w_name());
10781           return(ERR);
10782     }
10783
10784     /* Select drive. */
10785     out_byte(base + REG_LDH, cmd->ldh);
10786
10787     if (!waitfor(STATUS_BSY, 0)) {
10788           printf("%s: drive not ready\n", w_name());
10789           return(ERR);
10790     }
10791
10792     /* Schedule a wakeup call, some controllers are flaky. */
10793     clock_mess(WAKEUP, w_timeout);
10794
10795     out_byte(base + REG_CTL, wn->pheads >= 8 ? CTL_EIGHTHEADS : 0);
10796     out_byte(base + REG_PRECOMP, cmd->precomp);
10797     out_byte(base + REG_COUNT, cmd->count);
10798     out_byte(base + REG_SECTOR, cmd->sector);
10799     out_byte(base + REG_CYL_LO, cmd->cyl_lo);
10800     out_byte(base + REG_CYL_HI, cmd->cyl_hi);
10801     lock();
10802     out_byte(base + REG_COMMAND, cmd->command);
10803     w_command = cmd->command;
10804     w_status = STATUS_BSY;
10805     unlock();
10806     return(OK);
10807   }
```

```
10810   /*===========================================================================*
10811    *                              w_need_reset                                 *
10812    *===========================================================================*/
10813   PRIVATE void w_need_reset()
10814   {
10815   /* The controller needs to be reset. */
10816     struct wini *wn;
10817
10818     for (wn = wini; wn < &wini[MAX_DRIVES]; wn++) {
10819           wn->state |= DEAF;
10820           wn->state &= ~INITIALIZED;
10821     }
10822   }


10825   /*===========================================================================*
10826    *                              w_do_close                                   *
10827    *===========================================================================*/
10828   PRIVATE int w_do_close(dp, m_ptr)
10829   struct driver *dp;
10830   message *m_ptr;
10831   {
10832   /* Device close: Release a device. */
10833
10834     if (w_prepare(m_ptr->DEVICE) == NIL_DEV) return(ENXIO);
10835     w_wn->open_ct--;
10836     return(OK);
10837   }


10840   /*===========================================================================*
10841    *                              com_simple                                   *
10842    *===========================================================================*/
10843   PRIVATE int com_simple(cmd)
10844   struct command *cmd;            /* Command block */
10845   {
10846   /* A simple controller command, only one interrupt and no data-out phase. */
10847     int r;
10848
10849     if ((r = com_out(cmd)) == OK) r = w_intr_wait();
10850     w_command = CMD_IDLE;
10851     return(r);
10852   }


10855   /*===========================================================================*
10856    *                              w_timeout                                    *
10857    *===========================================================================*/
10858   PRIVATE void w_timeout()
10859   {
10860     struct wini *wn = w_wn;
10861
10862     switch (w_command) {
10863     case CMD_IDLE:
10864           break;          /* fine */
10865     case CMD_READ:
10866     case CMD_WRITE:
10867           /* Impossible, but not on PC's:  The controller does not respond. */
10868
10869           /* Limiting multisector I/O seems to help. */
```



```
10870           if (wn->max_count > 8 * SECTOR_SIZE) {
10871                   wn->max_count = 8 * SECTOR_SIZE;
10872           } else {
10873                   wn->max_count = SECTOR_SIZE;
10874           }
10875           /*FALL THROUGH*/
10876     default:
10877           /* Some other command. */
10878           printf("%s: timeout on command %02x\n", w_name(), w_command);
10879           w_need_reset();
10880           w_status = 0;
10881           interrupt(WINCHESTER);
10882     }
10883   }


10886   /*===========================================================================*
10887    *                              w_reset                                      *
10888    *===========================================================================*/
10889   PRIVATE int w_reset()
10890   {
10891   /* Issue a reset to the controller.  This is done after any catastrophe,
10892    * like the controller refusing to respond.
10893    */
10894
10895     struct wini *wn;
10896     int err;
10897
10898     /* Wait for any internal drive recovery. */
10899     milli_delay(RECOVERYTIME);
10900
10901     /* Strobe reset bit */
10902     out_byte(w_wn->base + REG_CTL, CTL_RESET);
10903     milli_delay(1);
10904     out_byte(w_wn->base + REG_CTL, 0);
10905     milli_delay(1);
10906
10907     /* Wait for controller ready */
10908     if (!w_waitfor(STATUS_BSY | STATUS_RDY, STATUS_RDY)) {
10909           printf("%s: reset failed, drive busy\n", w_name());
10910           return(ERR);
10911     }
10912
10913     /* The error register should be checked now, but some drives mess it up. */
10914
10915     for (wn = wini; wn < &wini[MAX_DRIVES]; wn++) {
10916           if (wn->base == w_wn->base) wn->state &= ~DEAF;
10917     }
10918     return(OK);
10919   }


10922   /*===========================================================================*
10923    *                              w_intr_wait                                  *
10924    *===========================================================================*/
10925   PRIVATE int w_intr_wait()
10926   {
10927   /* Wait for a task completion interrupt and return results. */
10928
10929     message mess;
```

```
10930     int r;
10931
10932     /* Wait for an interrupt that sets w_status to "not busy". */
10933     while (w_status & STATUS_BSY) receive(HARDWARE, &mess);
10934
10935     /* Check status. */
10936     lock();
10937     if ((w_status & (STATUS_BSY | STATUS_RDY | STATUS_WF | STATUS_ERR))
10938                                                     == STATUS_RDY) {
10939           r = OK;
10940           w_status |= STATUS_BSY; /* assume still busy with I/O */
10941     } else
10942     if ((w_status & STATUS_ERR) && (in_byte(w_wn->base + REG_ERROR) & ERROR_BB)) {
10943           r = ERR_BAD_SECTOR;     /* sector marked bad, retries won't help */
10944     } else {
10945           r = ERR;                /* any other error */
10946     }
10947     unlock();
10948     return(r);
10949   }


10952   /*============================================================================*
10953    *                              w_waitfor                                     *
10954    *============================================================================*/
10955   PRIVATE int w_waitfor(mask, value)
10956   int mask;                       /* status mask */
10957   int value;                      /* required status */
10958   {
10959   /* Wait until controller is in the required state.  Return zero on timeout. */
10960
10961     struct milli_state ms;
10962
10963     milli_start(&ms);
10964     do {
10965          if ((in_byte(w_wn->base + REG_STATUS) & mask) == value) return 1;
10966     } while (milli_elapsed(&ms) < TIMEOUT);
10967
10968     w_need_reset();       /* Controller gone deaf. */
10969     return(0);
10970   }


10973   /*============================================================================*
10974    *                              w_handler                                     *
10975    *============================================================================*/
10976   PRIVATE int w_handler(irq)
10977   int irq;
10978   {
10979   /* Disk interrupt, send message to winchester task and reenable interrupts. */
10980
10981     w_status = in_byte(w_wn->base + REG_STATUS);  /* acknowledge interrupt */
10982     interrupt(WINCHESTER);
10983     return 1;
10984   }


10987   /*============================================================================*
10988    *                              w_geometry                                    *
10989    *============================================================================*/
```



```
10990   PRIVATE void w_geometry(entry)
10991   struct partition *entry;
10992   {
10993     entry->cylinders = w_wn->lcylinders;
10994     entry->heads = w_wn->lheads;
10995     entry->sectors = w_wn->lsectors;
10996   }
10997   #endif /* ENABLE_AT_WINI */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/kernel/clock.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

11000   /* This file contains the code and data for the clock task.  The clock task
11001    * accepts six message types:
11002    *
11003    *   HARD_INT:    a clock interrupt has occurred
11004    *   GET_UPTIME:  get the time since boot in ticks
11005    *   GET_TIME:    a process wants the real time in seconds
11006    *   SET_TIME:    a process wants to set the real time in seconds
11007    *   SET_ALARM:   a process wants to be alerted after a specified interval
11008    *   SET_SYN_AL:  set the sync alarm
11009    *
11010    *
11011    * The input message is format m6.  The parameters are as follows:
11012    *
11013    *     m_type    CLOCK_PROC   FUNC    NEW_TIME
11014    * ----------------------------------------------
11015    * | HARD_INT   |          |         |         |
11016    * |------------+----------+---------+---------|
11017    * | GET_UPTIME |          |         |         |
11018    * |------------+----------+---------+---------|
11019    * | GET_TIME   |          |         |         |
11020    * |------------+----------+---------+---------|
11021    * | SET_TIME   |          |         | newtime |
11022    * |------------+----------+---------+---------|
11023    * | SET_ALARM  | proc_nr  |f to call|  delta  |
11024    * |------------+----------+---------+---------|
11025    * | SET_SYN_AL | proc_nr  |         |  delta  |
11026    * ----------------------------------------------
11027    * NEW_TIME, DELTA_CLICKS, and SECONDS_LEFT all refer to the same field in
11028    * the message, depending upon the message type.
11029    *
11030    * Reply messages are of type OK, except in the case of a HARD_INT, to
11031    * which no reply is generated. For the GET_* messages the time is returned
11032    * in the NEW_TIME field, and for the SET_ALARM and SET_SYN_AL the time
11033    * in seconds remaining until the alarm is returned is returned in the same
11034    * field.
11035    *
11036    * When an alarm goes off, if the caller is a user process, a SIGALRM signal
11037    * is sent to it.  If it is a task, a function specified by the caller will
11038    * be invoked.  This function may, for example, send a message, but only if
11039    * it is certain that the task will be blocked when the timer goes off. A
11040    * synchronous alarm sends a message to the synchronous alarm task, which
11041    * in turn can dispatch a message to another server. This is the only way
11042    * to send an alarm to a server, since servers cannot use the function-call
11043    * mechanism available to tasks and servers cannot receive signals.
11044    */
```

```
11045
11046   #include "kernel.h"
11047   #include <signal.h>
11048   #include <minix/callnr.h>
11049   #include <minix/com.h>
11050   #include "proc.h"
11051
11052   /* Constant definitions. */
11053   #define MILLISEC         100    /* how often to call the scheduler (msec) */
11054   #define SCHED_RATE (MILLISEC*HZ/1000)   /* number of ticks per schedule */
11055
11056   /* Clock parameters. */
11057   #define COUNTER_FREQ (2*TIMER_FREQ)     /* counter frequency using sqare wave*/
11058   #define LATCH_COUNT     0x00    /* cc00xxxx, c = channel, x = any */
11059   #define SQUARE_WAVE     0x36    /* ccaammmb, a = access, m = mode, b = BCD */
11060                                   /*   11x11, 11 = LSB then MSB, x11 = sq wave */
11061   #define TIMER_COUNT ((unsigned) (TIMER_FREQ/HZ)) /* initial value for counter*/
11062   #define TIMER_FREQ  1193182L    /* clock frequency for timer in PC and AT */
11063
11064   #define CLOCK_ACK_BIT   0x80    /* PS/2 clock interrupt acknowledge bit */
11065
11066   /* Clock task variables. */
11067   PRIVATE clock_t realtime;       /* real time clock */
11068   PRIVATE time_t boot_time;       /* time in seconds of system boot */
11069   PRIVATE clock_t next_alarm;     /* probable time of next alarm */
11070   PRIVATE message mc;             /* message buffer for both input and output */
11071   PRIVATE int watchdog_proc;      /* contains proc_nr at call of *watch_dog[]*/
11072   PRIVATE watchdog_t watch_dog[NR_TASKS+NR_PROCS];
11073
11074   /* Variables used by both clock task and synchronous alarm task */
11075   PRIVATE int syn_al_alive= TRUE; /* don't wake syn_alrm_task before inited*/
11076   PRIVATE int syn_table[NR_TASKS+NR_PROCS]; /* which tasks get CLOCK_INT*/
11077
11078   /* Variables changed by interrupt handler */
11079   PRIVATE clock_t pending_ticks;  /* ticks seen by low level only */
11080   PRIVATE int sched_ticks = SCHED_RATE;   /* counter: when 0, call scheduler */
11081   PRIVATE struct proc *prev_ptr;  /* last user process run by clock task */
11082
11083   FORWARD _PROTOTYPE( void common_setalarm, (int proc_nr,
11084                   long delta_ticks, watchdog_t fuction) );
11085   FORWARD _PROTOTYPE( void do_clocktick, (void) );
11086   FORWARD _PROTOTYPE( void do_get_time, (void) );
11087   FORWARD _PROTOTYPE( void do_getuptime, (void) );
11088   FORWARD _PROTOTYPE( void do_set_time, (message *m_ptr) );
11089   FORWARD _PROTOTYPE( void do_setalarm, (message *m_ptr) );
11090   FORWARD _PROTOTYPE( void init_clock, (void) );
11091   FORWARD _PROTOTYPE( void cause_alarm, (void) );
11092   FORWARD _PROTOTYPE( void do_setsyn_alrm, (message *m_ptr) );
11093   FORWARD _PROTOTYPE( int clock_handler, (int irq) );
11094
11095   /*===========================================================================*
11096    *                                clock_task                                 *
11097    *===========================================================================*/
11098   PUBLIC void clock_task()
11099   {
11100   /* Main program of clock task.  It corrects realtime by adding pending
11101    * ticks seen only by the interrupt service, then it determines which
11102    * of the 6 possible calls this is by looking at 'mc.m_type'.  Then
11103    * it dispatches.
11104    */
```



```
11105
11106     int opcode;
11107
11108     init_clock();                 /* initialize clock task */
11109
11110     /* Main loop of the clock task.  Get work, process it, sometimes reply. */
11111     while (TRUE) {
11112        receive(ANY, &mc);         /* go get a message */
11113        opcode = mc.m_type;        /* extract the function code */
11114
11115        lock();
11116        realtime += pending_ticks; /* transfer ticks from low level handler */
11117        pending_ticks = 0;         /* so we don't have to worry about them */
11118        unlock();
11119
11120        switch (opcode) {
11121           case HARD_INT:   do_clocktick();        break;
11122           case GET_UPTIME: do_getuptime();        break;
11123           case GET_TIME:   do_get_time();         break;
11124           case SET_TIME:   do_set_time(&mc);      break;
11125           case SET_ALARM:  do_setalarm(&mc);      break;
11126           case SET_SYNC_AL:do_setsyn_alrm(&mc);   break;
11127           default: panic("clock task got bad message", mc.m_type);
11128        }
11129
11130       /* Send reply, except for clock tick. */
11131       mc.m_type = OK;
11132       if (opcode != HARD_INT) send(mc.m_source, &mc);
11133     }
11134   }


11137   /*===========================================================================*
11138    *                                do_clocktick                               *
11139    *===========================================================================*/
11140   PRIVATE void do_clocktick()
11141   {
11142   /* Despite its name, this routine is not called on every clock tick. It
11143    * is called on those clock ticks when a lot of work needs to be done.
11144    */
11145
11146     register struct proc *rp;
11147     register int proc_nr;
11148
11149     if (next_alarm <= realtime) {
11150           /* An alarm may have gone off, but proc may have exited, so check. */
11151           next_alarm = LONG_MAX;  /* start computing next alarm */
11152           for (rp = BEG_PROC_ADDR; rp < END_PROC_ADDR; rp++) {
11153                   if (rp->p_alarm != 0) {
11154                           /* See if this alarm time has been reached. */
11155                           if (rp->p_alarm <= realtime) {
11156                                   /* A timer has gone off.  If it is a user proc,
11157                                    * send it a signal.  If it is a task, call the
11158                                    * function previously specified by the task.
11159                                    */
11160                                   proc_nr = proc_number(rp);
11161                                   if (watch_dog[proc_nr+NR_TASKS]) {
11162                                           watchdog_proc= proc_nr;
11163                                           (*watch_dog[proc_nr+NR_TASKS])();
11164                                   }
```

```
11165                                   else
11166                                           cause_sig(proc_nr, SIGALRM);
11167                                   rp->p_alarm = 0;
11168                           }
11169
11170                           /* Work on determining which alarm is next. */
11171                           if (rp->p_alarm != 0 && rp->p_alarm < next_alarm)
11172                                   next_alarm = rp->p_alarm;
11173                   }
11174           }
11175     }
11176
11177     /* If a user process has been running too long, pick another one. */
11178     if (--sched_ticks == 0) {
11179           if (bill_ptr == prev_ptr) lock_sched(); /* process has run too long */
11180           sched_ticks = SCHED_RATE;               /* reset quantum */
11181           prev_ptr = bill_ptr;                    /* new previous process */
11182     }
11183   }


11186   /*===========================================================================*
11187    *                                do_getuptime                               *
11188    *===========================================================================*/
11189   PRIVATE void do_getuptime()
11190   {
11191   /* Get and return the current clock uptime in ticks. */
11192
11193     mc.NEW_TIME = realtime;       /* current uptime */
11194   }


11197   /*===========================================================================*
11198    *                                get_uptime                                 *
11199    *===========================================================================*/
11200   PUBLIC clock_t get_uptime()
11201   {
11202   /* Get and return the current clock uptime in ticks.  This function is
11203    * designed to be called from other tasks, so they can get uptime without
11204    * the overhead of messages. It has to be careful about pending_ticks.
11205    */
11206
11207     clock_t uptime;
11208
11209     lock();
11210     uptime = realtime + pending_ticks;
11211     unlock();
11212     return(uptime);
11213   }


11216   /*===========================================================================*
11217    *                                do_get_time                                *
11218    *===========================================================================*/
11219   PRIVATE void do_get_time()
11220   {
11221   /* Get and return the current clock time in seconds. */
11222
11223     mc.NEW_TIME = boot_time + realtime/HZ;        /* current real time */
11224   }
```



```
11227   /*===========================================================================*
11228    *                                do_set_time                                *
11229    *===========================================================================*/
11230   PRIVATE void do_set_time(m_ptr)
11231   message *m_ptr;                 /* pointer to request message */
11232   {
11233   /* Set the real time clock.  Only the superuser can use this call. */
11234
11235     boot_time = m_ptr->NEW_TIME - realtime/HZ;
11236   }


11239   /*===========================================================================*
11240    *                                do_setalarm                                *
11241    *===========================================================================*/
11242   PRIVATE void do_setalarm(m_ptr)
11243   message *m_ptr;                 /* pointer to request message */
11244   {
11245   /* A process wants an alarm signal or a task wants a given watch_dog function
11246    * called after a specified interval.
11247    */
11248
11249     register struct proc *rp;
11250     int proc_nr;                  /* which process wants the alarm */
11251     long delta_ticks;             /* in how many clock ticks does he want it? */
11252     watchdog_t function;          /* function to call (tasks only) */
11253
11254     /* Extract the parameters from the message. */
11255     proc_nr = m_ptr->CLOCK_PROC_NR;       /* process to interrupt later */
11256     delta_ticks = m_ptr->DELTA_TICKS;     /* how many ticks to wait */
11257     function = (watchdog_t) m_ptr->FUNC_TO_CALL;
11258                                           /* function to call (tasks only) */
11259     rp = proc_addr(proc_nr);
11260     mc.SECONDS_LEFT = (rp->p_alarm == 0 ? 0 : (rp->p_alarm - realtime)/HZ );
11261     if (!istaskp(rp)) function= 0;        /* user processes get signaled */
11262     common_setalarm(proc_nr, delta_ticks, function);
11263   }


11266   /*===========================================================================*
11267    *                                do_setsyn_alrm                             *
11268    *===========================================================================*/
11269   PRIVATE void do_setsyn_alrm(m_ptr)
11270   message *m_ptr;                 /* pointer to request message */
11271   {
11272   /* A process wants a synchronous alarm.
11273    */
11274
11275     register struct proc *rp;
11276     int proc_nr;                  /* which process wants the alarm */
11277     long delta_ticks;             /* in how many clock ticks does he want it? */
11278
11279     /* Extract the parameters from the message. */
11280     proc_nr = m_ptr->CLOCK_PROC_NR;       /* process to interrupt later */
11281     delta_ticks = m_ptr->DELTA_TICKS;     /* how many ticks to wait */
11282     rp = proc_addr(proc_nr);
11283     mc.SECONDS_LEFT = (rp->p_alarm == 0 ? 0 : (rp->p_alarm - realtime)/HZ );
11284     common_setalarm(proc_nr, delta_ticks, cause_alarm);
```

```
11285   }


11288   /*==========================================================================*
11289    *                              common_setalarm                             *
11290    *==========================================================================*/
11291   PRIVATE void common_setalarm(proc_nr, delta_ticks, function)
11292   int proc_nr;                    /* which process wants the alarm */
11293   long delta_ticks;               /* in how many clock ticks does he want it? */
11294   watchdog_t function;            /* function to call (0 if cause_sig is
11295                                    * to be called */
11296   {
11297   /* Finish up work of do_set_alarm and do_setsyn_alrm.  Record an alarm
11298    * request and check to see if it is the next alarm needed.
11299    */
11300
11301     register struct proc *rp;
11302
11303     rp = proc_addr(proc_nr);
11304     rp->p_alarm = (delta_ticks == 0 ? 0 : realtime + delta_ticks);
11305     watch_dog[proc_nr+NR_TASKS] = function;
11306
11307     /* Which alarm is next? */
11308     next_alarm = LONG_MAX;
11309     for (rp = BEG_PROC_ADDR; rp < END_PROC_ADDR; rp++)
11310           if(rp->p_alarm != 0 && rp->p_alarm < next_alarm)next_alarm=rp->p_alarm;
11311
11312   }


11315   /*==========================================================================*
11316    *                              cause_alarm                                 *
11317    *==========================================================================*/
11318   PRIVATE void cause_alarm()
11319   {
11320   /* Routine called if a timer goes off and the process requested a synchronous
11321    * alarm. The process number is in the global variable watchdog_proc (HACK).
11322    */
11323     message mess;
11324
11325     syn_table[watchdog_proc + NR_TASKS]= TRUE;
11326     if (!syn_al_alive) send (SYN_ALRM_TASK, &mess);
11327   }


11330   /*==========================================================================*
11331    *                              syn_alrm_task                               *
11332    *==========================================================================*/
11333   PUBLIC void syn_alrm_task()
11334   {
11335   /* Main program of the synchronous alarm task.
11336    * This task receives messages only from cause_alarm in the clock task.
11337    * It sends a CLOCK_INT message to a process that requested a syn_alrm.
11338    * Synchronous alarms are so called because, unlike a signals or the
11339    * activation of a watchdog, a synchronous alarm is received by a process
11340    * when it is in a known part of its code, that is, when it has issued
11341    * a call to receive a message.
11342    */
11343
11344     message mess;
```



```
11345     int work_done;        /* ready to sleep ? */
11346     int *al_ptr;          /* pointer in syn_table */
11347     int i;
11348
11349     syn_al_alive= TRUE;
11350     for (i= 0, al_ptr= syn_table; i<NR_TASKS+NR_PROCS; i++, al_ptr++)
11351           *al_ptr= FALSE;
11352
11353     while (TRUE) {
11354           work_done= TRUE;
11355           for (i= 0, al_ptr= syn_table; i<NR_TASKS+NR_PROCS; i++, al_ptr++)
11356                   if (*al_ptr) {
11357                           *al_ptr= FALSE;
11358                           mess.m_type= CLOCK_INT;
11359                           send (i-NR_TASKS, &mess);
11360                           work_done= FALSE;
11361                   }
11362           if (work_done) {
11363                   syn_al_alive= FALSE;
11364                   receive (CLOCK, &mess);
11365                   syn_al_alive= TRUE;
11366           }
11367     }
11368   }


11371   /*==========================================================================*
11372    *                              clock_handler                               *
11373    *==========================================================================*/
11374   PRIVATE int clock_handler(irq)
11375   int irq;
11376   {
11377   /* This executes on every clock tick (i.e., every time the timer chip
11378    * generates an interrupt). It does a little bit of work so the clock
11379    * task does not have to be called on every tick.
11380    *
11381    * Switch context to do_clocktick if an alarm has gone off.
11382    * Also switch there to reschedule if the reschedule will do something.
11383    * This happens when
11384    *      (1) quantum has expired
11385    *      (2) current process received full quantum (as clock sampled it!)
11386    *      (3) something else is ready to run.
11387    * Also call TTY and PRINTER and let them do whatever is necessary.
11388    *
11389    * Many global global and static variables are accessed here.  The safety
11390    * of this must be justified.  Most of them are not changed here:
11391    *      k_reenter:
11392    *              This safely tells if the clock interrupt is nested.
11393    *      proc_ptr, bill_ptr:
11394    *              These are used for accounting.  It does not matter if proc.c
11395    *              is changing them, provided they are always valid pointers,
11396    *              since at worst the previous process would be billed.
11397    *      next_alarm, realtime, sched_ticks, bill_ptr, prev_ptr,
11398    *      rdy_head[USER_Q]:
11399    *              These are tested to decide whether to call interrupt().  It
11400    *              does not matter if the test is sometimes (rarely) backwards
11401    *              due to a race, since this will only delay the high-level
11402    *              processing by one tick, or call the high level unnecessarily.
11403    * The variables which are changed require more care:
11404    *      rp->user_time, rp->sys_time:
```

```
11405     *              These are protected by explicit locks in system.c.  They are
11406     *              not properly protected in dmp.c (the increment here is not
11407     *              atomic) but that hardly matters.
11408     *      pending_ticks:
11409     *              This is protected by explicit locks in clock.c.  Don't
11410     *              update realtime directly, since there are too many
11411     *              references to it to guard conveniently.
11412     *      lost_ticks:
11413     *              Clock ticks counted outside the clock task.
11414     *      sched_ticks, prev_ptr:
11415     *              Updating these competes with similar code in do_clocktick().
11416     *              No lock is necessary, because if bad things happen here
11417     *              (like sched_ticks going negative), the code in do_clocktick()
11418     *              will restore the variables to reasonable values, and an
11419     *              occasional missed or extra sched() is harmless.
11420     *
11421     * Are these complications worth the trouble?  Well, they make the system 15%
11422     * faster on a 5MHz 8088, and make task debugging much easier since there are
11423     * no task switches on an inactive system.
11424     */
11425
11426      register struct proc *rp;
11427      register unsigned ticks;
11428      clock_t now;
11429
11430      if (ps_mca) {
11431            /* Acknowledge the PS/2 clock interrupt. */
11432            out_byte(PORT_B, in_byte(PORT_B) | CLOCK_ACK_BIT);
11433      }
11434
11435      /* Update user and system accounting times.
11436       * First charge the current process for user time.
11437       * If the current process is not the billable process (usually because it
11438       * is a task), charge the billable process for system time as well.
11439       * Thus the unbillable tasks' user time is the billable users' system time.
11440       */
11441      if (k_reenter != 0)
11442            rp = proc_addr(HARDWARE);
11443      else
11444            rp = proc_ptr;
11445      ticks = lost_ticks + 1;
11446      lost_ticks = 0;
11447      rp->user_time += ticks;
11448      if (rp != bill_ptr && rp != proc_addr(IDLE)) bill_ptr->sys_time += ticks;
11449
11450      pending_ticks += ticks;
11451      now = realtime + pending_ticks;
11452      if (tty_timeout <= now) tty_wakeup(now);      /* possibly wake up TTY */
11453      pr_restart();                                 /* possibly restart printer */
11454
11455      if (next_alarm <= now ||
11456          sched_ticks == 1 &&
11457          bill_ptr == prev_ptr &&
11458          rdy_head[USER_Q] != NIL_PROC) {
11459            interrupt(CLOCK);
11460            return 1;       /* Reenable interrupts */
11461      }
11462
11463      if (--sched_ticks == 0) {
11464            /* If bill_ptr == prev_ptr, no ready users so don't need sched(). */
```



```
11465            sched_ticks = SCHED_RATE;       /* reset quantum */
11466            prev_ptr = bill_ptr;            /* new previous process */
11467      }
11468      return 1;     /* Reenable clock interrupt */
11469    }

11471    /*===========================================================================*
11472     *                                init_clock                                 *
11473     *===========================================================================*/
11474    PRIVATE void init_clock()
11475    {
11476    /* Initialize channel 0 of the 8253A timer to e.g. 60 Hz. */
11477
11478      out_byte(TIMER_MODE, SQUARE_WAVE);    /* set timer to run continuously */
11479      out_byte(TIMER0, TIMER_COUNT);        /* load timer low byte */
11480      out_byte(TIMER0, TIMER_COUNT >> 8);   /* load timer high byte */
11481      put_irq_handler(CLOCK_IRQ, clock_handler);    /* set the interrupt handler */
11482      enable_irq(CLOCK_IRQ);                /* ready for clock interrupts */
11483    }


11486    /*===========================================================================*
11487     *                                clock_stop                                 *
11488     *===========================================================================*/
11489    PUBLIC void clock_stop()
11490    {
11491    /* Reset the clock to the BIOS rate. (For rebooting) */
11492
11493      out_byte(TIMER_MODE, 0x36);
11494      out_byte(TIMER0, 0);
11495      out_byte(TIMER0, 0);
11496    }


11499    /*===========================================================================*
11500     *                                milli_delay                                *
11501     *===========================================================================*/
11502    PUBLIC void milli_delay(millisec)
11503    unsigned millisec;
11504    {
11505    /* Delay some milliseconds. */
11506
11507      struct milli_state ms;
11508
11509      milli_start(&ms);
11510      while (milli_elapsed(&ms) < millisec) {}
11511    }

11513    /*===========================================================================*
11514     *                                milli_start                                *
11515     *===========================================================================*/
11516    PUBLIC void milli_start(msp)
11517    struct milli_state *msp;
11518    {
11519    /* Prepare for calls to milli_elapsed(). */
11520
11521      msp->prev_count = 0;
11522      msp->accum_count = 0;
11523    }
```

```
11526   /*===========================================================================*
11527    *                                milli_elapsed                              *
11528    *===========================================================================*/
11529   PUBLIC unsigned milli_elapsed(msp)
11530   struct milli_state *msp;
11531   {
11532   /* Return the number of milliseconds since the call to milli_start().  Must be
11533    * polled rapidly.
11534    */
11535     unsigned count;
11536
11537     /* Read the counter for channel 0 of the 8253A timer.  The counter
11538      * decrements at twice the timer frequency (one full cycle for each
11539      * half of square wave).  The counter normally has a value between 0
11540      * and TIMER_COUNT, but before the clock task has been initialized,
11541      * its maximum value is 65535, as set by the BIOS.
11542      */
11543     out_byte(TIMER_MODE, LATCH_COUNT);    /* make chip copy count to latch */
11544     count = in_byte(TIMER0);      /* countdown continues during 2-step read */
11545     count |= in_byte(TIMER0) << 8;
11546
11547     /* Add difference between previous and new count unless the counter has
11548      * increased (restarted its cycle).  We may lose a tick now and then, but
11549      * microsecond precision is not needed.
11550      */
11551     msp->accum_count += count <= msp->prev_count ? (msp->prev_count - count) : 1;
11552     msp->prev_count = count;
11553
11554     return msp->accum_count / (TIMER_FREQ / 1000);
11555   }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/kernel/tty.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

11600   /*      tty.h - Terminals       */
11601
11602   #define TTY_IN_BYTES     256    /* tty input queue size */
11603   #define TAB_SIZE           8    /* distance between tab stops */
11604   #define TAB_MASK           7    /* mask to compute a tab stop position */
11605
11606   #define ESC             '\33'   /* escape */
11607
11608   #define O_NOCTTY       00400    /* from <fcntl.h>, or cc will choke */
11609   #define O_NONBLOCK     04000
11610
11611   typedef _PROTOTYPE( void (*devfun_t), (struct tty *tp) );
11612   typedef _PROTOTYPE( void (*devfunarg_t), (struct tty *tp, int c) );
11613
11614   typedef struct tty {
11615     int tty_events;               /* set when TTY should inspect this line */
11616
11617     /* Input queue.  Typed characters are stored here until read by a program. */
11618     u16_t *tty_inhead;            /* pointer to place where next char goes */
11619     u16_t *tty_intail;            /* pointer to next char to be given to prog */
```



```
11620     int tty_incount;              /* # chars in the input queue */
11621     int tty_eotct;                /* number of "line breaks" in input queue */
11622     devfun_t tty_devread;         /* routine to read from low level buffers */
11623     devfun_t tty_icancel;         /* cancel any device input */
11624     int tty_min;                  /* minimum requested #chars in input queue */
11625     clock_t tty_time;             /* time when the input is available */
11626     struct tty *tty_timenext;     /* for a list of ttys with active timers */
11627
11628     /* Output section. */
11629     devfun_t tty_devwrite;        /* routine to start actual device output */
11630     devfunarg_t tty_echo;         /* routine to echo characters input */
11631     devfun_t tty_ocancel;         /* cancel any ongoing device output */
11632     devfun_t tty_break;           /* let the device send a break */
11633
11634     /* Terminal parameters and status. */
11635     int tty_position;             /* current position on the screen for echoing */
11636     char tty_reprint;             /* 1 when echoed input messed up, else 0 */
11637     char tty_escaped;             /* 1 when LNEXT (^V) just seen, else 0 */
11638     char tty_inhibited;           /* 1 when STOP (^S) just seen (stops output) */
11639     char tty_pgrp;                /* slot number of controlling process */
11640     char tty_openct;              /* count of number of opens of this tty */
11641
11642     /* Information about incomplete I/O requests is stored here. */
11643     char tty_inrepcode;           /* reply code, TASK_REPLY or REVIVE */
11644     char tty_incaller;            /* process that made the call (usually FS) */
11645     char tty_inproc;              /* process that wants to read from tty */
11646     vir_bytes tty_in_vir;         /* virtual address where data is to go */
11647     int tty_inleft;               /* how many chars are still needed */
11648     int tty_incum;                /* # chars input so far */
11649     char tty_outrepcode;          /* reply code, TASK_REPLY or REVIVE */
11650     char tty_outcaller;           /* process that made the call (usually FS) */
11651     char tty_outproc;             /* process that wants to write to tty */
11652     vir_bytes tty_out_vir;        /* virtual address where data comes from */
11653     int tty_outleft;              /* # chars yet to be output */
11654     int tty_outcum;               /* # chars output so far */
11655     char tty_iocaller;            /* process that made the call (usually FS) */
11656     char tty_ioproc;              /* process that wants to do an ioctl */
11657     int tty_ioreq;                /* ioctl request code */
11658     vir_bytes tty_iovir;          /* virtual address of ioctl buffer */
11659
11660     /* Miscellaneous. */
11661     devfun_t tty_ioctl;           /* set line speed, etc. at the device level */
11662     devfun_t tty_close;           /* tell the device that the tty is closed */
11663     void *tty_priv;               /* pointer to per device private data */
11664     struct termios tty_termios;   /* terminal attributes */
11665     struct winsize tty_winsize;   /* window size (#lines and #columns) */
11666
11667     u16_t tty_inbuf[TTY_IN_BYTES];/* tty input buffer */
11668   } tty_t;
11669
11670   EXTERN tty_t tty_table[NR_CONS+NR_RS_LINES+NR_PTYS];
11671
11672   /* Values for the fields. */
11673   #define NOT_ESCAPED        0    /* previous character is not LNEXT (^V) */
11674   #define ESCAPED            1    /* previous character was LNEXT (^V) */
11675   #define RUNNING            0    /* no STOP (^S) has been typed to stop output */
11676   #define STOPPED            1    /* STOP (^S) has been typed to stop output */
11677
11678   /* Fields and flags on characters in the input queue. */
11679   #define IN_CHAR       0x00FF    /* low 8 bits are the character itself */
```

```
11680    #define IN_LEN        0x0F00    /* length of char if it has been echoed */
11681    #define IN_LSHIFT          8    /* length = (c & IN_LEN) >> IN_LSHIFT */
11682    #define IN_EOT        0x1000    /* char is a line break (^D, LF) */
11683    #define IN_EOF        0x2000    /* char is EOF (^D), do not return to user */
11684    #define IN_ESC        0x4000    /* escaped by LNEXT (^V), no interpretation */
11685
11686    /* Times and timeouts. */
11687    #define TIME_NEVER      ((clock_t) -1 < 0 ? (clock_t) LONG_MAX : (clock_t) -1)
11688    #define force_timeout() ((void) (tty_timeout = 0))
11689
11690    EXTERN tty_t *tty_timelist;     /* list of ttys with active timers */
11691
11692    /* Number of elements and limit of a buffer. */
11693    #define buflen(buf)     (sizeof(buf) / sizeof((buf)[0]))
11694    #define bufend(buf)     ((buf) + buflen(buf))
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/kernel/tty.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

11700    /* This file contains the terminal driver, both for the IBM console and regular
11701     * ASCII terminals.  It handles only the device-independent part of a TTY, the
11702     * device dependent parts are in console.c, rs232.c, etc.  This file contains
11703     * two main entry points, tty_task() and tty_wakeup(), and several minor entry
11704     * points for use by the device-dependent code.
11705     *
11706     * The device-independent part accepts "keyboard" input from the device-
11707     * dependent part, performs input processing (special key interpretation),
11708     * and sends the input to a process reading from the TTY.  Output to a TTY
11709     * is sent to the device-dependent code for output processing and "screen"
11710     * display.  Input processing is done by the device by calling 'in_process'
11711     * on the input characters, output processing may be done by the device itself
11712     * or by calling 'out_process'.  The TTY takes care of input queuing, the
11713     * device does the output queuing.  If a device receives an external signal,
11714     * like an interrupt, then it causes tty_wakeup() to be run by the CLOCK task
11715     * to, you guessed it, wake up the TTY to check if input or output can
11716     * continue.
11717     *
11718     * The valid messages and their parameters are:
11719     *
11720     *   HARD_INT:     output has been completed or input has arrived
11721     *   DEV_READ:     a process wants to read from a terminal
11722     *   DEV_WRITE:    a process wants to write on a terminal
11723     *   DEV_IOCTL:    a process wants to change a terminal's parameters
11724     *   DEV_OPEN:     a tty line has been opened
11725     *   DEV_CLOSE:    a tty line has been closed
11726     *   CANCEL:       terminate a previous incomplete system call immediately
11727     *
11728     *    m_type      TTY_LINE   PROC_NR    COUNT   TTY_SPEK  TTY_FLAGS  ADDRESS
11729     * ---------------------------------------------------------------------------
11730     * | HARD_INT    |         |         |         |         |         |
11731     * |-------------+---------+---------+---------+---------+---------+---------|
11732     * | DEV_READ    |minor dev| proc nr |  count  |         |O_NONBLOCK| buf ptr |
11733     * |-------------+---------+---------+---------+---------+---------+---------|
11734     * | DEV_WRITE   |minor dev| proc nr |  count  |         |         | buf ptr |
```



```
11735     * |-------------+---------+---------+---------+---------+---------+---------|
11736     * | DEV_IOCTL   |minor dev| proc nr |func code|erase etc|  flags  |         |
11737     * |-------------+---------+---------+---------+---------+---------+---------|
11738     * | DEV_OPEN    |minor dev| proc nr | O_NOCTTY|         |         |         |
11739     * |-------------+---------+---------+---------+---------+---------+---------|
11740     * | DEV_CLOSE   |minor dev| proc nr |         |         |         |         |
11741     * |-------------+---------+---------+---------+---------+---------+---------|
11742     * | CANCEL      |minor dev| proc nr |         |         |         |         |
11743     * ---------------------------------------------------------------------------
11744     */
11745
11746    #include "kernel.h"
11747    #include <termios.h>
11748    #include <sys/ioctl.h>
11749    #include <signal.h>
11750    #include <minix/callnr.h>
11751    #include <minix/com.h>
11752    #include <minix/keymap.h>
11753    #include "tty.h"
11754    #include "proc.h"
11755
11756    /* Address of a tty structure. */
11757    #define tty_addr(line)  (&tty_table[line])
11758
11759    /* First minor numbers for the various classes of TTY devices. */
11760    #define CONS_MINOR        0
11761    #define LOG_MINOR        15
11762    #define RS232_MINOR      16
11763    #define TTYPX_MINOR     128
11764    #define PTYPX_MINOR     192
11765
11766    /* Macros for magic tty types. */
11767    #define isconsole(tp)   ((tp) < tty_addr(NR_CONS))
11768
11769    /* Macros for magic tty structure pointers. */
11770    #define FIRST_TTY       tty_addr(0)
11771    #define END_TTY         tty_addr(sizeof(tty_table) / sizeof(tty_table[0]))
11772
11773    /* A device exists if at least its 'devread' function is defined. */
11774    #define tty_active(tp)  ((tp)->tty_devread != NULL)
11775
11776    /* RS232 lines or pseudo terminals can be completely configured out. */
11777    #if NR_RS_LINES == 0
11778    #define rs_init(tp)     ((void) 0)
11779    #endif
11780    #if NR_PTYS == 0
11781    #define pty_init(tp)    ((void) 0)
11782    #define do_pty(tp, mp)  ((void) 0)
11783    #endif
11784
11785    FORWARD _PROTOTYPE( void do_cancel, (tty_t *tp, message *m_ptr)         );
11786    FORWARD _PROTOTYPE( void do_ioctl, (tty_t *tp, message *m_ptr)          );
11787    FORWARD _PROTOTYPE( void do_open, (tty_t *tp, message *m_ptr)           );
11788    FORWARD _PROTOTYPE( void do_close, (tty_t *tp, message *m_ptr)          );
11789    FORWARD _PROTOTYPE( void do_read, (tty_t *tp, message *m_ptr)           );
11790    FORWARD _PROTOTYPE( void do_write, (tty_t *tp, message *m_ptr)          );
11791    FORWARD _PROTOTYPE( void in_transfer, (tty_t *tp)                       );
11792    FORWARD _PROTOTYPE( int echo, (tty_t *tp, int ch)                       );
11793    FORWARD _PROTOTYPE( void rawecho, (tty_t *tp, int ch)                   );
11794    FORWARD _PROTOTYPE( int back_over, (tty_t *tp)                          );
```

```
11795   FORWARD _PROTOTYPE( void reprint, (tty_t *tp)                          );
11796   FORWARD _PROTOTYPE( void dev_ioctl, (tty_t *tp)                        );
11797   FORWARD _PROTOTYPE( void setattr, (tty_t *tp)                          );
11798   FORWARD _PROTOTYPE( void tty_icancel, (tty_t *tp)                      );
11799   FORWARD _PROTOTYPE( void tty_init, (tty_t *tp)                         );
11800   FORWARD _PROTOTYPE( void settimer, (tty_t *tp, int on)                 );
11801
11802   /* Default attributes. */
11803   PRIVATE struct termios termios_defaults = {
11804     TINPUT_DEF, TOUTPUT_DEF, TCTRL_DEF, TLOCAL_DEF, TSPEED_DEF, TSPEED_DEF,
11805     {
11806           TEOF_DEF, TEOL_DEF, TERASE_DEF, TINTR_DEF, TKILL_DEF, TMIN_DEF,
11807           TQUIT_DEF, TTIME_DEF, TSUSP_DEF, TSTART_DEF, TSTOP_DEF,
11808           TREPRINT_DEF, TLNEXT_DEF, TDISCARD_DEF,
11809     },
11810   };
11811   PRIVATE struct winsize winsize_defaults;        /* = all zeroes */
11812
11813
11814   /*===========================================================================*
11815    *                                tty_task                                   *
11816    *===========================================================================*/
11817   PUBLIC void tty_task()
11818   {
11819   /* Main routine of the terminal task. */
11820
11821     message tty_mess;             /* buffer for all incoming messages */
11822     register tty_t *tp;
11823     unsigned line;
11824
11825     /* Initialize the terminal lines. */
11826     for (tp = FIRST_TTY; tp < END_TTY; tp++) tty_init(tp);
11827
11828     /* Display the Minix startup banner. */
11829     printf("Minix %s.%s  Copyright 1997 Prentice-Hall, Inc.\n\n",
11830                                             OS_RELEASE, OS_VERSION);
11831     printf("Executing in 32-bit protected mode\n\n");
11832
11833     while (TRUE) {
11834           /* Handle any events on any of the ttys. */
11835           for (tp = FIRST_TTY; tp < END_TTY; tp++) {
11836                 if (tp->tty_events) handle_events(tp);
11837           }
11838
11839           receive(ANY, &tty_mess);
11840
11841           /* A hardware interrupt is an invitation to check for events. */
11842           if (tty_mess.m_type == HARD_INT) continue;
11843
11844           /* Check the minor device number. */
11845           line = tty_mess.TTY_LINE;
11846           if ((line - CONS_MINOR) < NR_CONS) {
11847                 tp = tty_addr(line - CONS_MINOR);
11848           } else
11849           if (line == LOG_MINOR) {
11850                 tp = tty_addr(0);
11851           } else
11852           if ((line - RS232_MINOR) < NR_RS_LINES) {
11853                 tp = tty_addr(line - RS232_MINOR + NR_CONS);
11854           } else
```



```
11855           if ((line - TTYPX_MINOR) < NR_PTYS) {
11856                 tp = tty_addr(line - TTYPX_MINOR + NR_CONS + NR_RS_LINES);
11857           } else
11858           if ((line - PTYPX_MINOR) < NR_PTYS) {
11859                 tp = tty_addr(line - PTYPX_MINOR + NR_CONS + NR_RS_LINES);
11860                 do_pty(tp, &tty_mess);
11861                 continue;                       /* this is a pty, not a tty */
11862           } else {
11863                 tp = NULL;
11864           }
11865
11866           /* If the device doesn't exist or is not configured return ENXIO. */
11867           if (tp == NULL || !tty_active(tp)) {
11868                 tty_reply(TASK_REPLY, tty_mess.m_source,
11869                                                 tty_mess.PROC_NR, ENXIO);
11870                 continue;
11871           }
11872
11873           /* Execute the requested function. */
11874           switch (tty_mess.m_type) {
11875               case DEV_READ:      do_read(tp, &tty_mess);         break;
11876               case DEV_WRITE:     do_write(tp, &tty_mess);        break;
11877               case DEV_IOCTL:     do_ioctl(tp, &tty_mess);        break;
11878               case DEV_OPEN:      do_open(tp, &tty_mess);         break;
11879               case DEV_CLOSE:     do_close(tp, &tty_mess);        break;
11880               case CANCEL:        do_cancel(tp, &tty_mess);       break;
11881               default:            tty_reply(TASK_REPLY, tty_mess.m_source,
11882                                                 tty_mess.PROC_NR, EINVAL);
11883           }
11884     }
11885   }


11888   /*===========================================================================*
11889    *                                do_read                                    *
11890    *===========================================================================*/
11891   PRIVATE void do_read(tp, m_ptr)
11892   register tty_t *tp;             /* pointer to tty struct */
11893   message *m_ptr;                 /* pointer to message sent to the task */
11894   {
11895   /* A process wants to read from a terminal. */
11896     int r;
11897
11898     /* Check if there is already a process hanging in a read, check if the
11899      * parameters are correct, do I/O.
11900      */
11901     if (tp->tty_inleft > 0) {
11902           r = EIO;
11903     } else
11904     if (m_ptr->COUNT <= 0) {
11905           r = EINVAL;
11906     } else
11907     if (numap(m_ptr->PROC_NR, (vir_bytes) m_ptr->ADDRESS, m_ptr->COUNT) == 0) {
11908           r = EFAULT;
11909     } else {
11910           /* Copy information from the message to the tty struct. */
11911           tp->tty_inrepcode = TASK_REPLY;
11912           tp->tty_incaller = m_ptr->m_source;
11913           tp->tty_inproc = m_ptr->PROC_NR;
11914           tp->tty_in_vir = (vir_bytes) m_ptr->ADDRESS;
```

```
11915           tp->tty_inleft = m_ptr->COUNT;
11916
11917           if (!(tp->tty_termios.c_lflag & ICANON)
11918                                   && tp->tty_termios.c_cc[VTIME] > 0) {
11919                   if (tp->tty_termios.c_cc[VMIN] == 0) {
11920                           /* MIN & TIME specify a read timer that finishes the
11921                            * read in TIME/10 seconds if no bytes are available.
11922                            */
11923                           lock();
11924                           settimer(tp, TRUE);
11925                           tp->tty_min = 1;
11926                           unlock();
11927                   } else {
11928                           /* MIN & TIME specify an inter-byte timer that may
11929                            * have to be cancelled if there are no bytes yet.
11930                            */
11931                           if (tp->tty_eotct == 0) {
11932                                   lock();
11933                                   settimer(tp, FALSE);
11934                                   unlock();
11935                                   tp->tty_min = tp->tty_termios.c_cc[VMIN];
11936                           }
11937                   }
11938           }
11939
11940           /* Anything waiting in the input buffer? Clear it out... */
11941           in_transfer(tp);
11942           /* ...then go back for more */
11943           handle_events(tp);
11944           if (tp->tty_inleft == 0) return;                /* already done */
11945
11946           /* There were no bytes in the input queue available, so either suspend
11947            * the caller or break off the read if nonblocking.
11948            */
11949           if (m_ptr->TTY_FLAGS & O_NONBLOCK) {
11950                   r = EAGAIN;                             /* cancel the read */
11951                   tp->tty_inleft = tp->tty_incum = 0;
11952           } else {
11953                   r = SUSPEND;                            /* suspend the caller */
11954                   tp->tty_inrepcode = REVIVE;
11955           }
11956     }
11957     tty_reply(TASK_REPLY, m_ptr->m_source, m_ptr->PROC_NR, r);
11958   }


11961   /*===========================================================================*
11962    *                              do_write                                     *
11963    *===========================================================================*/
11964   PRIVATE void do_write(tp, m_ptr)
11965   register tty_t *tp;
11966   register message *m_ptr;        /* pointer to message sent to the task */
11967   {
11968   /* A process wants to write on a terminal. */
11969     int r;
11970
11971     /* Check if there is already a process hanging in a write, check if the
11972      * parameters are correct, do I/O.
11973      */
11974     if (tp->tty_outleft > 0) {
```



```
11975           r = EIO;
11976     } else
11977     if (m_ptr->COUNT <= 0) {
11978           r = EINVAL;
11979     } else
11980     if (numap(m_ptr->PROC_NR, (vir_bytes) m_ptr->ADDRESS, m_ptr->COUNT) == 0) {
11981           r = EFAULT;
11982     } else {
11983           /* Copy message parameters to the tty structure. */
11984           tp->tty_outrepcode = TASK_REPLY;
11985           tp->tty_outcaller = m_ptr->m_source;
11986           tp->tty_outproc = m_ptr->PROC_NR;
11987           tp->tty_out_vir = (vir_bytes) m_ptr->ADDRESS;
11988           tp->tty_outleft = m_ptr->COUNT;
11989
11990           /* Try to write. */
11991           handle_events(tp);
11992           if (tp->tty_outleft == 0) return;               /* already done */
11993
11994           /* None or not all the bytes could be written, so either suspend the
11995            * caller or break off the write if nonblocking.
11996            */
11997           if (m_ptr->TTY_FLAGS & O_NONBLOCK) {            /* cancel the write */
11998                   r = tp->tty_outcum > 0 ? tp->tty_outcum : EAGAIN;
11999                   tp->tty_outleft = tp->tty_outcum = 0;
12000           } else {
12001                   r = SUSPEND;                            /* suspend the caller */
12002                   tp->tty_outrepcode = REVIVE;
12003           }
12004     }
12005     tty_reply(TASK_REPLY, m_ptr->m_source, m_ptr->PROC_NR, r);
12006   }


12009   /*===========================================================================*
12010    *                              do_ioctl                                     *
12011    *===========================================================================*/
12012   PRIVATE void do_ioctl(tp, m_ptr)
12013   register tty_t *tp;
12014   message *m_ptr;                 /* pointer to message sent to task */
12015   {
12016   /* Perform an IOCTL on this terminal. Posix termios calls are handled
12017    * by the IOCTL system call
12018    */
12019
12020     int r;
12021     union {
12022           int i;
12023           /* these non-Posix params are not used now, but the union is retained
12024            * to minimize code differences with backward compatible version
12025            * struct sgttyb sg;
12026            * struct tchars tc;
12027            */
12028     } param;
12029     phys_bytes user_phys;
12030     size_t size;
12031
12032     /* Size of the ioctl parameter. */
12033     switch (m_ptr->TTY_REQUEST) {
12034       case TCGETS:        /* Posix tcgetattr function */
```

```
12035       case TCSETS:        /* Posix tcsetattr function, TCSANOW option */
12036       case TCSETSW:       /* Posix tcsetattr function, TCSADRAIN option */
12037       case TCSETSF:       /* Posix tcsetattr function, TCSAFLUSH option */
12038           size = sizeof(struct termios);
12039           break;
12040
12041       case TCSBRK:        /* Posix tcsendbreak function */
12042       case TCFLOW:        /* Posix tcflow function */
12043       case TCFLSH:        /* Posix tcflush function */
12044       case TIOCGPGRP:     /* Posix tcgetpgrp function */
12045       case TIOCSPGRP:     /* Posix tcsetpgrp function */
12046           size = sizeof(int);
12047           break;
12048
12049       case TIOCGWINSZ:    /* get window size (not Posix) */
12050       case TIOCSWINSZ:    /* set window size (not Posix) */
12051           size = sizeof(struct winsize);
12052           break;
12053
12054       case KIOCSMAP:      /* load keymap (Minix extension) */
12055           size = sizeof(keymap_t);
12056           break;
12057
12058       case TIOCSFON:      /* load font (Minix extension) */
12059           size = sizeof(u8_t [8192]);
12060           break;
12061
12062       case TCDRAIN:       /* Posix tcdrain function -- no parameter */
12063       default:            size = 0;
12064     }
12065
12066     if (size != 0) {
12067           user_phys = numap(m_ptr->PROC_NR, (vir_bytes) m_ptr->ADDRESS, size);
12068           if (user_phys == 0) {
12069                   tty_reply(TASK_REPLY, m_ptr->m_source, m_ptr->PROC_NR, EFAULT);
12070                   return;
12071           }
12072     }
12073
12074     r = OK;
12075     switch (m_ptr->TTY_REQUEST) {
12076       case TCGETS:
12077           /* Get the termios attributes. */
12078           phys_copy(vir2phys(&tp->tty_termios), user_phys, (phys_bytes) size);
12079           break;
12080
12081       case TCSETSW:
12082       case TCSETSF:
12083       case TCDRAIN:
12084           if (tp->tty_outleft > 0) {
12085                   /* Wait for all ongoing output processing to finish. */
12086                   tp->tty_iocaller = m_ptr->m_source;
12087                   tp->tty_ioproc = m_ptr->PROC_NR;
12088                   tp->tty_ioreq = m_ptr->REQUEST;
12089                   tp->tty_iovir = (vir_bytes) m_ptr->ADDRESS;
12090                   r = SUSPEND;
12091                   break;
12092           }
12093           if (m_ptr->TTY_REQUEST == TCDRAIN) break;
12094           if (m_ptr->TTY_REQUEST == TCSETSF) tty_icancel(tp);
```



```
12095           /*FALL THROUGH*/
12096       case TCSETS:
12097           /* Set the termios attributes. */
12098           phys_copy(user_phys, vir2phys(&tp->tty_termios), (phys_bytes) size);
12099           setattr(tp);
12100           break;
12101
12102       case TCFLSH:
12103           phys_copy(user_phys, vir2phys(&param.i), (phys_bytes) size);
12104           switch (param.i) {
12105               case TCIFLUSH:      tty_icancel(tp);                        break;
12106               case TCOFLUSH:      (*tp->tty_ocancel)(tp);                 break;
12107               case TCIOFLUSH:     tty_icancel(tp); (*tp->tty_ocancel)(tp);break;
12108               default:            r = EINVAL;
12109           }
12110           break;
12111
12112       case TCFLOW:
12113           phys_copy(user_phys, vir2phys(&param.i), (phys_bytes) size);
12114           switch (param.i) {
12115               case TCOOFF:
12116               case TCOON:
12117                   tp->tty_inhibited = (param.i == TCOOFF);
12118                   tp->tty_events = 1;
12119                   break;
12120               case TCIOFF:
12121                   (*tp->tty_echo)(tp, tp->tty_termios.c_cc[VSTOP]);
12122                   break;
12123               case TCION:
12124                   (*tp->tty_echo)(tp, tp->tty_termios.c_cc[VSTART]);
12125                   break;
12126               default:
12127                   r = EINVAL;
12128           }
12129           break;
12130
12131       case TCSBRK:
12132           if (tp->tty_break != NULL) (*tp->tty_break)(tp);
12133           break;
12134
12135       case TIOCGWINSZ:
12136           phys_copy(vir2phys(&tp->tty_winsize), user_phys, (phys_bytes) size);
12137           break;
12138
12139       case TIOCSWINSZ:
12140           phys_copy(user_phys, vir2phys(&tp->tty_winsize), (phys_bytes) size);
12141           /* SIGWINCH... */
12142           break;
12143
12144       case KIOCSMAP:
12145           /* Load a new keymap (only /dev/console). */
12146           if (isconsole(tp)) r = kbd_loadmap(user_phys);
12147           break;
12148
12149       case TIOCSFON:
12150           /* Load a font into an EGA or VGA card (hs@hck.hr) */
12151           if (isconsole(tp)) r = con_loadfont(user_phys);
12152           break;
12153
12154   /* These Posix functions are allowed to fail if _POSIX_JOB_CONTROL is
```

```
12155     * not defined.
12156     */
12157        case TIOCGPGRP:
12158        case TIOCSPGRP:
12159        default:
12160            r = ENOTTY;
12161     }
12162
12163     /* Send the reply. */
12164     tty_reply(TASK_REPLY, m_ptr->m_source, m_ptr->PROC_NR, r);
12165   }


12168   /*===========================================================================*
12169    *                              do_open                                      *
12170    *===========================================================================*/
12171   PRIVATE void do_open(tp, m_ptr)
12172   register tty_t *tp;
12173   message *m_ptr;                   /* pointer to message sent to task */
12174   {
12175   /* A tty line has been opened.  Make it the callers controlling tty if
12176    * O_NOCTTY is *not* set and it is not the log device.  1 is returned if
12177    * the tty is made the controlling tty, otherwise OK or an error code.
12178    */
12179     int r = OK;
12180
12181     if (m_ptr->TTY_LINE == LOG_MINOR) {
12182           /* The log device is a write-only diagnostics device. */
12183           if (m_ptr->COUNT & R_BIT) r = EACCES;
12184     } else {
12185           if (!(m_ptr->COUNT & O_NOCTTY)) {
12186                   tp->tty_pgrp = m_ptr->PROC_NR;
12187                   r = 1;
12188           }
12189           tp->tty_openct++;
12190     }
12191     tty_reply(TASK_REPLY, m_ptr->m_source, m_ptr->PROC_NR, r);
12192   }


12195   /*===========================================================================*
12196    *                              do_close                                     *
12197    *===========================================================================*/
12198   PRIVATE void do_close(tp, m_ptr)
12199   register tty_t *tp;
12200   message *m_ptr;                   /* pointer to message sent to task */
12201   {
12202   /* A tty line has been closed.  Clean up the line if it is the last close. */
12203
12204     if (m_ptr->TTY_LINE != LOG_MINOR && --tp->tty_openct == 0) {
12205           tp->tty_pgrp = 0;
12206           tty_icancel(tp);
12207           (*tp->tty_ocancel)(tp);
12208           (*tp->tty_close)(tp);
12209           tp->tty_termios = termios_defaults;
12210           tp->tty_winsize = winsize_defaults;
12211           setattr(tp);
12212     }
12213     tty_reply(TASK_REPLY, m_ptr->m_source, m_ptr->PROC_NR, OK);
12214   }
```



```
12217   /*===========================================================================*
12218    *                              do_cancel                                    *
12219    *===========================================================================*/
12220   PRIVATE void do_cancel(tp, m_ptr)
12221   register tty_t *tp;
12222   message *m_ptr;                   /* pointer to message sent to task */
12223   {
12224   /* A signal has been sent to a process that is hanging trying to read or write.
12225    * The pending read or write must be finished off immediately.
12226    */
12227
12228     int proc_nr;
12229     int mode;
12230
12231     /* Check the parameters carefully, to avoid cancelling twice. */
12232     proc_nr = m_ptr->PROC_NR;
12233     mode = m_ptr->COUNT;
12234     if ((mode & R_BIT) && tp->tty_inleft != 0 && proc_nr == tp->tty_inproc) {
12235           /* Process was reading when killed.  Clean up input. */
12236           tty_icancel(tp);
12237           tp->tty_inleft = tp->tty_incum = 0;
12238     }
12239     if ((mode & W_BIT) && tp->tty_outleft != 0 && proc_nr == tp->tty_outproc) {
12240           /* Process was writing when killed.  Clean up output. */
12241           (*tp->tty_ocancel)(tp);
12242           tp->tty_outleft = tp->tty_outcum = 0;
12243     }
12244     if (tp->tty_ioreq != 0 && proc_nr == tp->tty_ioproc) {
12245           /* Process was waiting for output to drain. */
12246           tp->tty_ioreq = 0;
12247     }
12248     tp->tty_events = 1;
12249     tty_reply(TASK_REPLY, m_ptr->m_source, proc_nr, EINTR);
12250   }


12253   /*===========================================================================*
12254    *                              handle_events                                *
12255    *===========================================================================*/
12256   PUBLIC void handle_events(tp)
12257   tty_t *tp;                        /* TTY to check for events. */
12258   {
12259   /* Handle any events pending on a TTY.  These events are usually device
12260    * interrupts.
12261    *
12262    * Two kinds of events are prominent:
12263    *      - a character has been received from the console or an RS232 line.
12264    *      - an RS232 line has completed a write request (on behalf of a user).
12265    * The interrupt handler may delay the interrupt message at its discretion
12266    * to avoid swamping the TTY task.  Messages may be overwritten when the
12267    * lines are fast or when there are races between different lines, input
12268    * and output, because MINIX only provides single buffering for interrupt
12269    * messages (in proc.c).  This is handled by explicitly checking each line
12270    * for fresh input and completed output on each interrupt.
12271    */
12272     char *buf;
12273     unsigned count;
12274
```

```
12275     do {
12276           tp->tty_events = 0;
12277
12278           /* Read input and perform input processing. */
12279           (*tp->tty_devread)(tp);
12280
12281           /* Perform output processing and write output. */
12282           (*tp->tty_devwrite)(tp);
12283
12284           /* Ioctl waiting for some event? */
12285           if (tp->tty_ioreq != 0) dev_ioctl(tp);
12286     } while (tp->tty_events);
12287
12288     /* Transfer characters from the input queue to a waiting process. */
12289     in_transfer(tp);
12290
12291     /* Reply if enough bytes are available. */
12292     if (tp->tty_incum >= tp->tty_min && tp->tty_inleft > 0) {
12293           tty_reply(tp->tty_inrepcode, tp->tty_incaller, tp->tty_inproc,
12294                                                           tp->tty_incum);
12295           tp->tty_inleft = tp->tty_incum = 0;
12296     }
12297   }


12300   /*===========================================================================*
12301    *                              in_transfer                                  *
12302    *===========================================================================*/
12303   PRIVATE void in_transfer(tp)
12304   register tty_t *tp;             /* pointer to terminal to read from */
12305   {
12306   /* Transfer bytes from the input queue to a process reading from a terminal. */
12307
12308     int ch;
12309     int count;
12310     phys_bytes buf_phys, user_base;
12311     char buf[64], *bp;
12312
12313     /* Anything to do? */
12314     if (tp->tty_inleft == 0 || tp->tty_eotct < tp->tty_min) return;
12315
12316     buf_phys = vir2phys(buf);
12317     user_base = proc_vir2phys(proc_addr(tp->tty_inproc), 0);
12318     bp = buf;
12319     while (tp->tty_inleft > 0 && tp->tty_eotct > 0) {
12320           ch = *tp->tty_intail;
12321
12322           if (!(ch & IN_EOF)) {
12323                   /* One character to be delivered to the user. */
12324                   *bp = ch & IN_CHAR;
12325                   tp->tty_inleft--;
12326                   if (++bp == bufend(buf)) {
12327                           /* Temp buffer full, copy to user space. */
12328                           phys_copy(buf_phys, user_base + tp->tty_in_vir,
12329                                                   (phys_bytes) buflen(buf));
12330                           tp->tty_in_vir += buflen(buf);
12331                           tp->tty_incum += buflen(buf);
12332                           bp = buf;
12333                   }
12334           }
```



```
12335
12336           /* Remove the character from the input queue. */
12337           if (++tp->tty_intail == bufend(tp->tty_inbuf))
12338                   tp->tty_intail = tp->tty_inbuf;
12339           tp->tty_incount--;
12340           if (ch & IN_EOT) {
12341                   tp->tty_eotct--;
12342                   /* Don't read past a line break in canonical mode. */
12343                   if (tp->tty_termios.c_lflag & ICANON) tp->tty_inleft = 0;
12344           }
12345     }
12346
12347     if (bp > buf) {
12348           /* Leftover characters in the buffer. */
12349           count = bp - buf;
12350           phys_copy(buf_phys, user_base + tp->tty_in_vir, (phys_bytes) count);
12351           tp->tty_in_vir += count;
12352           tp->tty_incum += count;
12353     }
12354
12355     /* Usually reply to the reader, possibly even if incum == 0 (EOF). */
12356     if (tp->tty_inleft == 0) {
12357           tty_reply(tp->tty_inrepcode, tp->tty_incaller, tp->tty_inproc,
12358                                                           tp->tty_incum);
12359           tp->tty_inleft = tp->tty_incum = 0;
12360     }
12361   }


12364   /*===========================================================================*
12365    *                              in_process                                   *
12366    *===========================================================================*/
12367   PUBLIC int in_process(tp, buf, count)
12368   register tty_t *tp;             /* terminal on which character has arrived */
12369   char *buf;                      /* buffer with input characters */
12370   int count;                      /* number of input characters */
12371   {
12372   /* Characters have just been typed in.  Process, save, and echo them.  Return
12373    * the number of characters processed.
12374    */
12375
12376     int ch, sig, ct;
12377     int timeset = FALSE;
12378     static unsigned char csize_mask[] = { 0x1F, 0x3F, 0x7F, 0xFF };
12379
12380     for (ct = 0; ct < count; ct++) {
12381           /* Take one character. */
12382           ch = *buf++ & BYTE;
12383
12384           /* Strip to seven bits? */
12385           if (tp->tty_termios.c_iflag & ISTRIP) ch &= 0x7F;
12386
12387           /* Input extensions? */
12388           if (tp->tty_termios.c_lflag & IEXTEN) {
12389
12390                   /* Previous character was a character escape? */
12391                   if (tp->tty_escaped) {
12392                           tp->tty_escaped = NOT_ESCAPED;
12393                           ch |= IN_ESC;   /* protect character */
12394                   }
```

```
12395
12396                   /* LNEXT (^V) to escape the next character? */
12397                   if (ch == tp->tty_termios.c_cc[VLNEXT]) {
12398                           tp->tty_escaped = ESCAPED;
12399                           rawecho(tp, '^');
12400                           rawecho(tp, '\b');
12401                           continue;       /* do not store the escape */
12402                   }
12403
12404                   /* REPRINT (^R) to reprint echoed characters? */
12405                   if (ch == tp->tty_termios.c_cc[VREPRINT]) {
12406                           reprint(tp);
12407                           continue;
12408                   }
12409           }
12410
12411           /* _POSIX_VDISABLE is a normal character value, so better escape it. */
12412           if (ch == _POSIX_VDISABLE) ch |= IN_ESC;
12413
12414           /* Map CR to LF, ignore CR, or map LF to CR. */
12415           if (ch == '\r') {
12416                   if (tp->tty_termios.c_iflag & IGNCR) continue;
12417                   if (tp->tty_termios.c_iflag & ICRNL) ch = '\n';
12418           } else
12419           if (ch == '\n') {
12420                   if (tp->tty_termios.c_iflag & INLCR) ch = '\r';
12421           }
12422
12423           /* Canonical mode? */
12424           if (tp->tty_termios.c_lflag & ICANON) {
12425
12426                   /* Erase processing (rub out of last character). */
12427                   if (ch == tp->tty_termios.c_cc[VERASE]) {
12428                           (void) back_over(tp);
12429                           if (!(tp->tty_termios.c_lflag & ECHOE)) {
12430                                   (void) echo(tp, ch);
12431                           }
12432                           continue;
12433                   }
12434
12435                   /* Kill processing (remove current line). */
12436                   if (ch == tp->tty_termios.c_cc[VKILL]) {
12437                           while (back_over(tp)) {}
12438                           if (!(tp->tty_termios.c_lflag & ECHOE)) {
12439                                   (void) echo(tp, ch);
12440                                   if (tp->tty_termios.c_lflag & ECHOK)
12441                                           rawecho(tp, '\n');
12442                           }
12443                           continue;
12444                   }
12445
12446                   /* EOF (^D) means end-of-file, an invisible "line break". */
12447                   if (ch == tp->tty_termios.c_cc[VEOF]) ch |= IN_EOT | IN_EOF;
12448
12449                   /* The line may be returned to the user after an LF. */
12450                   if (ch == '\n') ch |= IN_EOT;
12451
12452                   /* Same thing with EOL, whatever it may be. */
12453                   if (ch == tp->tty_termios.c_cc[VEOL]) ch |= IN_EOT;
12454           }
```



```
12455
12456           /* Start/stop input control? */
12457           if (tp->tty_termios.c_iflag & IXON) {
12458
12459                   /* Output stops on STOP (^S). */
12460                   if (ch == tp->tty_termios.c_cc[VSTOP]) {
12461                           tp->tty_inhibited = STOPPED;
12462                           tp->tty_events = 1;
12463                           continue;
12464                   }
12465
12466                   /* Output restarts on START (^Q) or any character if IXANY. */
12467                   if (tp->tty_inhibited) {
12468                           if (ch == tp->tty_termios.c_cc[VSTART]
12469                                           || (tp->tty_termios.c_iflag & IXANY)) {
12470                                   tp->tty_inhibited = RUNNING;
12471                                   tp->tty_events = 1;
12472                                   if (ch == tp->tty_termios.c_cc[VSTART])
12473                                           continue;
12474                           }
12475                   }
12476           }
12477
12478           if (tp->tty_termios.c_lflag & ISIG) {
12479                   /* Check for INTR (^?) and QUIT (^\) characters. */
12480                   if (ch == tp->tty_termios.c_cc[VINTR]
12481                                           || ch == tp->tty_termios.c_cc[VQUIT]) {
12482                           sig = SIGINT;
12483                           if (ch == tp->tty_termios.c_cc[VQUIT]) sig = SIGQUIT;
12484                           sigchar(tp, sig);
12485                           (void) echo(tp, ch);
12486                           continue;
12487                   }
12488           }
12489
12490           /* Is there space in the input buffer? */
12491           if (tp->tty_incount == buflen(tp->tty_inbuf)) {
12492                   /* No space; discard in canonical mode, keep in raw mode. */
12493                   if (tp->tty_termios.c_lflag & ICANON) continue;
12494                   break;
12495           }
12496
12497           if (!(tp->tty_termios.c_lflag & ICANON)) {
12498                   /* In raw mode all characters are "line breaks". */
12499                   ch |= IN_EOT;
12500
12501                   /* Start an inter-byte timer? */
12502                   if (!timeset && tp->tty_termios.c_cc[VMIN] > 0
12503                                   && tp->tty_termios.c_cc[VTIME] > 0) {
12504                           lock();
12505                           settimer(tp, TRUE);
12506                           unlock();
12507                           timeset = TRUE;
12508                   }
12509           }
12510
12511           /* Perform the intricate function of echoing. */
12512           if (tp->tty_termios.c_lflag & (ECHO|ECHONL)) ch = echo(tp, ch);
12513
12514           /* Save the character in the input queue. */
```

```
12515           *tp->tty_inhead++ = ch;
12516           if (tp->tty_inhead == bufend(tp->tty_inbuf))
12517                   tp->tty_inhead = tp->tty_inbuf;
12518           tp->tty_incount++;
12519           if (ch & IN_EOT) tp->tty_eotct++;
12520
12521           /* Try to finish input if the queue threatens to overflow. */
12522           if (tp->tty_incount == buflen(tp->tty_inbuf)) in_transfer(tp);
12523       }
12524       return ct;
12525   }


12528   /*==========================================================================*
12529    *                              echo                                        *
12530    *==========================================================================*/
12531   PRIVATE int echo(tp, ch)
12532   register tty_t *tp;             /* terminal on which to echo */
12533   register int ch;                /* pointer to character to echo */
12534   {
12535   /* Echo the character if echoing is on.  Some control characters are echoed
12536    * with their normal effect, other control characters are echoed as "^X",
12537    * normal characters are echoed normally.  EOF (^D) is echoed, but immediately
12538    * backspaced over.  Return the character with the echoed length added to its
12539    * attributes.
12540    */
12541     int len, rp;
12542
12543     ch &= ~IN_LEN;
12544     if (!(tp->tty_termios.c_lflag & ECHO)) {
12545           if (ch == ('\n' | IN_EOT) && (tp->tty_termios.c_lflag
12546                                       & (ICANON|ECHONL)) == (ICANON|ECHONL))
12547                   (*tp->tty_echo)(tp, '\n');
12548           return(ch);
12549     }
12550
12551     /* "Reprint" tells if the echo output has been messed up by other output. */
12552     rp = tp->tty_incount == 0 ? FALSE : tp->tty_reprint;
12553
12554     if ((ch & IN_CHAR) < ' ') {
12555           switch (ch & (IN_ESC|IN_EOF|IN_EOT|IN_CHAR)) {
12556               case '\t':
12557                   len = 0;
12558                   do {
12559                           (*tp->tty_echo)(tp, ' ');
12560                           len++;
12561                   } while (len < TAB_SIZE && (tp->tty_position & TAB_MASK) != 0);
12562                   break;
12563               case '\r' | IN_EOT:
12564               case '\n' | IN_EOT:
12565                   (*tp->tty_echo)(tp, ch & IN_CHAR);
12566                   len = 0;
12567                   break;
12568               default:
12569                   (*tp->tty_echo)(tp, '^');
12570                   (*tp->tty_echo)(tp, '@' + (ch & IN_CHAR));
12571                   len = 2;
12572           }
12573     } else
12574     if ((ch & IN_CHAR) == '\177') {
```



```
12575           /* A DEL prints as "^?". */
12576           (*tp->tty_echo)(tp, '^');
12577           (*tp->tty_echo)(tp, '?');
12578           len = 2;
12579       } else {
12580           (*tp->tty_echo)(tp, ch & IN_CHAR);
12581           len = 1;
12582       }
12583     if (ch & IN_EOF) while (len > 0) { (*tp->tty_echo)(tp, '\b'); len--; }
12584
12585     tp->tty_reprint = rp;
12586     return(ch | (len << IN_LSHIFT));
12587   }


12590   /*==========================================================================*
12591    *                              rawecho                                     *
12592    *==========================================================================*/
12593   PRIVATE void rawecho(tp, ch)
12594   register tty_t *tp;
12595   int ch;
12596   {
12597   /* Echo without interpretation if ECHO is set. */
12598     int rp = tp->tty_reprint;
12599     if (tp->tty_termios.c_lflag & ECHO) (*tp->tty_echo)(tp, ch);
12600     tp->tty_reprint = rp;
12601   }


12604   /*==========================================================================*
12605    *                              back_over                                   *
12606    *==========================================================================*/
12607   PRIVATE int back_over(tp)
12608   register tty_t *tp;
12609   {
12610   /* Backspace to previous character on screen and erase it. */
12611     u16_t *head;
12612     int len;
12613
12614     if (tp->tty_incount == 0) return(0);  /* queue empty */
12615     head = tp->tty_inhead;
12616     if (head == tp->tty_inbuf) head = bufend(tp->tty_inbuf);
12617     if (*--head & IN_EOT) return(0);              /* can't erase "line breaks" */
12618     if (tp->tty_reprint) reprint(tp);             /* reprint if messed up */
12619     tp->tty_inhead = head;
12620     tp->tty_incount--;
12621     if (tp->tty_termios.c_lflag & ECHOE) {
12622           len = (*head & IN_LEN) >> IN_LSHIFT;
12623           while (len > 0) {
12624                   rawecho(tp, '\b');
12625                   rawecho(tp, ' ');
12626                   rawecho(tp, '\b');
12627                   len--;
12628           }
12629     }
12630     return(1);                          /* one character erased */
12631   }


12634   /*==========================================================================*
```

```
12635    *                              reprint                                  *
12636    *===========================================================================*/
12637   PRIVATE void reprint(tp)
12638   register tty_t *tp;             /* pointer to tty struct */
12639   {
12640   /* Restore what has been echoed to screen before if the user input has been
12641    * messed up by output, or if REPRINT (^R) is typed.
12642    */
12643     int count;
12644     u16_t *head;
12645
12646     tp->tty_reprint = FALSE;
12647
12648     /* Find the last line break in the input. */
12649     head = tp->tty_inhead;
12650     count = tp->tty_incount;
12651     while (count > 0) {
12652           if (head == tp->tty_inbuf) head = bufend(tp->tty_inbuf);
12653           if (head[-1] & IN_EOT) break;
12654           head--;
12655           count--;
12656     }
12657     if (count == tp->tty_incount) return;         /* no reason to reprint */
12658
12659     /* Show REPRINT (^R) and move to a new line. */
12660     (void) echo(tp, tp->tty_termios.c_cc[VREPRINT] | IN_ESC);
12661     rawecho(tp, '\r');
12662     rawecho(tp, '\n');
12663
12664     /* Reprint from the last break onwards. */
12665     do {
12666           if (head == bufend(tp->tty_inbuf)) head = tp->tty_inbuf;
12667           *head = echo(tp, *head);
12668           head++;
12669           count++;
12670     } while (count < tp->tty_incount);
12671   }


12674   /*===========================================================================*
12675    *                              out_process                                  *
12676    *===========================================================================*/
12677   PUBLIC void out_process(tp, bstart, bpos, bend, icount, ocount)
12678   tty_t *tp;
12679   char *bstart, *bpos, *bend;     /* start/pos/end of circular buffer */
12680   int *icount;                    /* # input chars / input chars used */
12681   int *ocount;                    /* max output chars / output chars used */
12682   {
12683   /* Perform output processing on a circular buffer.  *icount is the number of
12684    * bytes to process, and the number of bytes actually processed on return.
12685    * *ocount is the space available on input and the space used on output.
12686    * (Naturally *icount < *ocount.)  The column position is updated modulo
12687    * the TAB size, because we really only need it for tabs.
12688    */
12689
12690     int tablen;
12691     int ict = *icount;
12692     int oct = *ocount;
12693     int pos = tp->tty_position;
12694
```



```
12695     while (ict > 0) {
12696           switch (*bpos) {
12697           case '\7':
12698                   break;
12699           case '\b':
12700                   pos--;
12701                   break;
12702           case '\r':
12703                   pos = 0;
12704                   break;
12705           case '\n':
12706                   if ((tp->tty_termios.c_oflag & (OPOST|ONLCR))
12707                                                   == (OPOST|ONLCR)) {
12708                           /* Map LF to CR+LF if there is space.  Note that the
12709                            * next character in the buffer is overwritten, so
12710                            * we stop at this point.
12711                            */
12712                           if (oct >= 2) {
12713                                   *bpos = '\r';
12714                                   if (++bpos == bend) bpos = bstart;
12715                                   *bpos = '\n';
12716                                   pos = 0;
12717                                   ict--;
12718                                   oct -= 2;
12719                           }
12720                           goto out_done;  /* no space or buffer got changed */
12721                   }
12722                   break;
12723           case '\t':
12724                   /* Best guess for the tab length. */
12725                   tablen = TAB_SIZE - (pos & TAB_MASK);
12726
12727                   if ((tp->tty_termios.c_oflag & (OPOST|XTABS))
12728                                                   == (OPOST|XTABS)) {
12729                           /* Tabs must be expanded. */
12730                           if (oct >= tablen) {
12731                                   pos += tablen;
12732                                   ict--;
12733                                   oct -= tablen;
12734                                   do {
12735                                           *bpos = ' ';
12736                                           if (++bpos == bend) bpos = bstart;
12737                                   } while (--tablen != 0);
12738                           }
12739                           goto out_done;
12740                   }
12741                   /* Tabs are output directly. */
12742                   pos += tablen;
12743                   break;
12744           default:
12745                   /* Assume any other character prints as one character. */
12746                   pos++;
12747           }
12748           if (++bpos == bend) bpos = bstart;
12749           ict--;
12750           oct--;
12751     }
12752   out_done:
12753     tp->tty_position = pos & TAB_MASK;
12754
```

```
12755     *icount -= ict;       /* [io]ct are the number of chars not used */
12756     *ocount -= oct;       /* *[io]count are the number of chars that are used */
12757   }


12760   /*===========================================================================*
12761    *                              dev_ioctl                                    *
12762    *===========================================================================*/
12763   PRIVATE void dev_ioctl(tp)
12764   tty_t *tp;
12765   {
12766   /* The ioctl's TCSETSW, TCSETSF and TCDRAIN wait for output to finish to make
12767    * sure that an attribute change doesn't affect the processing of current
12768    * output.  Once output finishes the ioctl is executed as in do_ioctl().
12769    */
12770     phys_bytes user_phys;
12771
12772     if (tp->tty_outleft > 0) return;              /* output not finished */
12773
12774     if (tp->tty_ioreq != TCDRAIN) {
12775           if (tp->tty_ioreq == TCSETSF) tty_icancel(tp);
12776           user_phys = proc_vir2phys(proc_addr(tp->tty_ioproc), tp->tty_iovir);
12777           phys_copy(user_phys, vir2phys(&tp->tty_termios),
12778                                       (phys_bytes) sizeof(tp->tty_termios));
12779           setattr(tp);
12780     }
12781     tp->tty_ioreq = 0;
12782     tty_reply(REVIVE, tp->tty_iocaller, tp->tty_ioproc, OK);
12783   }


12786   /*===========================================================================*
12787    *                              setattr                                      *
12788    *===========================================================================*/
12789   PRIVATE void setattr(tp)
12790   tty_t *tp;
12791   {
12792   /* Apply the new line attributes (raw/canonical, line speed, etc.) */
12793     u16_t *inp;
12794     int count;
12795
12796     if (!(tp->tty_termios.c_lflag & ICANON)) {
12797           /* Raw mode; put a "line break" on all characters in the input queue.
12798            * It is undefined what happens to the input queue when ICANON is
12799            * switched off, a process should use TCSAFLUSH to flush the queue.
12800            * Keeping the queue to preserve typeahead is the Right Thing, however
12801            * when a process does use TCSANOW to switch to raw mode.
12802            */
12803           count = tp->tty_eotct = tp->tty_incount;
12804           inp = tp->tty_intail;
12805           while (count > 0) {
12806                   *inp |= IN_EOT;
12807                   if (++inp == bufend(tp->tty_inbuf)) inp = tp->tty_inbuf;
12808                   --count;
12809           }
12810     }
12811
12812     /* Inspect MIN and TIME. */
12813     lock();
12814     settimer(tp, FALSE);
```



```
12815     unlock();
12816     if (tp->tty_termios.c_lflag & ICANON) {
12817           /* No MIN & TIME in canonical mode. */
12818           tp->tty_min = 1;
12819     } else {
12820           /* In raw mode MIN is the number of chars wanted, and TIME how long
12821            * to wait for them.  With interesting exceptions if either is zero.
12822            */
12823           tp->tty_min = tp->tty_termios.c_cc[VMIN];
12824           if (tp->tty_min == 0 && tp->tty_termios.c_cc[VTIME] > 0)
12825                   tp->tty_min = 1;
12826     }
12827
12828     if (!(tp->tty_termios.c_iflag & IXON)) {
12829           /* No start/stop output control, so don't leave output inhibited. */
12830           tp->tty_inhibited = RUNNING;
12831           tp->tty_events = 1;
12832     }
12833
12834     /* Setting the output speed to zero hangs up the phone. */
12835     if (tp->tty_termios.c_ospeed == B0) sigchar(tp, SIGHUP);
12836
12837     /* Set new line speed, character size, etc at the device level. */
12838     (*tp->tty_ioctl)(tp);
12839   }


12842   /*===========================================================================*
12843    *                              tty_reply                                    *
12844    *===========================================================================*/
12845   PUBLIC void tty_reply(code, replyee, proc_nr, status)
12846   int code;                     /* TASK_REPLY or REVIVE */
12847   int replyee;                  /* destination address for the reply */
12848   int proc_nr;                  /* to whom should the reply go? */
12849   int status;                   /* reply code */
12850   {
12851   /* Send a reply to a process that wanted to read or write data. */
12852
12853     message tty_mess;
12854
12855     tty_mess.m_type = code;
12856     tty_mess.REP_PROC_NR = proc_nr;
12857     tty_mess.REP_STATUS = status;
12858     if ((status = send(replyee, &tty_mess)) != OK)
12859           panic("tty_reply failed, status\n", status);
12860   }


12863   /*===========================================================================*
12864    *                              sigchar                                      *
12865    *===========================================================================*/
12866   PUBLIC void sigchar(tp, sig)
12867   register tty_t *tp;
12868   int sig;                      /* SIGINT, SIGQUIT, SIGKILL or SIGHUP */
12869   {
12870   /* Process a SIGINT, SIGQUIT or SIGKILL char from the keyboard or SIGHUP from
12871    * a tty close, "stty 0", or a real RS-232 hangup.  MM will send the signal to
12872    * the process group (INT, QUIT), all processes (KILL), or the session leader
12873    * (HUP).
12874    */
```

```
12875
12876     if (tp->tty_pgrp != 0) cause_sig(tp->tty_pgrp, sig);
12877
12878     if (!(tp->tty_termios.c_lflag & NOFLSH)) {
12879           tp->tty_incount = tp->tty_eotct = 0;    /* kill earlier input */
12880           tp->tty_intail = tp->tty_inhead;
12881           (*tp->tty_ocancel)(tp);                 /* kill all output */
12882           tp->tty_inhibited = RUNNING;
12883           tp->tty_events = 1;
12884     }
12885   }


12888   /*===========================================================================*
12889    *                              tty_icancel                                  *
12890    *===========================================================================*/
12891   PRIVATE void tty_icancel(tp)
12892   register tty_t *tp;
12893   {
12894   /* Discard all pending input, tty buffer or device. */
12895
12896     tp->tty_incount = tp->tty_eotct = 0;
12897     tp->tty_intail = tp->tty_inhead;
12898     (*tp->tty_icancel)(tp);
12899   }


12902   /*===========================================================================*
12903    *                              tty_init                                     *
12904    *===========================================================================*/
12905   PRIVATE void tty_init(tp)
12906   tty_t *tp;                    /* TTY line to initialize. */
12907   {
12908   /* Initialize tty structure and call device initialization routines. */
12909
12910     tp->tty_intail = tp->tty_inhead = tp->tty_inbuf;
12911     tp->tty_min = 1;
12912     tp->tty_termios = termios_defaults;
12913     tp->tty_icancel = tp->tty_ocancel = tp->tty_ioctl = tp->tty_close =
12914                                                             tty_devnop;
12915     if (tp < tty_addr(NR_CONS)) {
12916           scr_init(tp);
12917     } else
12918     if (tp < tty_addr(NR_CONS+NR_RS_LINES)) {
12919           rs_init(tp);
12920     } else {
12921           pty_init(tp);
12922     }
12923   }


12926   /*===========================================================================*
12927    *                              tty_wakeup                                   *
12928    *===========================================================================*/
12929   PUBLIC void tty_wakeup(now)
12930   clock_t now;                          /* current time */
12931   {
12932   /* Wake up TTY when something interesting is happening on one of the terminal
12933    * lines, like a character arriving on an RS232 line, a key being typed, or
12934    * a timer on a line expiring by TIME.
```



```
12935    */
12936     tty_t *tp;
12937
12938     /* Scan the timerlist for expired timers and compute the next timeout time. */
12939     tty_timeout = TIME_NEVER;
12940     while ((tp = tty_timelist) != NULL) {
12941           if (tp->tty_time > now) {
12942                 tty_timeout = tp->tty_time;     /* this timer is next */
12943                 break;
12944           }
12945           tp->tty_min = 0;                      /* force read to succeed */
12946           tp->tty_events = 1;
12947           tty_timelist = tp->tty_timenext;
12948     }
12949
12950     /* Let TTY know there is something afoot. */
12951     interrupt(TTY);
12952   }


12955   /*===========================================================================*
12956    *                              settimer                                     *
12957    *===========================================================================*/
12958   PRIVATE void settimer(tp, on)
12959   tty_t *tp;                    /* line to set or unset a timer on */
12960   int on;                       /* set timer if true, otherwise unset */
12961   {
12962   /* Set or unset a TIME inspired timer.  This function is interrupt sensitive
12963    * due to tty_wakeup(), so it must be called from within lock()/unlock().
12964    */
12965     tty_t **ptp;
12966
12967     /* Take tp out of the timerlist if present. */
12968     for (ptp = &tty_timelist; *ptp != NULL; ptp = &(*ptp)->tty_timenext) {
12969           if (tp == *ptp) {
12970                 *ptp = tp->tty_timenext;        /* take tp out of the list */
12971                 break;
12972           }
12973     }
12974     if (!on) return;                            /* unsetting it is enough */
12975
12976     /* Timeout occurs TIME deciseconds from now. */
12977     tp->tty_time = get_uptime() + tp->tty_termios.c_cc[VTIME] * (HZ/10);
12978
12979     /* Find a new place in the list. */
12980     for (ptp = &tty_timelist; *ptp != NULL; ptp = &(*ptp)->tty_timenext) {
12981           if (tp->tty_time <= (*ptp)->tty_time) break;
12982     }
12983     tp->tty_timenext = *ptp;
12984     *ptp = tp;
12985     if (tp->tty_time < tty_timeout) tty_timeout = tp->tty_time;
12986   }


12989   /*===========================================================================*
12990    *                              tty_devnop                                   *
12991    *===========================================================================*/
12992   PUBLIC void tty_devnop(tp)
12993   tty_t *tp;
12994   {
```

```
12995       /* Some functions need not be implemented at the device level. */
12996     }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/kernel/keyboard.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

13000     /* Keyboard driver for PC's and AT's.
13001      *
13002      * Changed by Marcus Hampel     (04/02/1994)
13003      *  - Loadable keymaps
13004      */
13005
13006     #include "kernel.h"
13007     #include <termios.h>
13008     #include <signal.h>
13009     #include <unistd.h>
13010     #include <minix/callnr.h>
13011     #include <minix/com.h>
13012     #include <minix/keymap.h>
13013     #include "tty.h"
13014     #include "keymaps/us-std.src"
13015
13016     /* Standard and AT keyboard.  (PS/2 MCA implies AT throughout.) */
13017     #define KEYBD           0x60    /* I/O port for keyboard data */
13018
13019     /* AT keyboard. */
13020     #define KB_COMMAND      0x64    /* I/O port for commands on AT */
13021     #define KB_GATE_A20     0x02    /* bit in output port to enable A20 line */
13022     #define KB_PULSE_OUTPUT 0xF0    /* base for commands to pulse output port */
13023     #define KB_RESET        0x01    /* bit in output port to reset CPU */
13024     #define KB_STATUS       0x64    /* I/O port for status on AT */
13025     #define KB_ACK          0xFA    /* keyboard ack response */
13026     #define KB_BUSY         0x02    /* status bit set when KEYBD port ready */
13027     #define LED_CODE        0xED    /* command to keyboard to set LEDs */
13028     #define MAX_KB_ACK_RETRIES 0x1000       /* max #times to wait for kb ack */
13029     #define MAX_KB_BUSY_RETRIES 0x1000      /* max #times to loop while kb busy */
13030     #define KBIT            0x80    /* bit used to ack characters to keyboard */
13031
13032     /* Miscellaneous. */
13033     #define ESC_SCAN           1    /* Reboot key when panicking */
13034     #define SLASH_SCAN        53    /* to recognize numeric slash */
13035     #define HOME_SCAN         71    /* first key on the numeric keypad */
13036     #define DEL_SCAN          83    /* DEL for use in CTRL-ALT-DEL reboot */
13037     #define CONSOLE            0    /* line number for console */
13038     #define MEMCHECK_ADR   0x472    /* address to stop memory check after reboot */
13039     #define MEMCHECK_MAG  0x1234    /* magic number to stop memory check */
13040
13041     #define kb_addr()       (&kb_lines[0])  /* there is only one keyboard */
13042     #define KB_IN_BYTES       32    /* size of keyboard input buffer */
13043
13044     PRIVATE int alt1;               /* left alt key state */
13045     PRIVATE int alt2;               /* right alt key state */
13046     PRIVATE int capslock;           /* caps lock key state */
13047     PRIVATE int esc;                /* escape scan code detected? */
13048     PRIVATE int control;            /* control key state */
13049     PRIVATE int caps_off;           /* 1 = normal position, 0 = depressed */
```



```
13050     PRIVATE int numlock;            /* number lock key state */
13051     PRIVATE int num_off;            /* 1 = normal position, 0 = depressed */
13052     PRIVATE int slock;              /* scroll lock key state */
13053     PRIVATE int slock_off;          /* 1 = normal position, 0 = depressed */
13054     PRIVATE int shift;              /* shift key state */
13055
13056     PRIVATE char numpad_map[] =
13057                     {'H', 'Y', 'A', 'B', 'D', 'C', 'V', 'U', 'G', 'S', 'T', '@'};
13058
13059     /* Keyboard structure, 1 per console. */
13060     struct kb_s {
13061       char *ihead;                  /* next free spot in input buffer */
13062       char *itail;                  /* scan code to return to TTY */
13063       int icount;                   /* # codes in buffer */
13064       char ibuf[KB_IN_BYTES];       /* input buffer */
13065     };
13066
13067     PRIVATE struct kb_s kb_lines[NR_CONS];
13068
13069     FORWARD _PROTOTYPE( int kb_ack, (void) );
13070     FORWARD _PROTOTYPE( int kb_wait, (void) );
13071     FORWARD _PROTOTYPE( int func_key, (int scode) );
13072     FORWARD _PROTOTYPE( int scan_keyboard, (void) );
13073     FORWARD _PROTOTYPE( unsigned make_break, (int scode) );
13074     FORWARD _PROTOTYPE( void set_leds, (void) );
13075     FORWARD _PROTOTYPE( int kbd_hw_int, (int irq) );
13076     FORWARD _PROTOTYPE( void kb_read, (struct tty *tp) );
13077     FORWARD _PROTOTYPE( unsigned map_key, (int scode) );
13078
13079
13080     /*==========================================================================*
13081      *                               map_key0                                   *
13082      *==========================================================================*/
13083     /* Map a scan code to an ASCII code ignoring modifiers. */
13084     #define map_key0(scode)  \
13085             ((unsigned) keymap[(scode) * MAP_COLS])
13086
13087
13088     /*==========================================================================*
13089      *                               map_key                                    *
13090      *==========================================================================*/
13091     PRIVATE unsigned map_key(scode)
13092     int scode;
13093     {
13094     /* Map a scan code to an ASCII code. */
13095
13096       int caps, column;
13097       u16_t *keyrow;
13098
13099       if (scode == SLASH_SCAN && esc) return '/';   /* don't map numeric slash */
13100
13101       keyrow = &keymap[scode * MAP_COLS];
13102
13103       caps = shift;
13104       if (numlock && HOME_SCAN <= scode && scode <= DEL_SCAN) caps = !caps;
13105       if (capslock && (keyrow[0] & HASCAPS)) caps = !caps;
13106
13107       if (alt1 || alt2) {
13108             column = 2;
13109             if (control || alt2) column = 3;        /* Ctrl + Alt1 == Alt2 */
```

```
13110           if (caps) column = 4;
13111     } else {
13112           column = 0;
13113           if (caps) column = 1;
13114           if (control) column = 5;
13115     }
13116     return keyrow[column] & ~HASCAPS;
13117   }


13120   /*==========================================================================*
13121    *                              kbd_hw_int                                  *
13122    *==========================================================================*/
13123   PRIVATE int kbd_hw_int(irq)
13124   int irq;
13125   {
13126   /* A keyboard interrupt has occurred.  Process it. */
13127
13128     int code;
13129     unsigned km;
13130     register struct kb_s *kb;
13131
13132     /* Fetch the character from the keyboard hardware and acknowledge it. */
13133     code = scan_keyboard();
13134
13135     /* The IBM keyboard interrupts twice per key, once when depressed, once when
13136      * released.  Filter out the latter, ignoring all but the shift-type keys.
13137      * The shift-type keys 29, 42, 54, 56, 58, and 69 must be processed normally.
13138      */
13139
13140     if (code & 0200) {
13141           /* A key has been released (high bit is set). */
13142           km = map_key0(code & 0177);
13143           if (km != CTRL && km != SHIFT && km != ALT && km != CALOCK
13144                           && km != NLOCK && km != SLOCK && km != EXTKEY)
13145                   return 1;
13146     }
13147
13148     /* Store the character in memory so the task can get at it later. */
13149     kb = kb_addr();
13150     if (kb->icount < KB_IN_BYTES) {
13151           *kb->ihead++ = code;
13152           if (kb->ihead == kb->ibuf + KB_IN_BYTES) kb->ihead = kb->ibuf;
13153           kb->icount++;
13154           tty_table[current].tty_events = 1;
13155           force_timeout();
13156     }
13157     /* Else it doesn't fit - discard it. */
13158     return 1;     /* Reenable keyboard interrupt */
13159   }


13162   /*==========================================================================*
13163    *                              kb_read                                     *
13164    *==========================================================================*/
13165   PRIVATE void kb_read(tp)
13166   tty_t *tp;
13167   {
13168   /* Process characters from the circular keyboard buffer. */
13169
```



```
13170     struct kb_s *kb;
13171     char buf[3];
13172     int scode;
13173     unsigned ch;
13174
13175     kb = kb_addr();
13176     tp = &tty_table[current];             /* always use the current console */
13177
13178     while (kb->icount > 0) {
13179           scode = *kb->itail++;                   /* take one key scan code */
13180           if (kb->itail == kb->ibuf + KB_IN_BYTES) kb->itail = kb->ibuf;
13181           lock();
13182           kb->icount--;
13183           unlock();
13184
13185           /* Function keys are being used for debug dumps. */
13186           if (func_key(scode)) continue;
13187
13188           /* Perform make/break processing. */
13189           ch = make_break(scode);
13190
13191           if (ch <= 0xFF) {
13192                   /* A normal character. */
13193                   buf[0] = ch;
13194                   (void) in_process(tp, buf, 1);
13195           } else
13196           if (HOME <= ch && ch <= INSRT) {
13197                   /* An ASCII escape sequence generated by the numeric pad. */
13198                   buf[0] = ESC;
13199                   buf[1] = '[';
13200                   buf[2] = numpad_map[ch - HOME];
13201                   (void) in_process(tp, buf, 3);
13202           } else
13203           if (ch == ALEFT) {
13204                   /* Choose lower numbered console as current console. */
13205                   select_console(current - 1);
13206           } else
13207           if (ch == ARIGHT) {
13208                   /* Choose higher numbered console as current console. */
13209                   select_console(current + 1);
13210           } else
13211           if (AF1 <= ch && ch <= AF12) {
13212                   /* Alt-F1 is console, Alt-F2 is ttyc1, etc. */
13213                   select_console(ch - AF1);
13214           }
13215     }
13216   }


13219   /*==========================================================================*
13220    *                              make_break                                  *
13221    *==========================================================================*/
13222   PRIVATE unsigned make_break(scode)
13223   int scode;                    /* scan code of key just struck or released */
13224   {
13225   /* This routine can handle keyboards that interrupt only on key depression,
13226    * as well as keyboards that interrupt on key depression and key release.
13227    * For efficiency, the interrupt routine filters out most key releases.
13228    */
13229     int ch, make;
```

```
13230     static int CAD_count = 0;
13231
13232     /* Check for CTRL-ALT-DEL, and if found, halt the computer. This would
13233      * be better done in keyboard() in case TTY is hung, except control and
13234      * alt are set in the high level code.
13235      */
13236     if (control && (alt1 || alt2) && scode == DEL_SCAN)
13237     {
13238           if (++CAD_count == 3) wreboot(RBT_HALT);
13239           cause_sig(INIT_PROC_NR, SIGABRT);
13240           return -1;
13241     }
13242
13243     /* High-order bit set on key release. */
13244     make = (scode & 0200 ? 0 : 1);        /* 0 = release, 1 = press */
13245
13246     ch = map_key(scode & 0177);           /* map to ASCII */
13247
13248     switch (ch) {
13249           case CTRL:
13250                 control = make;
13251                 ch = -1;
13252                 break;
13253           case SHIFT:
13254                 shift = make;
13255                 ch = -1;
13256                 break;
13257           case ALT:
13258                 if (make) {
13259                       if (esc) alt2 = 1; else alt1 = 1;
13260                 } else {
13261                       alt1 = alt2 = 0;
13262                 }
13263                 ch = -1;
13264                 break;
13265           case CALOCK:
13266                 if (make && caps_off) {
13267                       capslock = 1 - capslock;
13268                       set_leds();
13269                 }
13270                 caps_off = 1 - make;
13271                 ch = -1;
13272                 break;
13273           case NLOCK:
13274                 if (make && num_off) {
13275                       numlock = 1 - numlock;
13276                       set_leds();
13277                 }
13278                 num_off = 1 - make;
13279                 ch = -1;
13280                 break;
13281           case SLOCK:
13282                 if (make & slock_off) {
13283                       slock = 1 - slock;
13284                       set_leds();
13285                 }
13286                 slock_off = 1 - make;
13287                 ch = -1;
13288                 break;
13289           case EXTKEY:
```



```
13290                 esc = 1;
13291                 return(-1);
13292           default:
13293                 if (!make) ch = -1;
13294     }
13295     esc = 0;
13296     return(ch);
13297   }


13300   /*==========================================================================*
13301    *                              set_leds                                    *
13302    *==========================================================================*/
13303   PRIVATE void set_leds()
13304   {
13305   /* Set the LEDs on the caps lock and num lock keys */
13306
13307     unsigned leds;
13308
13309     if (!pc_at) return;   /* PC/XT doesn't have LEDs */
13310
13311     /* encode LED bits */
13312     leds = (slock << 0) | (numlock << 1) | (capslock << 2);
13313
13314     kb_wait();                    /* wait for buffer empty  */
13315     out_byte(KEYBD, LED_CODE);    /* prepare keyboard to accept LED values */
13316     kb_ack();                     /* wait for ack response  */
13317
13318     kb_wait();                    /* wait for buffer empty  */
13319     out_byte(KEYBD, leds);        /* give keyboard LED values */
13320     kb_ack();                     /* wait for ack response  */
13321   }


13324   /*==========================================================================*
13325    *                              kb_wait                                     *
13326    *==========================================================================*/
13327   PRIVATE int kb_wait()
13328   {
13329   /* Wait until the controller is ready; return zero if this times out. */
13330
13331     int retries;
13332
13333     retries = MAX_KB_BUSY_RETRIES + 1;
13334     while (--retries != 0 && in_byte(KB_STATUS) & KB_BUSY)
13335           ;                       /* wait until not busy */
13336     return(retries);              /* nonzero if ready */
13337   }


13340   /*==========================================================================*
13341    *                              kb_ack                                      *
13342    *==========================================================================*/
13343   PRIVATE int kb_ack()
13344   {
13345   /* Wait until kbd acknowledges last command; return zero if this times out. */
13346
13347     int retries;
13348
13349     retries = MAX_KB_ACK_RETRIES + 1;
```

```
13350     while (--retries != 0 && in_byte(KEYBD) != KB_ACK)
13351           ;                       /* wait for ack */
13352     return(retries);              /* nonzero if ack received */
13353   }


13356   /*===========================================================================*
13357    *                              kb_init                                      *
13358    *===========================================================================*/
13359   PUBLIC void kb_init(tp)
13360   tty_t *tp;
13361   {
13362   /* Initialize the keyboard driver. */
13363
13364     register struct kb_s *kb;
13365
13366     /* Input function. */
13367     tp->tty_devread = kb_read;
13368
13369     kb = kb_addr();
13370
13371     /* Set up input queue. */
13372     kb->ihead = kb->itail = kb->ibuf;
13373
13374     /* Set initial values. */
13375     caps_off = 1;
13376     num_off = 1;
13377     slock_off = 1;
13378     esc = 0;
13379
13380     set_leds();                   /* turn off numlock led */
13381
13382     scan_keyboard();              /* stop lockup from leftover keystroke */
13383
13384     put_irq_handler(KEYBOARD_IRQ, kbd_hw_int);    /* set the interrupt handler */
13385     enable_irq(KEYBOARD_IRQ);     /* safe now everything initialised! */
13386   }


13389   /*===========================================================================*
13390    *                              kbd_loadmap                                  *
13391    *===========================================================================*/
13392   PUBLIC int kbd_loadmap(user_phys)
13393   phys_bytes user_phys;
13394   {
13395   /* Load a new keymap. */
13396
13397     phys_copy(user_phys, vir2phys(keymap), (phys_bytes) sizeof(keymap));
13398     return(OK);
13399   }


13402   /*===========================================================================*
13403    *                              func_key                                     *
13404    *===========================================================================*/
13405   PRIVATE int func_key(scode)
13406   int scode;                      /* scan code for a function key */
13407   {
13408   /* This procedure traps function keys for debugging and control purposes. */
13409
```



```
13410     unsigned code;
13411
13412     code = map_key0(scode);                       /* first ignore modifiers */
13413     if (code < F1 || code > F12) return(FALSE);   /* not our job */
13414
13415     switch (map_key(scode)) {                     /* include modifiers */
13416
13417     case F1:      p_dmp(); break;         /* print process table */
13418     case F2:      map_dmp(); break;       /* print memory map */
13419     case F3:      toggle_scroll(); break; /* hardware vs. software scrolling */
13420     case CF7:     sigchar(&tty_table[CONSOLE], SIGQUIT); break;
13421     case CF8:     sigchar(&tty_table[CONSOLE], SIGINT); break;
13422     case CF9:     sigchar(&tty_table[CONSOLE], SIGKILL); break;
13423     default:      return(FALSE);
13424     }
13425     return(TRUE);
13426   }


13429   /*===========================================================================*
13430    *                              scan_keyboard                                *
13431    *===========================================================================*/
13432   PRIVATE int scan_keyboard()
13433   {
13434   /* Fetch the character from the keyboard hardware and acknowledge it. */
13435
13436     int code;
13437     int val;
13438
13439     code = in_byte(KEYBD);        /* get the scan code for the key struck */
13440     val = in_byte(PORT_B);        /* strobe the keyboard to ack the char */
13441     out_byte(PORT_B, val | KBIT); /* strobe the bit high */
13442     out_byte(PORT_B, val);        /* now strobe it low */
13443     return code;
13444   }


13447   /*===========================================================================*
13448    *                              wreboot                                      *
13449    *===========================================================================*/
13450   PUBLIC void wreboot(how)
13451   int how;                /* 0 = halt, 1 = reboot, 2 = panic!, ... */
13452   {
13453   /* Wait for keystrokes for printing debugging info and reboot. */
13454
13455     int quiet, code;
13456     static u16_t magic = MEMCHECK_MAG;
13457     struct tasktab *ttp;
13458
13459     /* Mask all interrupts. */
13460     out_byte(INT_CTLMASK, ~0);
13461
13462     /* Tell several tasks to stop. */
13463     cons_stop();
13464     floppy_stop();
13465     clock_stop();
13466
13467     if (how == RBT_HALT) {
13468           printf("System Halted\n");
13469           if (!mon_return) how = RBT_PANIC;
```

```
13470     }
13471
13472     if (how == RBT_PANIC) {
13473           /* A panic! */
13474           printf("Hit ESC to reboot, F-keys for debug dumps\n");
13475
13476           (void) scan_keyboard(); /* ack any old input */
13477           quiet = scan_keyboard();/* quiescent value (0 on PC, last code on AT)*/
13478           for (;;) {
13479                   milli_delay(100);       /* pause for a decisecond */
13480                   code = scan_keyboard();
13481                   if (code != quiet) {
13482                           /* A key has been pressed. */
13483                           if (code == ESC_SCAN) break; /* reboot if ESC typed */
13484                           (void) func_key(code);       /* process function key */
13485                           quiet = scan_keyboard();
13486                   }
13487           }
13488           how = RBT_REBOOT;
13489     }
13490
13491     if (how == RBT_REBOOT) printf("Rebooting\n");
13492
13493     if (mon_return && how != RBT_RESET) {
13494           /* Reinitialize the interrupt controllers to the BIOS defaults. */
13495           intr_init(0);
13496           out_byte(INT_CTLMASK, 0);
13497           out_byte(INT2_CTLMASK, 0);
13498
13499           /* Return to the boot monitor. */
13500           if (how == RBT_HALT) {
13501                   reboot_code = vir2phys("");
13502           } else
13503           if (how == RBT_REBOOT) {
13504                   reboot_code = vir2phys("delay;boot");
13505           }
13506           level0(monitor);
13507     }
13508
13509     /* Stop BIOS memory test. */
13510     phys_copy(vir2phys(&magic), (phys_bytes) MEMCHECK_ADR,
13511                                                 (phys_bytes) sizeof(magic));
13512
13513     if (protected_mode) {
13514           /* Use the AT keyboard controller to reset the processor.
13515            * The A20 line is kept enabled in case this code is ever
13516            * run from extended memory, and because some machines
13517            * appear to drive the fake A20 high instead of low just
13518            * after reset, leading to an illegal opode trap.  This bug
13519            * is more of a problem if the fake A20 is in use, as it
13520            * would be if the keyboard reset were used for real mode.
13521            */
13522           kb_wait();
13523           out_byte(KB_COMMAND,
13524                    KB_PULSE_OUTPUT | (0x0F & ~(KB_GATE_A20 | KB_RESET)));
13525           milli_delay(10);
13526
13527           /* If the nice method fails then do a reset.  In protected
13528            * mode this means a processor shutdown.
13529            */
```



```
13530           printf("Hard reset...\n");
13531           milli_delay(250);
13532      }
13533      /* In real mode, jumping to the reset address is good enough. */
13534      level0(reset);
13535    }
```

++
src/kernel/console.c
++

```
13600     /* Code and data for the IBM console driver.
13601      *
13602      * The 6845 video controller used by the IBM PC shares its video memory with
13603      * the CPU somewhere in the 0xB0000 memory bank.  To the 6845 this memory
13604      * consists of 16-bit words.  Each word has a character code in the low byte
13605      * and a so-called attribute byte in the high byte.  The CPU directly modifies
13606      * video memory to display characters, and sets two registers on the 6845 that
13607      * specify the video origin and the cursor position.  The video origin is the
13608      * place in video memory where the first character (upper left corner) can
13609      * be found.  Moving the origin is a fast way to scroll the screen.  Some
13610      * video adapters wrap around the top of video memory, so the origin can
13611      * move without bounds.  For other adapters screen memory must sometimes be
13612      * moved to reset the origin.  All computations on video memory use character
13613      * (word) addresses for simplicity and assume there is no wrapping.  The
13614      * assembly support functions translate the word addresses to byte addresses
13615      * and the scrolling function worries about wrapping.
13616      */
13617
13618     #include "kernel.h"
13619     #include <termios.h>
13620     #include <minix/callnr.h>
13621     #include <minix/com.h>
13622     #include "protect.h"
13623     #include "tty.h"
13624     #include "proc.h"
13625
13626     /* Definitions used by the console driver. */
13627     #define MONO_BASE    0xB0000L   /* base of mono video memory */
13628     #define COLOR_BASE   0xB8000L   /* base of color video memory */
13629     #define MONO_SIZE     0x1000    /* 4K mono video memory */
13630     #define COLOR_SIZE    0x4000    /* 16K color video memory */
13631     #define EGA_SIZE      0x8000    /* EGA & VGA have at least 32K */
13632     #define BLANK_COLOR   0x0700    /* determines cursor color on blank screen */
13633     #define SCROLL_UP          0    /* scroll forward */
13634     #define SCROLL_DOWN        1    /* scroll backward */
13635     #define BLANK_MEM ((u16_t *) 0) /* tells mem_vid_copy() to blank the screen */
13636     #define CONS_RAM_WORDS    80    /* video ram buffer size */
13637     #define MAX_ESC_PARMS      2    /* number of escape sequence params allowed */
13638
13639     /* Constants relating to the controller chips. */
13640     #define M_6845         0x3B4    /* port for 6845 mono */
13641     #define C_6845         0x3D4    /* port for 6845 color */
13642     #define EGA            0x3C4    /* port for EGA or VGA card */
13643     #define INDEX              0    /* 6845's index register */
13644     #define DATA               1    /* 6845's data register */
```

```
13645   #define VID_ORG           12    /* 6845's origin register */
13646   #define CURSOR            14    /* 6845's cursor register */
13647
13648   /* Beeper. */
13649   #define BEEP_FREQ     0x0533    /* value to put into timer to set beep freq */
13650   #define B_TIME             3    /* length of CTRL-G beep is ticks */
13651
13652   /* definitions used for font management */
13653   #define GA_SEQUENCER_INDEX      0x3C4
13654   #define GA_SEQUENCER_DATA       0x3C5
13655   #define GA_GRAPHICS_INDEX       0x3CE
13656   #define GA_GRAPHICS_DATA        0x3CF
13657   #define GA_VIDEO_ADDRESS        0xA0000L
13658   #define GA_FONT_SIZE            8192
13659
13660   /* Global variables used by the console driver. */
13661   PUBLIC unsigned vid_seg;        /* video ram selector (0xB0000 or 0xB8000) */
13662   PUBLIC unsigned vid_size;       /* 0x2000 for color or 0x0800 for mono */
13663   PUBLIC unsigned vid_mask;       /* 0x1FFF for color or 0x07FF for mono */
13664   PUBLIC unsigned blank_color = BLANK_COLOR; /* display code for blank */
13665
13666   /* Private variables used by the console driver. */
13667   PRIVATE int vid_port;           /* I/O port for accessing 6845 */
13668   PRIVATE int wrap;               /* hardware can wrap? */
13669   PRIVATE int softscroll;         /* 1 = software scrolling, 0 = hardware */
13670   PRIVATE unsigned vid_base;      /* base of video ram (0xB000 or 0xB800) */
13671   PRIVATE int beeping;            /* speaker is beeping? */
13672   #define scr_width       80      /* # characters on a line */
13673   #define scr_lines       25      /* # lines on the screen */
13674   #define scr_size        (80*25) /* # characters on the screen */
13675
13676   /* Per console data. */
13677   typedef struct console {
13678     tty_t *c_tty;                 /* associated TTY struct */
13679     int c_column;                 /* current column number (0-origin) */
13680     int c_row;                    /* current row (0 at top of screen) */
13681     int c_rwords;                 /* number of WORDS (not bytes) in outqueue */
13682     unsigned c_start;             /* start of video memory of this console */
13683     unsigned c_limit;             /* limit of this console's video memory */
13684     unsigned c_org;               /* location in RAM where 6845 base points */
13685     unsigned c_cur;               /* current position of cursor in video RAM */
13686     unsigned c_attr;              /* character attribute */
13687     unsigned c_blank;             /* blank attribute */
13688     char c_esc_state;             /* 0=normal, 1=ESC, 2=ESC[ */
13689     char c_esc_intro;             /* Distinguishing character following ESC */
13690     int *c_esc_parmp;             /* pointer to current escape parameter */
13691     int c_esc_parmv[MAX_ESC_PARMS];       /* list of escape parameters */
13692     u16_t c_ramqueue[CONS_RAM_WORDS];     /* buffer for video RAM */
13693   } console_t;
13694
13695   PRIVATE int nr_cons= 1;         /* actual number of consoles */
13696   PRIVATE console_t cons_table[NR_CONS];
13697   PRIVATE console_t *curcons;     /* currently visible */
13698
13699   /* Color if using a color controller. */
13700   #define color   (vid_port == C_6845)
13701
13702   /* Map from ANSI colors to the attributes used by the PC */
13703   PRIVATE int ansi_colors[8] = {0, 4, 2, 6, 1, 5, 3, 7};
13704
```



```
13705   /* Structure used for font management */
13706   struct sequence {
13707           unsigned short index;
13708           unsigned char port;
13709           unsigned char value;
13710   };
13711
13712   FORWARD _PROTOTYPE( void cons_write, (struct tty *tp)                   );
13713   FORWARD _PROTOTYPE( void cons_echo, (tty_t *tp, int c)                  );
13714   FORWARD _PROTOTYPE( void out_char, (console_t *cons, int c)             );
13715   FORWARD _PROTOTYPE( void beep, (void)                                   );
13716   FORWARD _PROTOTYPE( void do_escape, (console_t *cons, int c)            );
13717   FORWARD _PROTOTYPE( void flush, (console_t *cons)                       );
13718   FORWARD _PROTOTYPE( void parse_escape, (console_t *cons, int c)         );
13719   FORWARD _PROTOTYPE( void scroll_screen, (console_t *cons, int dir)      );
13720   FORWARD _PROTOTYPE( void set_6845, (int reg, unsigned val)              );
13721   FORWARD _PROTOTYPE( void stop_beep, (void)                              );
13722   FORWARD _PROTOTYPE( void cons_org0, (void)                              );
13723   FORWARD _PROTOTYPE( void ga_program, (struct sequence *seq) );
13724
13725
13726   /*===========================================================================*
13727    *                              cons_write                                   *
13728    *===========================================================================*/
13729   PRIVATE void cons_write(tp)
13730   register struct tty *tp;        /* tells which terminal is to be used */
13731   {
13732   /* Copy as much data as possible to the output queue, then start I/O.  On
13733    * memory-mapped terminals, such as the IBM console, the I/O will also be
13734    * finished, and the counts updated.  Keep repeating until all I/O done.
13735    */
13736
13737     int count;
13738     register char *tbuf;
13739     char buf[64];
13740     phys_bytes user_phys;
13741     console_t *cons = tp->tty_priv;
13742
13743     /* Check quickly for nothing to do, so this can be called often without
13744      * unmodular tests elsewhere.
13745      */
13746     if ((count = tp->tty_outleft) == 0 || tp->tty_inhibited) return;
13747
13748     /* Copy the user bytes to buf[] for decent addressing. Loop over the
13749      * copies, since the user buffer may be much larger than buf[].
13750      */
13751     do {
13752           if (count > sizeof(buf)) count = sizeof(buf);
13753           user_phys = proc_vir2phys(proc_addr(tp->tty_outproc), tp->tty_out_vir);
13754           phys_copy(user_phys, vir2phys(buf), (phys_bytes) count);
13755           tbuf = buf;
13756
13757           /* Update terminal data structure. */
13758           tp->tty_out_vir += count;
13759           tp->tty_outcum += count;
13760           tp->tty_outleft -= count;
13761
13762           /* Output each byte of the copy to the screen.  Avoid calling
13763            * out_char() for the "easy" characters, put them into the buffer
13764            * directly.
```

```
13765            */
13766           do {
13767                   if ((unsigned) *tbuf < ' ' || cons->c_esc_state > 0
13768                           || cons->c_column >= scr_width
13769                           || cons->c_rwords >= buflen(cons->c_ramqueue))
13770                   {
13771                           out_char(cons, *tbuf++);
13772                   } else {
13773                           cons->c_ramqueue[cons->c_rwords++] =
13774                                           cons->c_attr | (*tbuf++ & BYTE);
13775                           cons->c_column++;
13776                   }
13777           } while (--count != 0);
13778   } while ((count = tp->tty_outleft) != 0 && !tp->tty_inhibited);
13779
13780   flush(cons);                    /* transfer anything buffered to the screen */
13781
13782   /* Reply to the writer if all output is finished. */
13783   if (tp->tty_outleft == 0) {
13784         tty_reply(tp->tty_outrepcode, tp->tty_outcaller, tp->tty_outproc,
13785                                                         tp->tty_outcum);
13786         tp->tty_outcum = 0;
13787   }
13788   }


13791   /*==========================================================================*
13792    *                              cons_echo                                   *
13793    *==========================================================================*/
13794   PRIVATE void cons_echo(tp, c)
13795   register tty_t *tp;             /* pointer to tty struct */
13796   int c;                          /* character to be echoed */
13797   {
13798   /* Echo keyboard input (print & flush). */
13799     console_t *cons = tp->tty_priv;
13800
13801     out_char(cons, c);
13802     flush(cons);
13803   }


13806   /*==========================================================================*
13807    *                              out_char                                    *
13808    *==========================================================================*/
13809   PRIVATE void out_char(cons, c)
13810   register console_t *cons;       /* pointer to console struct */
13811   int c;                          /* character to be output */
13812   {
13813   /* Output a character on the console.  Check for escape sequences first. */
13814     if (cons->c_esc_state > 0) {
13815           parse_escape(cons, c);
13816           return;
13817     }
13818
13819     switch(c) {
13820           case 000:               /* null is typically used for padding */
13821                   return;         /* better not do anything */
13822
13823           case 007:               /* ring the bell */
13824                   flush(cons);    /* print any chars queued for output */
```



```
13825                   beep();
13826                   return;
13827
13828           case '\b':              /* backspace */
13829                   if (--cons->c_column < 0) {
13830                           if (--cons->c_row >= 0) cons->c_column += scr_width;
13831                   }
13832                   flush(cons);
13833                   return;
13834
13835           case '\n':              /* line feed */
13836                   if ((cons->c_tty->tty_termios.c_oflag & (OPOST|ONLCR))
13837                                                   == (OPOST|ONLCR)) {
13838                           cons->c_column = 0;
13839                   }
13840                   /*FALL THROUGH*/
13841           case 013:               /* CTRL-K */
13842           case 014:               /* CTRL-L */
13843                   if (cons->c_row == scr_lines-1) {
13844                           scroll_screen(cons, SCROLL_UP);
13845                   } else {
13846                           cons->c_row++;
13847                   }
13848                   flush(cons);
13849                   return;
13850
13851           case '\r':              /* carriage return */
13852                   cons->c_column = 0;
13853                   flush(cons);
13854                   return;
13855
13856           case '\t':              /* tab */
13857                   cons->c_column = (cons->c_column + TAB_SIZE) & ~TAB_MASK;
13858                   if (cons->c_column > scr_width) {
13859                           cons->c_column -= scr_width;
13860                           if (cons->c_row == scr_lines-1) {
13861                                   scroll_screen(cons, SCROLL_UP);
13862                           } else {
13863                                   cons->c_row++;
13864                           }
13865                   }
13866                   flush(cons);
13867                   return;
13868
13869           case 033:               /* ESC - start of an escape sequence */
13870                   flush(cons);    /* print any chars queued for output */
13871                   cons->c_esc_state = 1;  /* mark ESC as seen */
13872                   return;
13873
13874           default:                /* printable chars are stored in ramqueue */
13875                   if (cons->c_column >= scr_width) {
13876                           if (!LINEWRAP) return;
13877                           if (cons->c_row == scr_lines-1) {
13878                                   scroll_screen(cons, SCROLL_UP);
13879                           } else {
13880                                   cons->c_row++;
13881                           }
13882                           cons->c_column = 0;
13883                           flush(cons);
13884                   }
```

```
13885                   if (cons->c_rwords == buflen(cons->c_ramqueue)) flush(cons);
13886                   cons->c_ramqueue[cons->c_rwords++] = cons->c_attr | (c & BYTE);
13887                   cons->c_column++;                       /* next column */
13888                   return;
13889           }
13890   }


13893   /*==========================================================================*
13894    *                              scroll_screen                               *
13895    *==========================================================================*/
13896   PRIVATE void scroll_screen(cons, dir)
13897   register console_t *cons;       /* pointer to console struct */
13898   int dir;                        /* SCROLL_UP or SCROLL_DOWN */
13899   {
13900     unsigned new_line, new_org, chars;
13901
13902     flush(cons);
13903     chars = scr_size - scr_width;         /* one screen minus one line */
13904
13905     /* Scrolling the screen is a real nuisance due to the various incompatible
13906      * video cards.  This driver supports software scrolling (Hercules?),
13907      * hardware scrolling (mono and CGA cards) and hardware scrolling without
13908      * wrapping (EGA and VGA cards).  In the latter case we must make sure that
13909      *           c_start <= c_org && c_org + scr_size <= c_limit
13910      * holds, because EGA and VGA don't wrap around the end of video memory.
13911      */
13912     if (dir == SCROLL_UP) {
13913           /* Scroll one line up in 3 ways: soft, avoid wrap, use origin. */
13914           if (softscroll) {
13915                   vid_vid_copy(cons->c_start + scr_width, cons->c_start, chars);
13916           } else
13917           if (!wrap && cons->c_org + scr_size + scr_width >= cons->c_limit) {
13918                   vid_vid_copy(cons->c_org + scr_width, cons->c_start, chars);
13919                   cons->c_org = cons->c_start;
13920           } else {
13921                   cons->c_org = (cons->c_org + scr_width) & vid_mask;
13922           }
13923           new_line = (cons->c_org + chars) & vid_mask;
13924     } else {
13925           /* Scroll one line down in 3 ways: soft, avoid wrap, use origin. */
13926           if (softscroll) {
13927                   vid_vid_copy(cons->c_start, cons->c_start + scr_width, chars);
13928           } else
13929           if (!wrap && cons->c_org < cons->c_start + scr_width) {
13930                   new_org = cons->c_limit - scr_size;
13931                   vid_vid_copy(cons->c_org, new_org + scr_width, chars);
13932                   cons->c_org = new_org;
13933           } else {
13934                   cons->c_org = (cons->c_org - scr_width) & vid_mask;
13935           }
13936           new_line = cons->c_org;
13937     }
13938     /* Blank the new line at top or bottom. */
13939     blank_color = cons->c_blank;
13940     mem_vid_copy(BLANK_MEM, new_line, scr_width);
13941
13942     /* Set the new video origin. */
13943     if (cons == curcons) set_6845(VID_ORG, cons->c_org);
13944     flush(cons);
```



```
13945   }


13948   /*==========================================================================*
13949    *                              flush                                       *
13950    *==========================================================================*/
13951   PRIVATE void flush(cons)
13952   register console_t *cons;       /* pointer to console struct */
13953   {
13954   /* Send characters buffered in 'ramqueue' to screen memory, check the new
13955    * cursor position, compute the new hardware cursor position and set it.
13956    */
13957     unsigned cur;
13958     tty_t *tp = cons->c_tty;
13959
13960     /* Have the characters in 'ramqueue' transferred to the screen. */
13961     if (cons->c_rwords > 0) {
13962           mem_vid_copy(cons->c_ramqueue, cons->c_cur, cons->c_rwords);
13963           cons->c_rwords = 0;
13964
13965           /* TTY likes to know the current column and if echoing messed up. */
13966           tp->tty_position = cons->c_column;
13967           tp->tty_reprint = TRUE;
13968     }
13969
13970     /* Check and update the cursor position. */
13971     if (cons->c_column < 0) cons->c_column = 0;
13972     if (cons->c_column > scr_width) cons->c_column = scr_width;
13973     if (cons->c_row < 0) cons->c_row = 0;
13974     if (cons->c_row >= scr_lines) cons->c_row = scr_lines - 1;
13975     cur = cons->c_org + cons->c_row * scr_width + cons->c_column;
13976     if (cur != cons->c_cur) {
13977           if (cons == curcons) set_6845(CURSOR, cur);
13978           cons->c_cur = cur;
13979     }
13980   }


13983   /*==========================================================================*
13984    *                              parse_escape                                *
13985    *==========================================================================*/
13986   PRIVATE void parse_escape(cons, c)
13987   register console_t *cons;       /* pointer to console struct */
13988   char c;                         /* next character in escape sequence */
13989   {
13990   /* The following ANSI escape sequences are currently supported.
13991    * If n and/or m are omitted, they default to 1. Omitted s defaults to 0.
13992    *   ESC [nA moves up n lines
13993    *   ESC [nB moves down n lines
13994    *   ESC [nC moves right n spaces
13995    *   ESC [nD moves left n spaces
13996    *   ESC [m;nH moves cursor to (m,n)
13997    *   ESC [sJ clears screen relative to cursor (0 to end, 1 from start, 2 all)
13998    *   ESC [sK clears line relative to cursor (0 to end, 1 from start, 2 all)
13999    *   ESC [nL inserts n lines at cursor
14000    *   ESC [nM deletes n lines at cursor
14001    *   ESC [nP deletes n chars at cursor
14002    *   ESC [n@ inserts n chars at cursor
14003    *   ESC [nm enables rendition n (0= normal, 1=bold, 4=underline, 5=blinking,
14004    *        7=reverse, 30..37 set foreground color, 40..47 set background color)
```

```
14005     *   ESC M scrolls the screen backwards if the cursor is on the top line
14006     */
14007
14008      switch (cons->c_esc_state) {
14009        case 1:                     /* ESC seen */
14010            cons->c_esc_intro = '\0';
14011            cons->c_esc_parmp = cons->c_esc_parmv;
14012            cons->c_esc_parmv[0] = cons->c_esc_parmv[1] = 0;
14013            switch (c) {
14014                case '[':   /* Control Sequence Introducer */
14015                    cons->c_esc_intro = c;
14016                    cons->c_esc_state = 2;
14017                    break;
14018                case 'M':   /* Reverse Index */
14019                    do_escape(cons, c);
14020                    break;
14021                default:
14022                    cons->c_esc_state = 0;
14023            }
14024            break;
14025
14026        case 2:                     /* ESC [ seen */
14027            if (c >= '0' && c <= '9') {
14028                    if (cons->c_esc_parmp < bufend(cons->c_esc_parmv))
14029                            *cons->c_esc_parmp = *cons->c_esc_parmp * 10 + (c-'0');
14030            } else
14031            if (c == ';') {
14032                    if (++cons->c_esc_parmp < bufend(cons->c_esc_parmv))
14033                            *cons->c_esc_parmp = 0;
14034            } else {
14035                    do_escape(cons, c);
14036            }
14037            break;
14038      }
14039    }


14042    /*===========================================================================*
14043     *                              do_escape                                    *
14044     *===========================================================================*/
14045    PRIVATE void do_escape(cons, c)
14046    register console_t *cons;       /* pointer to console struct */
14047    char c;                         /* next character in escape sequence */
14048    {
14049      int value, n;
14050      unsigned src, dst, count;
14051
14052      /* Some of these things hack on screen RAM, so it had better be up to date */
14053      flush(cons);
14054
14055      if (cons->c_esc_intro == '\0') {
14056            /* Handle a sequence beginning with just ESC */
14057            switch (c) {
14058                case 'M':           /* Reverse Index */
14059                    if (cons->c_row == 0) {
14060                            scroll_screen(cons, SCROLL_DOWN);
14061                    } else {
14062                            cons->c_row--;
14063                    }
14064                    flush(cons);
```



```
14065                    break;
14066
14067                default: break;
14068            }
14069      } else
14070      if (cons->c_esc_intro == '[') {
14071            /* Handle a sequence beginning with ESC [ and parameters */
14072            value = cons->c_esc_parmv[0];
14073            switch (c) {
14074                case 'A':           /* ESC [nA moves up n lines */
14075                    n = (value == 0 ? 1 : value);
14076                    cons->c_row -= n;
14077                    flush(cons);
14078                    break;
14079
14080                case 'B':           /* ESC [nB moves down n lines */
14081                    n = (value == 0 ? 1 : value);
14082                    cons->c_row += n;
14083                    flush(cons);
14084                    break;
14085
14086                case 'C':           /* ESC [nC moves right n spaces */
14087                    n = (value == 0 ? 1 : value);
14088                    cons->c_column += n;
14089                    flush(cons);
14090                    break;
14091
14092                case 'D':           /* ESC [nD moves left n spaces */
14093                    n = (value == 0 ? 1 : value);
14094                    cons->c_column -= n;
14095                    flush(cons);
14096                    break;
14097
14098                case 'H':           /* ESC [m;nH" moves cursor to (m,n) */
14099                    cons->c_row = cons->c_esc_parmv[0] - 1;
14100                    cons->c_column = cons->c_esc_parmv[1] - 1;
14101                    flush(cons);
14102                    break;
14103
14104                case 'J':           /* ESC [sJ clears in display */
14105                    switch (value) {
14106                        case 0:     /* Clear from cursor to end of screen */
14107                            count = scr_size - (cons->c_cur - cons->c_org);
14108                            dst = cons->c_cur;
14109                            break;
14110                        case 1:     /* Clear from start of screen to cursor */
14111                            count = cons->c_cur - cons->c_org;
14112                            dst = cons->c_org;
14113                            break;
14114                        case 2:     /* Clear entire screen */
14115                            count = scr_size;
14116                            dst = cons->c_org;
14117                            break;
14118                        default:    /* Do nothing */
14119                            count = 0;
14120                            dst = cons->c_org;
14121                    }
14122                    blank_color = cons->c_blank;
14123                    mem_vid_copy(BLANK_MEM, dst, count);
14124                    break;
```

```
14125
14126                 case 'K':           /* ESC [sK clears line from cursor */
14127                     switch (value) {
14128                         case 0:     /* Clear from cursor to end of line */
14129                             count = scr_width - cons->c_column;
14130                             dst = cons->c_cur;
14131                             break;
14132                         case 1:     /* Clear from beginning of line to cursor */
14133                             count = cons->c_column;
14134                             dst = cons->c_cur - cons->c_column;
14135                             break;
14136                         case 2:     /* Clear entire line */
14137                             count = scr_width;
14138                             dst = cons->c_cur - cons->c_column;
14139                             break;
14140                         default:    /* Do nothing */
14141                             count = 0;
14142                             dst = cons->c_cur;
14143                     }
14144                     blank_color = cons->c_blank;
14145                     mem_vid_copy(BLANK_MEM, dst, count);
14146                     break;
14147
14148                 case 'L':           /* ESC [nL inserts n lines at cursor */
14149                     n = value;
14150                     if (n < 1) n = 1;
14151                     if (n > (scr_lines - cons->c_row))
14152                             n = scr_lines - cons->c_row;
14153
14154                     src = cons->c_org + cons->c_row * scr_width;
14155                     dst = src + n * scr_width;
14156                     count = (scr_lines - cons->c_row - n) * scr_width;
14157                     vid_vid_copy(src, dst, count);
14158                     blank_color = cons->c_blank;
14159                     mem_vid_copy(BLANK_MEM, src, n * scr_width);
14160                     break;
14161
14162                 case 'M':           /* ESC [nM deletes n lines at cursor */
14163                     n = value;
14164                     if (n < 1) n = 1;
14165                     if (n > (scr_lines - cons->c_row))
14166                             n = scr_lines - cons->c_row;
14167
14168                     dst = cons->c_org + cons->c_row * scr_width;
14169                     src = dst + n * scr_width;
14170                     count = (scr_lines - cons->c_row - n) * scr_width;
14171                     vid_vid_copy(src, dst, count);
14172                     blank_color = cons->c_blank;
14173                     mem_vid_copy(BLANK_MEM, dst + count, n * scr_width);
14174                     break;
14175
14176                 case '@':           /* ESC [n@ inserts n chars at cursor */
14177                     n = value;
14178                     if (n < 1) n = 1;
14179                     if (n > (scr_width - cons->c_column))
14180                             n = scr_width - cons->c_column;
14181
14182                     src = cons->c_cur;
14183                     dst = src + n;
14184                     count = scr_width - cons->c_column - n;
```



```
14185                     vid_vid_copy(src, dst, count);
14186                     blank_color = cons->c_blank;
14187                     mem_vid_copy(BLANK_MEM, src, n);
14188                     break;
14189
14190                 case 'P':           /* ESC [nP deletes n chars at cursor */
14191                     n = value;
14192                     if (n < 1) n = 1;
14193                     if (n > (scr_width - cons->c_column))
14194                             n = scr_width - cons->c_column;
14195
14196                     dst = cons->c_cur;
14197                     src = dst + n;
14198                     count = scr_width - cons->c_column - n;
14199                     vid_vid_copy(src, dst, count);
14200                     blank_color = cons->c_blank;
14201                     mem_vid_copy(BLANK_MEM, dst + count, n);
14202                     break;
14203
14204                 case 'm':           /* ESC [nm enables rendition n */
14205                     switch (value) {
14206                         case 1:     /* BOLD  */
14207                             if (color) {
14208                                     /* Can't do bold, so use yellow */
14209                                     cons->c_attr = (cons->c_attr & 0xf0ff) | 0x0E00;
14210                             } else {
14211                                     /* Set intensity bit */
14212                                     cons->c_attr |= 0x0800;
14213                             }
14214                             break;
14215
14216                         case 4:     /* UNDERLINE */
14217                             if (color) {
14218                                     /* Use light green */
14219                                     cons->c_attr = (cons->c_attr & 0xf0ff) | 0x0A00;
14220                             } else {
14221                                     cons->c_attr = (cons->c_attr & 0x8900);
14222                             }
14223                             break;
14224
14225                         case 5:     /* BLINKING */
14226                             if (color) {
14227                                     /* Use magenta */
14228                                     cons->c_attr = (cons->c_attr & 0xf0ff) | 0x0500;
14229                             } else {
14230                                     /* Set the blink bit */
14231                                     cons->c_attr |= 0x8000;
14232                             }
14233                             break;
14234
14235                         case 7:     /* REVERSE */
14236                             if (color) {
14237                                     /* Swap fg and bg colors */
14238                                     cons->c_attr =
14239                                             ((cons->c_attr & 0xf000) >> 4) |
14240                                             ((cons->c_attr & 0x0f00) << 4);
14241                             } else
14242                             if ((cons->c_attr & 0x7000) == 0) {
14243                                     cons->c_attr = (cons->c_attr & 0x8800) | 0x7000;
14244                             } else {
```

```
14245                                   cons->c_attr = (cons->c_attr & 0x8800) | 0x0700;
14246                           }
14247                           break;
14248
14249                       default:    /* COLOR */
14250                           if (30 <= value && value <= 37) {
14251                                   cons->c_attr =
14252                                           (cons->c_attr & 0xf0ff) |
14253                                           (ansi_colors[(value - 30)] << 8);
14254                                   cons->c_blank =
14255                                           (cons->c_blank & 0xf0ff) |
14256                                           (ansi_colors[(value - 30)] << 8);
14257                           } else
14258                           if (40 <= value && value <= 47) {
14259                                   cons->c_attr =
14260                                           (cons->c_attr & 0x0fff) |
14261                                           (ansi_colors[(value - 40)] << 12);
14262                                   cons->c_blank =
14263                                           (cons->c_blank & 0x0fff) |
14264                                           (ansi_colors[(value - 40)] << 12);
14265                           } else {
14266                                   cons->c_attr = cons->c_blank;
14267                           }
14268                           break;
14269                   }
14270                   break;
14271           }
14272     }
14273     cons->c_esc_state = 0;
14274   }


14277   /*==========================================================================*
14278    *                              set_6845                                    *
14279    *==========================================================================*/
14280   PRIVATE void set_6845(reg, val)
14281   int reg;                        /* which register pair to set */
14282   unsigned val;                   /* 16-bit value to set it to */
14283   {
14284   /* Set a register pair inside the 6845.
14285    * Registers 12-13 tell the 6845 where in video ram to start
14286    * Registers 14-15 tell the 6845 where to put the cursor
14287    */
14288     lock();                       /* try to stop h/w loading in-between value */
14289     out_byte(vid_port + INDEX, reg);              /* set the index register */
14290     out_byte(vid_port + DATA, (val>>8) & BYTE);   /* output high byte */
14291     out_byte(vid_port + INDEX, reg + 1);          /* again */
14292     out_byte(vid_port + DATA, val&BYTE);          /* output low byte */
14293     unlock();
14294   }


14297   /*==========================================================================*
14298    *                              beep                                        *
14299    *==========================================================================*/
14300   PRIVATE void beep()
14301   {
14302   /* Making a beeping sound on the speaker (output for CRTL-G).
14303    * This routine works by turning on the bits 0 and 1 in port B of the 8255
14304    * chip that drives the speaker.
```



```
14305    */
14306
14307     message mess;
14308
14309     if (beeping) return;
14310     out_byte(TIMER_MODE, 0xB6);   /* set up timer channel 2 (square wave) */
14311     out_byte(TIMER2, BEEP_FREQ & BYTE);   /* load low-order bits of frequency */
14312     out_byte(TIMER2, (BEEP_FREQ >> 8) & BYTE);    /* now high-order bits */
14313     lock();                       /* guard PORT_B from keyboard intr handler */
14314     out_byte(PORT_B, in_byte(PORT_B) | 3);        /* turn on beep bits */
14315     unlock();
14316     beeping = TRUE;
14317
14318     mess.m_type = SET_ALARM;
14319     mess.CLOCK_PROC_NR = TTY;
14320     mess.DELTA_TICKS = B_TIME;
14321     mess.FUNC_TO_CALL = (sighandler_t) stop_beep;
14322     sendrec(CLOCK, &mess);
14323   }


14326   /*==========================================================================*
14327    *                              stop_beep                                   *
14328    *==========================================================================*/
14329   PRIVATE void stop_beep()
14330   {
14331   /* Turn off the beeper by turning off bits 0 and 1 in PORT_B. */
14332
14333     lock();                       /* guard PORT_B from keyboard intr handler */
14334     out_byte(PORT_B, in_byte(PORT_B) & ~3);
14335     beeping = FALSE;
14336     unlock();
14337   }


14340   /*==========================================================================*
14341    *                              scr_init                                    *
14342    *==========================================================================*/
14343   PUBLIC void scr_init(tp)
14344   tty_t *tp;
14345   {
14346   /* Initialize the screen driver. */
14347     console_t *cons;
14348     phys_bytes vid_base;
14349     u16_t bios_crtbase;
14350     int line;
14351     unsigned page_size;
14352
14353     /* Associate console and TTY. */
14354     line = tp - &tty_table[0];
14355     if (line >= nr_cons) return;
14356     cons = &cons_table[line];
14357     cons->c_tty = tp;
14358     tp->tty_priv = cons;
14359
14360     /* Initialize the keyboard driver. */
14361     kb_init(tp);
14362
14363     /* Output functions. */
14364     tp->tty_devwrite = cons_write;
```

```
14365     tp->tty_echo = cons_echo;
14366
14367     /* Get the BIOS parameters that tells the VDU I/O base register. */
14368     phys_copy(0x463L, vir2phys(&bios_crtbase), 2L);
14369
14370     vid_port = bios_crtbase;
14371
14372     if (color) {
14373           vid_base = COLOR_BASE;
14374           vid_size = COLOR_SIZE;
14375     } else {
14376           vid_base = MONO_BASE;
14377           vid_size = MONO_SIZE;
14378     }
14379     if (ega) vid_size = EGA_SIZE; /* for both EGA and VGA */
14380     wrap = !ega;
14381
14382     vid_seg = protected_mode ? VIDEO_SELECTOR : physb_to_hclick(vid_base);
14383     init_dataseg(&gdt[VIDEO_INDEX], vid_base, (phys_bytes) vid_size,
14384                                                     TASK_PRIVILEGE);
14385     vid_size >>= 1;               /* word count */
14386     vid_mask = vid_size - 1;
14387
14388     /* There can be as many consoles as video memory allows. */
14389     nr_cons = vid_size / scr_size;
14390     if (nr_cons > NR_CONS) nr_cons = NR_CONS;
14391     if (nr_cons > 1) wrap = 0;
14392     page_size = vid_size / nr_cons;
14393     cons->c_start = line * page_size;
14394     cons->c_limit = cons->c_start + page_size;
14395     cons->c_org = cons->c_start;
14396     cons->c_attr = cons->c_blank = BLANK_COLOR;
14397
14398     /* Clear the screen. */
14399     blank_color = BLANK_COLOR;
14400     mem_vid_copy(BLANK_MEM, cons->c_start, scr_size);
14401     select_console(0);
14402   }


14405   /*===========================================================================*
14406    *                              putk                                         *
14407    *===========================================================================*/
14408   PUBLIC void putk(c)
14409   int c;                        /* character to print */
14410   {
14411   /* This procedure is used by the version of printf() that is linked with
14412    * the kernel itself.  The one in the library sends a message to FS, which is
14413    * not what is needed for printing within the kernel.  This version just queues
14414    * the character and starts the output.
14415    */
14416
14417     if (c != 0) {
14418           if (c == '\n') putk('\r');
14419           out_char(&cons_table[0], (int) c);
14420     } else {
14421           flush(&cons_table[0]);
14422     }
14423   }
```



```
14426   /*===========================================================================*
14427    *                              toggle_scroll                                *
14428    *===========================================================================*/
14429   PUBLIC void toggle_scroll()
14430   {
14431   /* Toggle between hardware and software scroll. */
14432
14433     cons_org0();
14434     softscroll = !softscroll;
14435     printf("%sware scrolling enabled.\n", softscroll ? "Soft" : "Hard");
14436   }


14439   /*===========================================================================*
14440    *                              cons_stop                                    *
14441    *===========================================================================*/
14442   PUBLIC void cons_stop()
14443   {
14444   /* Prepare for halt or reboot. */
14445
14446     cons_org0();
14447     softscroll = 1;
14448     select_console(0);
14449     cons_table[0].c_attr = cons_table[0].c_blank = BLANK_COLOR;
14450   }


14453   /*===========================================================================*
14454    *                              cons_org0                                    *
14455    *===========================================================================*/
14456   PRIVATE void cons_org0()
14457   {
14458   /* Scroll video memory back to put the origin at 0. */
14459
14460     int cons_line;
14461     console_t *cons;
14462     unsigned n;
14463
14464     for (cons_line = 0; cons_line < nr_cons; cons_line++) {
14465           cons = &cons_table[cons_line];
14466           while (cons->c_org > cons->c_start) {
14467                 n = vid_size - scr_size;        /* amount of unused memory */
14468                 if (n > cons->c_org - cons->c_start)
14469                       n = cons->c_org - cons->c_start;
14470                 vid_vid_copy(cons->c_org, cons->c_org - n, scr_size);
14471                 cons->c_org -= n;
14472           }
14473           flush(cons);
14474     }
14475     select_console(current);
14476   }


14479   /*===========================================================================*
14480    *                              select_console                               *
14481    *===========================================================================*/
14482   PUBLIC void select_console(int cons_line)
14483   {
14484   /* Set the current console to console number 'cons_line'. */
```

```
14485
14486     if (cons_line < 0 || cons_line >= nr_cons) return;
14487     current = cons_line;
14488     curcons = &cons_table[cons_line];
14489     set_6845(VID_ORG, curcons->c_org);
14490     set_6845(CURSOR, curcons->c_cur);
14491   }


14494   /*==========================================================================*
14495    *                              con_loadfont                                *
14496    *==========================================================================*/
14497   PUBLIC int con_loadfont(user_phys)
14498   phys_bytes user_phys;
14499   {
14500   /* Load a font into the EGA or VGA adapter. */
14501
14502     static struct sequence seq1[7] = {
14503           { GA_SEQUENCER_INDEX, 0x00, 0x01 },
14504           { GA_SEQUENCER_INDEX, 0x02, 0x04 },
14505           { GA_SEQUENCER_INDEX, 0x04, 0x07 },
14506           { GA_SEQUENCER_INDEX, 0x00, 0x03 },
14507           { GA_GRAPHICS_INDEX, 0x04, 0x02 },
14508           { GA_GRAPHICS_INDEX, 0x05, 0x00 },
14509           { GA_GRAPHICS_INDEX, 0x06, 0x00 },
14510     };
14511     static struct sequence seq2[7] = {
14512           { GA_SEQUENCER_INDEX, 0x00, 0x01 },
14513           { GA_SEQUENCER_INDEX, 0x02, 0x03 },
14514           { GA_SEQUENCER_INDEX, 0x04, 0x03 },
14515           { GA_SEQUENCER_INDEX, 0x00, 0x03 },
14516           { GA_GRAPHICS_INDEX, 0x04, 0x00 },
14517           { GA_GRAPHICS_INDEX, 0x05, 0x10 },
14518           { GA_GRAPHICS_INDEX, 0x06,    0 },
14519     };
14520
14521     seq2[6].value= color ? 0x0E : 0x0A;
14522
14523     if (!ega) return(ENOTTY);
14524
14525     lock();
14526     ga_program(seq1);     /* bring font memory into view */
14527
14528     phys_copy(user_phys, (phys_bytes)GA_VIDEO_ADDRESS, (phys_bytes)GA_FONT_SIZE);
14529
14530     ga_program(seq2);     /* restore */
14531     unlock();
14532
14533     return(OK);
14534   }


14537   /*==========================================================================*
14538    *                              ga_program                                  *
14539    *==========================================================================*/
14540   PRIVATE void ga_program(seq)
14541   struct sequence *seq;
14542   {
14543   /* support function for con_loadfont */
14544
```



```
14545     int len= 7;
14546     do {
14547           out_byte(seq->index, seq->port);
14548           out_byte(seq->index+1, seq->value);
14549           seq++;
14550     } while (--len > 0);
14551   }



++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                  src/kernel/dmp.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

14600   /* This file contains some dumping routines for debugging. */
14601
14602   #include "kernel.h"
14603   #include <minix/com.h>
14604   #include "proc.h"
14605
14606   char *vargv;
14607
14608   FORWARD _PROTOTYPE(char *proc_name, (int proc_nr));
14609
14610   /*==========================================================================*
14611    *                                p_dmp                                     *
14612    *==========================================================================*/
14613   PUBLIC void p_dmp()
14614   {
14615   /* Proc table dump */
14616
14617     register struct proc *rp;
14618     static struct proc *oldrp = BEG_PROC_ADDR;
14619     int n = 0;
14620     phys_clicks text, data, size;
14621     int proc_nr;
14622
14623     printf("\n--pid --pc- ---sp- flag -user --sys-- -text- -data- -size- -recv- command\n");
14624
14625     for (rp = oldrp; rp < END_PROC_ADDR; rp++) {
14626           proc_nr = proc_number(rp);
14627           if (rp->p_flags & P_SLOT_FREE) continue;
14628           if (++n > 20) break;
14629           text = rp->p_map[T].mem_phys;
14630           data = rp->p_map[D].mem_phys;
14631           size = rp->p_map[T].mem_len
14632                   + ((rp->p_map[S].mem_phys + rp->p_map[S].mem_len) - data);
14633           printf("%5d %5lx %6lx %2x %7U %7U %5uK %5uK %5uK ",
14634                  proc_nr < 0 ? proc_nr : rp->p_pid,
14635                  (unsigned long) rp->p_reg.pc,
14636                  (unsigned long) rp->p_reg.sp,
14637                  rp->p_flags,
14638                  rp->user_time, rp->sys_time,
14639                  click_to_round_k(text), click_to_round_k(data),
14640                  click_to_round_k(size));
14641           if (rp->p_flags & RECEIVING) {
14642                   printf("%-7.7s", proc_name(rp->p_getfrom));
14643           } else
14644           if (rp->p_flags & SENDING) {
```

```
14645                   printf("S:%-5.5s", proc_name(rp->p_sendto));
14646           } else
14647           if (rp->p_flags == 0) {
14648                   printf("       ");
14649           }
14650           printf("%s\n", rp->p_name);
14651     }
14652     if (rp == END_PROC_ADDR) rp = BEG_PROC_ADDR; else printf("--more--\r");
14653     oldrp = rp;
14654   }


14657   /*===========================================================================*
14658    *                                map_dmp                                    *
14659    *===========================================================================*/
14660   PUBLIC void map_dmp()
14661   {
14662     register struct proc *rp;
14663     static struct proc *oldrp = cproc_addr(HARDWARE);
14664     int n = 0;
14665     phys_clicks size;
14666
14667     printf("\nPROC NAME-  -----TEXT-----  -----DATA-----  ----STACK-----  -SIZE-\n");
14668     for (rp = oldrp; rp < END_PROC_ADDR; rp++) {
14669           if (rp->p_flags & P_SLOT_FREE) continue;
14670           if (++n > 20) break;
14671           size = rp->p_map[T].mem_len
14672                   + ((rp->p_map[S].mem_phys + rp->p_map[S].mem_len)
14673                                           - rp->p_map[D].mem_phys);
14674           printf("%3d %-6.6s  %4x %4x %4x  %4x %4x %4x  %4x %4x %4x  %5uK\n",
14675                   proc_number(rp),
14676                   rp->p_name,
14677                   rp->p_map[T].mem_vir, rp->p_map[T].mem_phys, rp->p_map[T].mem_len,
14678                   rp->p_map[D].mem_vir, rp->p_map[D].mem_phys, rp->p_map[D].mem_len,
14679                   rp->p_map[S].mem_vir, rp->p_map[S].mem_phys, rp->p_map[S].mem_len,
14680                   click_to_round_k(size));
14681     }
14682     if (rp == END_PROC_ADDR) rp = cproc_addr(HARDWARE); else printf("--more--\r");
14683     oldrp = rp;
14684   }


14687   /*===========================================================================*
14688    *                                proc_name                                  *
14689    *===========================================================================*/
14690   PRIVATE char *proc_name(proc_nr)
14691   int proc_nr;
14692   {
14693     if (proc_nr == ANY) return "ANY";
14694     return proc_addr(proc_nr)->p_name;
14695   }
```



```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/kernel/system.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

14700   /* This task handles the interface between file system and kernel as well as
14701    * between memory manager and kernel.  System services are obtained by sending
14702    * sys_task() a message specifying what is needed.  To make life easier for
14703    * MM and FS, a library is provided with routines whose names are of the
14704    * form sys_xxx, e.g. sys_xit sends the SYS_XIT message to sys_task.  The
14705    * message types and parameters are:
14706    *
14707    *   SYS_FORK    informs kernel that a process has forked
14708    *   SYS_NEWMAP  allows MM to set up a process memory map
14709    *   SYS_GETMAP  allows MM to get a process' memory map
14710    *   SYS_EXEC    sets program counter and stack pointer after EXEC
14711    *   SYS_XIT     informs kernel that a process has exited
14712    *   SYS_GETSP   caller wants to read out some process' stack pointer
14713    *   SYS_TIMES   caller wants to get accounting times for a process
14714    *   SYS_ABORT   MM or FS cannot go on; abort MINIX
14715    *   SYS_FRESH   start with a fresh process image during EXEC (68000 only)
14716    *   SYS_SENDSIG send a signal to a process (POSIX style)
14717    *   SYS_SIGRETURN complete POSIX-style signalling
14718    *   SYS_KILL    cause a signal to be sent via MM
14719    *   SYS_ENDSIG  finish up after SYS_KILL-type signal
14720    *   SYS_COPY    request a block of data to be copied between processes
14721    *   SYS_VCOPY   request a series of data blocks to be copied between procs
14722    *   SYS_GBOOT   copies the boot parameters to a process
14723    *   SYS_MEM     returns the next free chunk of physical memory
14724    *   SYS_UMAP    compute the physical address for a given virtual address
14725    *   SYS_TRACE   request a trace operation
14726    *
14727    * Message types and parameters:
14728    *
14729    *    m_type       PROC1     PROC2      PID     MEM_PTR
14730    * ------------------------------------------------------
14731    * | SYS_FORK   | parent  |  child  |   pid   |         |
14732    * |------------+---------+---------+---------+---------|
14733    * | SYS_NEWMAP | proc nr |         |         | map ptr |
14734    * |------------+---------+---------+---------+---------|
14735    * | SYS_EXEC   | proc nr | traced  | new sp  |         |
14736    * |------------+---------+---------+---------+---------|
14737    * | SYS_XIT    | parent  | exitee  |         |         |
14738    * |------------+---------+---------+---------+---------|
14739    * | SYS_GETSP  | proc nr |         |         |         |
14740    * |------------+---------+---------+---------+---------|
14741    * | SYS_TIMES  | proc nr |         | buf ptr |         |
14742    * |------------+---------+---------+---------+---------|
14743    * | SYS_ABORT  |         |         |         |         |
14744    * |------------+---------+---------+---------+---------|
14745    * | SYS_FRESH  | proc nr | data_cl |         |         |
14746    * |------------+---------+---------+---------+---------|
14747    * | SYS_GBOOT  | proc nr |         |         | bootptr |
14748    * |------------+---------+---------+---------+---------|
14749    * | SYS_GETMAP | proc nr |         |         | map ptr |
14750    * ------------------------------------------------------
14751    *
14752    *    m_type          m1_i1     m1_i2     m1_i3       m1_p1
14753    * ----------------+---------+---------+---------+--------------
14754    * | SYS_VCOPY     |  src p  |  dst p  | vec siz | vc addr     |
```

```
14755     * |---------------+---------+---------+---------+-------------|
14756     * | SYS_SENDSIG   | proc nr |         |         | smp         |
14757     * |---------------+---------+---------+---------+-------------|
14758     * | SYS_SIGRETURN | proc nr |         |         | scp         |
14759     * |---------------+---------+---------+---------+-------------|
14760     * | SYS_ENDSIG    | proc nr |         |         |             |
14761     * ------------------------------------------------------------
14762     *
14763     *    m_type       m2_i1     m2_i2     m2_l1     m2_l2
14764     * ------------------------------------------------------
14765     * | SYS_TRACE  | proc_nr | request |  addr   |  data   |
14766     * ------------------------------------------------------
14767     *
14768     *
14769     *    m_type       m6_i1     m6_i2     m6_i3     m6_f1
14770     * ------------------------------------------------------
14771     * | SYS_KILL   | proc_nr  |  sig    |         |         |
14772     * ------------------------------------------------------
14773     *
14774     *
14775     *    m_type      m5_c1   m5_i1    m5_l1   m5_c2   m5_i2    m5_l2   m5_l3
14776     * ------------------------------------------------------------------------
14777     * | SYS_COPY   |src seg|src proc|src vir|dst seg|dst proc|dst vir| byte ct |
14778     * ------------------------------------------------------------------------
14779     * | SYS_UMAP   |  seg  |proc nr |vir adr|       |        |       | byte ct |
14780     * ------------------------------------------------------------------------
14781     *
14782     *
14783     *    m_type      m1_i1      m1_i2      m1_i3
14784     * |------------+----------+----------+----------
14785     * | SYS_MEM    | mem base | mem size | tot mem |
14786     * ---------------------------------------------
14787     *
14788     * In addition to the main sys_task() entry point, there are 5 other minor
14789     * entry points:
14790     *   cause_sig: take action to cause a signal to occur, sooner or later
14791     *   inform:    tell MM about pending signals
14792     *   numap:     umap D segment starting from process number instead of pointer
14793     *   umap:      compute the physical address for a given virtual address
14794     *   alloc_segments: allocate segments for 8088 or higher processor
14795     */
14796
14797    #include "kernel.h"
14798    #include <signal.h>
14799    #include <unistd.h>
14800    #include <sys/sigcontext.h>
14801    #include <sys/ptrace.h>
14802    #include <minix/boot.h>
14803    #include <minix/callnr.h>
14804    #include <minix/com.h>
14805    #include "proc.h"
14806    #include "protect.h"
14807
14808    /* PSW masks. */
14809    #define IF_MASK 0x00000200
14810    #define IOPL_MASK 0x003000
14811
14812    PRIVATE message m;
14813
14814    FORWARD _PROTOTYPE( int do_abort, (message *m_ptr) );
```



```
14815    FORWARD _PROTOTYPE( int do_copy, (message *m_ptr) );
14816    FORWARD _PROTOTYPE( int do_exec, (message *m_ptr) );
14817    FORWARD _PROTOTYPE( int do_fork, (message *m_ptr) );
14818    FORWARD _PROTOTYPE( int do_gboot, (message *m_ptr) );
14819    FORWARD _PROTOTYPE( int do_getsp, (message *m_ptr) );
14820    FORWARD _PROTOTYPE( int do_kill, (message *m_ptr) );
14821    FORWARD _PROTOTYPE( int do_mem, (message *m_ptr) );
14822    FORWARD _PROTOTYPE( int do_newmap, (message *m_ptr) );
14823    FORWARD _PROTOTYPE( int do_sendsig, (message *m_ptr) );
14824    FORWARD _PROTOTYPE( int do_sigreturn, (message *m_ptr) );
14825    FORWARD _PROTOTYPE( int do_endsig, (message *m_ptr) );
14826    FORWARD _PROTOTYPE( int do_times, (message *m_ptr) );
14827    FORWARD _PROTOTYPE( int do_trace, (message *m_ptr) );
14828    FORWARD _PROTOTYPE( int do_umap, (message *m_ptr) );
14829    FORWARD _PROTOTYPE( int do_xit, (message *m_ptr) );
14830    FORWARD _PROTOTYPE( int do_vcopy, (message *m_ptr) );
14831    FORWARD _PROTOTYPE( int do_getmap, (message *m_ptr) );
14832
14833
14834    /*===========================================================================*
14835     *                                sys_task                                   *
14836     *===========================================================================*/
14837    PUBLIC void sys_task()
14838    {
14839    /* Main entry point of sys_task.  Get the message and dispatch on type. */
14840
14841      register int r;
14842
14843      while (TRUE) {
14844            receive(ANY, &m);
14845
14846            switch (m.m_type) {     /* which system call */
14847                case SYS_FORK:      r = do_fork(&m);        break;
14848                case SYS_NEWMAP:    r = do_newmap(&m);      break;
14849                case SYS_GETMAP:    r = do_getmap(&m);      break;
14850                case SYS_EXEC:      r = do_exec(&m);        break;
14851                case SYS_XIT:       r = do_xit(&m);         break;
14852                case SYS_GETSP:     r = do_getsp(&m);       break;
14853                case SYS_TIMES:     r = do_times(&m);       break;
14854                case SYS_ABORT:     r = do_abort(&m);       break;
14855                case SYS_SENDSIG:   r = do_sendsig(&m);     break;
14856                case SYS_SIGRETURN: r = do_sigreturn(&m);   break;
14857                case SYS_KILL:      r = do_kill(&m);        break;
14858                case SYS_ENDSIG:    r = do_endsig(&m);      break;
14859                case SYS_COPY:      r = do_copy(&m);        break;
14860                case SYS_VCOPY:     r = do_vcopy(&m);       break;
14861                case SYS_GBOOT:     r = do_gboot(&m);       break;
14862                case SYS_MEM:       r = do_mem(&m);         break;
14863                case SYS_UMAP:      r = do_umap(&m);        break;
14864                case SYS_TRACE:     r = do_trace(&m);       break;
14865                default:            r = E_BAD_FCN;
14866            }
14867
14868            m.m_type = r;           /* 'r' reports status of call */
14869            send(m.m_source, &m);   /* send reply to caller */
14870      }
14871    }
14872
14873
14874    /*===========================================================================*
```

```
14875    *                              do_fork                                  *
14876    *===========================================================================*/
14877   PRIVATE int do_fork(m_ptr)
14878   register message *m_ptr;        /* pointer to request message */
14879   {
14880   /* Handle sys_fork().  m_ptr->PROC1 has forked.  The child is m_ptr->PROC2. */
14881
14882     reg_t old_ldt_sel;
14883     register struct proc *rpc;
14884     struct proc *rpp;
14885
14886     if (!isoksusern(m_ptr->PROC1) || !isoksusern(m_ptr->PROC2))
14887           return(E_BAD_PROC);
14888     rpp = proc_addr(m_ptr->PROC1);
14889     rpc = proc_addr(m_ptr->PROC2);
14890
14891     /* Copy parent 'proc' struct to child. */
14892     old_ldt_sel = rpc->p_ldt_sel; /* stop this being obliterated by copy */
14893
14894     *rpc = *rpp;                  /* copy 'proc' struct */
14895
14896     rpc->p_ldt_sel = old_ldt_sel;
14897     rpc->p_nr = m_ptr->PROC2;     /* this was obliterated by copy */
14898
14899     rpc->p_flags |= NO_MAP;       /* inhibit the process from running */
14900
14901     rpc->p_flags &= ~(PENDING | SIG_PENDING | P_STOP);
14902
14903     /* Only 1 in group should have PENDING, child does not inherit trace status*/
14904     sigemptyset(&rpc->p_pending);
14905     rpc->p_pendcount = 0;
14906     rpc->p_pid = m_ptr->PID;      /* install child's pid */
14907     rpc->p_reg.retreg = 0;        /* child sees pid = 0 to know it is child */
14908
14909     rpc->user_time = 0;           /* set all the accounting times to 0 */
14910     rpc->sys_time = 0;
14911     rpc->child_utime = 0;
14912     rpc->child_stime = 0;
14913
14914     return(OK);
14915   }


14918   /*===========================================================================*
14919    *                              do_newmap                                *
14920    *===========================================================================*/
14921   PRIVATE int do_newmap(m_ptr)
14922   message *m_ptr;                 /* pointer to request message */
14923   {
14924   /* Handle sys_newmap().  Fetch the memory map from MM. */
14925
14926     register struct proc *rp;
14927     phys_bytes src_phys;
14928     int caller;                   /* whose space has the new map (usually MM) */
14929     int k;                        /* process whose map is to be loaded */
14930     int old_flags;                /* value of flags before modification */
14931     struct mem_map *map_ptr;      /* virtual address of map inside caller (MM) */
14932
14933     /* Extract message parameters and copy new memory map from MM. */
14934     caller = m_ptr->m_source;
```



```
14935     k = m_ptr->PROC1;
14936     map_ptr = (struct mem_map *) m_ptr->MEM_PTR;
14937     if (!isokprocn(k)) return(E_BAD_PROC);
14938     rp = proc_addr(k);            /* ptr to entry of user getting new map */
14939
14940     /* Copy the map from MM. */
14941     src_phys = umap(proc_addr(caller), D, (vir_bytes) map_ptr, sizeof(rp->p_map));
14942     if (src_phys == 0) panic("bad call to sys_newmap", NO_NUM);
14943     phys_copy(src_phys, vir2phys(rp->p_map), (phys_bytes) sizeof(rp->p_map));
14944
14945     alloc_segments(rp);
14946     old_flags = rp->p_flags;      /* save the previous value of the flags */
14947     rp->p_flags &= ~NO_MAP;
14948     if (old_flags != 0 && rp->p_flags == 0) lock_ready(rp);
14949
14950     return(OK);
14951   }


14954   /*===========================================================================*
14955    *                              do_getmap                                *
14956    *===========================================================================*/
14957   PRIVATE int do_getmap(m_ptr)
14958   message *m_ptr;                 /* pointer to request message */
14959   {
14960   /* Handle sys_getmap().  Report the memory map to MM. */
14961
14962     register struct proc *rp;
14963     phys_bytes dst_phys;
14964     int caller;                   /* where the map has to be stored */
14965     int k;                        /* process whose map is to be loaded */
14966     struct mem_map *map_ptr;      /* virtual address of map inside caller (MM) */
14967
14968     /* Extract message parameters and copy new memory map to MM. */
14969     caller = m_ptr->m_source;
14970     k = m_ptr->PROC1;
14971     map_ptr = (struct mem_map *) m_ptr->MEM_PTR;
14972
14973     if (!isokprocn(k))
14974           panic("do_getmap got bad proc: ", m_ptr->PROC1);
14975
14976     rp = proc_addr(k);            /* ptr to entry of the map */
14977
14978     /* Copy the map to MM. */
14979     dst_phys = umap(proc_addr(caller), D, (vir_bytes) map_ptr, sizeof(rp->p_map));
14980     if (dst_phys == 0) panic("bad call to sys_getmap", NO_NUM);
14981     phys_copy(vir2phys(rp->p_map), dst_phys, sizeof(rp->p_map));
14982
14983     return(OK);
14984   }


14987   /*===========================================================================*
14988    *                              do_exec                                  *
14989    *===========================================================================*/
14990   PRIVATE int do_exec(m_ptr)
14991   register message *m_ptr;        /* pointer to request message */
14992   {
14993   /* Handle sys_exec().  A process has done a successful EXEC. Patch it up. */
14994
```

```
14995     register struct proc *rp;
14996     reg_t sp;                   /* new sp */
14997     phys_bytes phys_name;
14998     char *np;
14999   #define NLEN (sizeof(rp->p_name)-1)
15000
15001     if (!isoksusern(m_ptr->PROC1)) return E_BAD_PROC;
15002     /* PROC2 field is used as flag to indicate process is being traced */
15003     if (m_ptr->PROC2) cause_sig(m_ptr->PROC1, SIGTRAP);
15004     sp = (reg_t) m_ptr->STACK_PTR;
15005     rp = proc_addr(m_ptr->PROC1);
15006     rp->p_reg.sp = sp;          /* set the stack pointer */
15007     rp->p_reg.pc = (reg_t) m_ptr->IP_PTR; /* set pc */
15008     rp->p_alarm = 0;            /* reset alarm timer */
15009     rp->p_flags &= ~RECEIVING;  /* MM does not reply to EXEC call */
15010     if (rp->p_flags == 0) lock_ready(rp);
15011
15012     /* Save command name for debugging, ps(1) output, etc. */
15013     phys_name = numap(m_ptr->m_source, (vir_bytes) m_ptr->NAME_PTR,
15014                                                 (vir_bytes) NLEN);
15015     if (phys_name != 0) {
15016           phys_copy(phys_name, vir2phys(rp->p_name), (phys_bytes) NLEN);
15017           for (np = rp->p_name; (*np & BYTE) >= ' '; np++) {}
15018           *np = 0;
15019     }
15020     return(OK);
15021   }


15024   /*===========================================================================*
15025    *                              do_xit                                       *
15026    *===========================================================================*/
15027   PRIVATE int do_xit(m_ptr)
15028   message *m_ptr;               /* pointer to request message */
15029   {
15030   /* Handle sys_xit().  A process has exited. */
15031
15032     register struct proc *rp, *rc;
15033     struct proc *np, *xp;
15034     int parent;                 /* number of exiting proc's parent */
15035     int proc_nr;                /* number of process doing the exit */
15036     phys_clicks base, size;
15037
15038     parent = m_ptr->PROC1;      /* slot number of parent process */
15039     proc_nr = m_ptr->PROC2;     /* slot number of exiting process */
15040     if (!isoksusern(parent) || !isoksusern(proc_nr)) return(E_BAD_PROC);
15041     rp = proc_addr(parent);
15042     rc = proc_addr(proc_nr);
15043     lock();
15044     rp->child_utime += rc->user_time + rc->child_utime;   /* accum child times */
15045     rp->child_stime += rc->sys_time + rc->child_stime;
15046     unlock();
15047     rc->p_alarm = 0;            /* turn off alarm timer */
15048     if (rc->p_flags == 0) lock_unready(rc);
15049
15050     strcpy(rc->p_name, "<noname>");       /* process no longer has a name */
15051
15052     /* If the process being terminated happens to be queued trying to send a
15053      * message (i.e., the process was killed by a signal, rather than it doing an
15054      * EXIT), then it must be removed from the message queues.
```



```
15055      */
15056     if (rc->p_flags & SENDING) {
15057           /* Check all proc slots to see if the exiting process is queued. */
15058           for (rp = BEG_PROC_ADDR; rp < END_PROC_ADDR; rp++) {
15059                   if (rp->p_callerq == NIL_PROC) continue;
15060                   if (rp->p_callerq == rc) {
15061                           /* Exiting process is on front of this queue. */
15062                           rp->p_callerq = rc->p_sendlink;
15063                           break;
15064                   } else {
15065                           /* See if exiting process is in middle of queue. */
15066                           np = rp->p_callerq;
15067                           while ( ( xp = np->p_sendlink) != NIL_PROC)
15068                                   if (xp == rc) {
15069                                           np->p_sendlink = xp->p_sendlink;
15070                                           break;
15071                                   } else {
15072                                           np = xp;
15073                                   }
15074                   }
15075           }
15076     }
15077
15078     if (rc->p_flags & PENDING) --sig_procs;
15079     sigemptyset(&rc->p_pending);
15080     rc->p_pendcount = 0;
15081     rc->p_flags = P_SLOT_FREE;
15082     return(OK);
15083   }


15086   /*===========================================================================*
15087    *                              do_getsp                                     *
15088    *===========================================================================*/
15089   PRIVATE int do_getsp(m_ptr)
15090   register message *m_ptr;      /* pointer to request message */
15091   {
15092   /* Handle sys_getsp().  MM wants to know what sp is. */
15093
15094     register struct proc *rp;
15095
15096     if (!isoksusern(m_ptr->PROC1)) return(E_BAD_PROC);
15097     rp = proc_addr(m_ptr->PROC1);
15098     m_ptr->STACK_PTR = (char *) rp->p_reg.sp;     /* return sp here (bad type) */
15099     return(OK);
15100   }


15103   /*===========================================================================*
15104    *                              do_times                                     *
15105    *===========================================================================*/
15106   PRIVATE int do_times(m_ptr)
15107   register message *m_ptr;      /* pointer to request message */
15108   {
15109   /* Handle sys_times().  Retrieve the accounting information. */
15110
15111     register struct proc *rp;
15112
15113     if (!isoksusern(m_ptr->PROC1)) return E_BAD_PROC;
15114     rp = proc_addr(m_ptr->PROC1);
```

```
15115
15116     /* Insert the times needed by the TIMES system call in the message. */
15117     lock();                       /* halt the volatile time counters in rp */
15118     m_ptr->USER_TIME   = rp->user_time;
15119     m_ptr->SYSTEM_TIME = rp->sys_time;
15120     unlock();
15121     m_ptr->CHILD_UTIME = rp->child_utime;
15122     m_ptr->CHILD_STIME = rp->child_stime;
15123     m_ptr->BOOT_TICKS  = get_uptime();
15124     return(OK);
15125   }


15128   /*===========================================================================*
15129    *                                do_abort                                   *
15130    *===========================================================================*/
15131   PRIVATE int do_abort(m_ptr)
15132   message *m_ptr;                 /* pointer to request message */
15133   {
15134   /* Handle sys_abort.  MINIX is unable to continue.  Terminate operation. */
15135     char monitor_code[64];
15136     phys_bytes src_phys;
15137
15138     if (m_ptr->m1_i1 == RBT_MONITOR) {
15139           /* The monitor is to run user specified instructions. */
15140           src_phys = numap(m_ptr->m_source, (vir_bytes) m_ptr->m1_p1,
15141                                       (vir_bytes) sizeof(monitor_code));
15142           if (src_phys == 0) panic("bad monitor code from", m_ptr->m_source);
15143           phys_copy(src_phys, vir2phys(monitor_code),
15144                                       (phys_bytes) sizeof(monitor_code));
15145           reboot_code = vir2phys(monitor_code);
15146     }
15147     wreboot(m_ptr->m1_i1);
15148     return(OK);                   /* pro-forma (really EDISASTER) */
15149   }




15154   /*===========================================================================*
15155    *                               do_sendsig                                  *
15156    *===========================================================================*/
15157   PRIVATE int do_sendsig(m_ptr)
15158   message *m_ptr;                 /* pointer to request message */
15159   {
15160   /* Handle sys_sendsig, POSIX-style signal */
15161
15162     struct sigmsg smsg;
15163     register struct proc *rp;
15164     phys_bytes src_phys, dst_phys;
15165     struct sigcontext sc, *scp;
15166     struct sigframe fr, *frp;
15167
15168     if (!isokusern(m_ptr->PROC1)) return(E_BAD_PROC);
15169     rp = proc_addr(m_ptr->PROC1);
15170
15171     /* Get the sigmsg structure into our address space.  */
15172     src_phys = umap(proc_addr(MM_PROC_NR), D, (vir_bytes) m_ptr->SIG_CTXT_PTR,
15173                     (vir_bytes) sizeof(struct sigmsg));
15174     if (src_phys == 0)
```



```
15175           panic("do_sendsig can't signal: bad sigmsg address from MM", NO_NUM);
15176     phys_copy(src_phys, vir2phys(&smsg), (phys_bytes) sizeof(struct sigmsg));
15177
15178     /* Compute the usr stack pointer value where sigcontext will be stored. */
15179     scp = (struct sigcontext *) smsg.sm_stkptr - 1;
15180
15181     /* Copy the registers to the sigcontext structure. */
15182     memcpy(&sc.sc_regs, &rp->p_reg, sizeof(struct sigregs));
15183
15184     /* Finish the sigcontext initialization. */
15185     sc.sc_flags = SC_SIGCONTEXT;
15186
15187     sc.sc_mask = smsg.sm_mask;
15188
15189     /* Copy the sigcontext structure to the user's stack. */
15190     dst_phys = umap(rp, D, (vir_bytes) scp,
15191                     (vir_bytes) sizeof(struct sigcontext));
15192     if (dst_phys == 0) return(EFAULT);
15193     phys_copy(vir2phys(&sc), dst_phys, (phys_bytes) sizeof(struct sigcontext));
15194
15195     /* Initialize the sigframe structure. */
15196     frp = (struct sigframe *) scp - 1;
15197     fr.sf_scpcopy = scp;
15198     fr.sf_retadr2= (void (*)()) rp->p_reg.pc;
15199     fr.sf_fp = rp->p_reg.fp;
15200     rp->p_reg.fp = (reg_t) &frp->sf_fp;
15201     fr.sf_scp = scp;
15202     fr.sf_code = 0;       /* XXX - should be used for type of FP exception */
15203     fr.sf_signo = smsg.sm_signo;
15204     fr.sf_retadr = (void (*)()) smsg.sm_sigreturn;
15205
15206     /* Copy the sigframe structure to the user's stack. */
15207     dst_phys = umap(rp, D, (vir_bytes) frp, (vir_bytes) sizeof(struct sigframe));
15208     if (dst_phys == 0) return(EFAULT);
15209     phys_copy(vir2phys(&fr), dst_phys, (phys_bytes) sizeof(struct sigframe));
15210
15211     /* Reset user registers to execute the signal handler. */
15212     rp->p_reg.sp = (reg_t) frp;
15213     rp->p_reg.pc = (reg_t) smsg.sm_sighandler;
15214
15215     return(OK);
15216   }

15218   /*===========================================================================*
15219    *                              do_sigreturn                                 *
15220    *===========================================================================*/
15221   PRIVATE int do_sigreturn(m_ptr)
15222   register message *m_ptr;
15223   {
15224   /* POSIX style signals require sys_sigreturn to put things in order before the
15225    * signalled process can resume execution
15226    */
15227
15228     struct sigcontext sc;
15229     register struct proc *rp;
15230     phys_bytes src_phys;
15231
15232     if (!isokusern(m_ptr->PROC1)) return(E_BAD_PROC);
15233     rp = proc_addr(m_ptr->PROC1);
15234
```

```
15235     /* Copy in the sigcontext structure. */
15236     src_phys = umap(rp, D, (vir_bytes) m_ptr->SIG_CTXT_PTR,
15237                     (vir_bytes) sizeof(struct sigcontext));
15238     if (src_phys == 0) return(EFAULT);
15239     phys_copy(src_phys, vir2phys(&sc), (phys_bytes) sizeof(struct sigcontext));
15240
15241     /* Make sure that this is not just a jmp_buf. */
15242     if ((sc.sc_flags & SC_SIGCONTEXT) == 0) return(EINVAL);
15243
15244     /* Fix up only certain key registers if the compiler doesn't use
15245      * register variables within functions containing setjmp.
15246      */
15247     if (sc.sc_flags & SC_NOREGLOCALS) {
15248           rp->p_reg.retreg = sc.sc_retreg;
15249           rp->p_reg.fp = sc.sc_fp;
15250           rp->p_reg.pc = sc.sc_pc;
15251           rp->p_reg.sp = sc.sc_sp;
15252           return (OK);
15253     }
15254     sc.sc_psw  = rp->p_reg.psw;
15255
15256   #if (CHIP == INTEL)
15257     /* Don't panic kernel if user gave bad selectors. */
15258     sc.sc_cs = rp->p_reg.cs;
15259     sc.sc_ds = rp->p_reg.ds;
15260     sc.sc_es = rp->p_reg.es;
15261   #if _WORD_SIZE == 4
15262     sc.sc_fs = rp->p_reg.fs;
15263     sc.sc_gs = rp->p_reg.gs;
15264   #endif
15265   #endif
15266
15267     /* Restore the registers. */
15268     memcpy(&rp->p_reg, (char *)&sc.sc_regs, sizeof(struct sigregs));
15269
15270     return(OK);
15271   }

15273   /*===========================================================================*
15274    *                                do_kill                                    *
15275    *===========================================================================*/
15276   PRIVATE int do_kill(m_ptr)
15277   register message *m_ptr;        /* pointer to request message */
15278   {
15279   /* Handle sys_kill(). Cause a signal to be sent to a process via MM.
15280    * Note that this has nothing to do with the kill (2) system call, this
15281    * is how the FS (and possibly other servers) get access to cause_sig to
15282    * send a KSIG message to MM
15283    */
15284
15285     if (!isokusern(m_ptr->PR)) return(E_BAD_PROC);
15286     cause_sig(m_ptr->PR, m_ptr->SIGNUM);
15287     return(OK);
15288   }


15291   /*===========================================================================*
15292    *                                do_endsig                                  *
15293    *===========================================================================*/
15294   PRIVATE int do_endsig(m_ptr)
```



```
15295   register message *m_ptr;        /* pointer to request message */
15296   {
15297   /* Finish up after a KSIG-type signal, caused by a SYS_KILL message or a call
15298    * to cause_sig by a task
15299    */
15300
15301     register struct proc *rp;
15302
15303     if (!isokusern(m_ptr->PROC1)) return(E_BAD_PROC);
15304     rp = proc_addr(m_ptr->PROC1);
15305
15306     /* MM has finished one KSIG. */
15307     if (rp->p_pendcount != 0 && --rp->p_pendcount == 0
15308         && (rp->p_flags &= ~SIG_PENDING) == 0)
15309           lock_ready(rp);
15310     return(OK);
15311   }

15313   /*===========================================================================*
15314    *                                do_copy                                    *
15315    *===========================================================================*/
15316   PRIVATE int do_copy(m_ptr)
15317   register message *m_ptr;        /* pointer to request message */
15318   {
15319   /* Handle sys_copy().  Copy data for MM or FS. */
15320
15321     int src_proc, dst_proc, src_space, dst_space;
15322     vir_bytes src_vir, dst_vir;
15323     phys_bytes src_phys, dst_phys, bytes;
15324
15325     /* Dismember the command message. */
15326     src_proc = m_ptr->SRC_PROC_NR;
15327     dst_proc = m_ptr->DST_PROC_NR;
15328     src_space = m_ptr->SRC_SPACE;
15329     dst_space = m_ptr->DST_SPACE;
15330     src_vir = (vir_bytes) m_ptr->SRC_BUFFER;
15331     dst_vir = (vir_bytes) m_ptr->DST_BUFFER;
15332     bytes = (phys_bytes) m_ptr->COPY_BYTES;
15333
15334     /* Compute the source and destination addresses and do the copy. */
15335     if (src_proc == ABS)
15336           src_phys = (phys_bytes) m_ptr->SRC_BUFFER;
15337     else {
15338           if (bytes != (vir_bytes) bytes)
15339                   /* This would happen for 64K segments and 16-bit vir_bytes.
15340                    * It would happen a lot for do_fork except MM uses ABS
15341                    * copies for that case.
15342                    */
15343                   panic("overflow in count in do_copy", NO_NUM);
15344
15345           src_phys = umap(proc_addr(src_proc), src_space, src_vir,
15346                           (vir_bytes) bytes);
15347           }
15348
15349     if (dst_proc == ABS)
15350           dst_phys = (phys_bytes) m_ptr->DST_BUFFER;
15351     else
15352           dst_phys = umap(proc_addr(dst_proc), dst_space, dst_vir,
15353                           (vir_bytes) bytes);
15354
```

```
15355     if (src_phys == 0 || dst_phys == 0) return(EFAULT);
15356     phys_copy(src_phys, dst_phys, bytes);
15357     return(OK);
15358   }


15361   /*===========================================================================*
15362    *                                do_vcopy                                   *
15363    *===========================================================================*/
15364   PRIVATE int do_vcopy(m_ptr)
15365   register message *m_ptr;        /* pointer to request message */
15366   {
15367   /* Handle sys_vcopy(). Copy multiple blocks of memory */
15368
15369     int src_proc, dst_proc, vect_s, i;
15370     vir_bytes src_vir, dst_vir, vect_addr;
15371     phys_bytes src_phys, dst_phys, bytes;
15372     cpvec_t cpvec_table[CPVEC_NR];
15373
15374     /* Dismember the command message. */
15375     src_proc = m_ptr->m1_i1;
15376     dst_proc = m_ptr->m1_i2;
15377     vect_s = m_ptr->m1_i3;
15378     vect_addr = (vir_bytes)m_ptr->m1_p1;
15379
15380     if (vect_s > CPVEC_NR) return EDOM;
15381
15382     src_phys= numap (m_ptr->m_source, vect_addr, vect_s * sizeof(cpvec_t));
15383     if (!src_phys) return EFAULT;
15384     phys_copy(src_phys, vir2phys(cpvec_table),
15385                                 (phys_bytes) (vect_s * sizeof(cpvec_t)));
15386
15387     for (i = 0; i < vect_s; i++) {
15388           src_vir= cpvec_table[i].cpv_src;
15389           dst_vir= cpvec_table[i].cpv_dst;
15390           bytes= cpvec_table[i].cpv_size;
15391           src_phys = numap(src_proc,src_vir,(vir_bytes)bytes);
15392           dst_phys = numap(dst_proc,dst_vir,(vir_bytes)bytes);
15393           if (src_phys == 0 || dst_phys == 0) return(EFAULT);
15394           phys_copy(src_phys, dst_phys, bytes);
15395     }
15396     return(OK);
15397   }


15400   /*===========================================================================*
15401    *                                do_gboot                                   *
15402    *===========================================================================*/
15403   PUBLIC struct bparam_s boot_parameters;
15404
15405   PRIVATE int do_gboot(m_ptr)
15406   message *m_ptr;                 /* pointer to request message */
15407   {
15408   /* Copy the boot parameters.  Normally only called during fs init. */
15409
15410     phys_bytes dst_phys;
15411
15412     dst_phys = umap(proc_addr(m_ptr->PROC1), D, (vir_bytes) m_ptr->MEM_PTR,
15413                                 (vir_bytes) sizeof(boot_parameters));
15414     if (dst_phys == 0) panic("bad call to SYS_GBOOT", NO_NUM);
```



```
15415     phys_copy(vir2phys(&boot_parameters), dst_phys,
15416                                 (phys_bytes) sizeof(boot_parameters));
15417     return(OK);
15418   }


15421   /*===========================================================================*
15422    *                                do_mem                                     *
15423    *===========================================================================*/
15424   PRIVATE int do_mem(m_ptr)
15425   register message *m_ptr;        /* pointer to request message */
15426   {
15427   /* Return the base and size of the next chunk of memory. */
15428
15429     struct memory *memp;
15430
15431     for (memp = mem; memp < &mem[NR_MEMS]; ++memp) {
15432           m_ptr->m1_i1 = memp->base;
15433           m_ptr->m1_i2 = memp->size;
15434           m_ptr->m1_i3 = tot_mem_size;
15435           memp->size = 0;
15436           if (m_ptr->m1_i2 != 0) break;           /* found a chunk */
15437     }
15438     return(OK);
15439   }


15442   /*===========================================================================*
15443    *                                do_umap                                    *
15444    *===========================================================================*/
15445   PRIVATE int do_umap(m_ptr)
15446   register message *m_ptr;        /* pointer to request message */
15447   {
15448   /* Same as umap(), for non-kernel processes. */
15449
15450     m_ptr->SRC_BUFFER = umap(proc_addr((int) m_ptr->SRC_PROC_NR),
15451                             (int) m_ptr->SRC_SPACE,
15452                             (vir_bytes) m_ptr->SRC_BUFFER,
15453                             (vir_bytes) m_ptr->COPY_BYTES);
15454     return(OK);
15455   }


15458   /*===========================================================================*
15459    *                                do_trace                                   *
15460    *===========================================================================*/
15461   #define TR_PROCNR       (m_ptr->m2_i1)
15462   #define TR_REQUEST      (m_ptr->m2_i2)
15463   #define TR_ADDR         ((vir_bytes) m_ptr->m2_l1)
15464   #define TR_DATA         (m_ptr->m2_l2)
15465   #define TR_VLSIZE       ((vir_bytes) sizeof(long))
15466
15467   PRIVATE int do_trace(m_ptr)
15468   register message *m_ptr;
15469   {
15470   /* Handle the debugging commands supported by the ptrace system call
15471    * The commands are:
15472    * T_STOP       stop the process
15473    * T_OK         enable tracing by parent for this process
15474    * T_GETINS     return value from instruction space
```

```
15475  * T_GETDATA    return value from data space
15476  * T_GETUSER    return value from user process table
15477  * T_SETINS     set value from instruction space
15478  * T_SETDATA    set value from data space
15479  * T_SETUSER    set value in user process table
15480  * T_RESUME     resume execution
15481  * T_EXIT       exit
15482  * T_STEP       set trace bit
15483  *
15484  * The T_OK and T_EXIT commands are handled completely by the memory manager,
15485  * all others come here.
15486  */
15487
15488  register struct proc *rp;
15489  phys_bytes src, dst;
15490  int i;
15491
15492  rp = proc_addr(TR_PROCNR);
15493  if (rp->p_flags & P_SLOT_FREE) return(EIO);
15494  switch (TR_REQUEST) {
15495  case T_STOP:                  /* stop process */
15496        if (rp->p_flags == 0) lock_unready(rp);
15497        rp->p_flags |= P_STOP;
15498        rp->p_reg.psw &= ~TRACEBIT;     /* clear trace bit */
15499        return(OK);
15500
15501  case T_GETINS:                /* return value from instruction space */
15502        if (rp->p_map[T].mem_len != 0) {
15503                if ((src = umap(rp, T, TR_ADDR, TR_VLSIZE)) == 0) return(EIO);
15504                phys_copy(src, vir2phys(&TR_DATA), (phys_bytes) sizeof(long));
15505                break;
15506        }
15507        /* Text space is actually data space - fall through. */
15508
15509  case T_GETDATA:               /* return value from data space */
15510        if ((src = umap(rp, D, TR_ADDR, TR_VLSIZE)) == 0) return(EIO);
15511        phys_copy(src, vir2phys(&TR_DATA), (phys_bytes) sizeof(long));
15512        break;
15513
15514  case T_GETUSER:               /* return value from process table */
15515        if ((TR_ADDR & (sizeof(long) - 1)) != 0 ||
15516            TR_ADDR > sizeof(struct proc) - sizeof(long))
15517                return(EIO);
15518        TR_DATA = *(long *) ((char *) rp + (int) TR_ADDR);
15519        break;
15520
15521  case T_SETINS:                /* set value in instruction space */
15522        if (rp->p_map[T].mem_len != 0) {
15523                if ((dst = umap(rp, T, TR_ADDR, TR_VLSIZE)) == 0) return(EIO);
15524                phys_copy(vir2phys(&TR_DATA), dst, (phys_bytes) sizeof(long));
15525                TR_DATA = 0;
15526                break;
15527        }
15528        /* Text space is actually data space - fall through. */
15529
15530  case T_SETDATA:                       /* set value in data space */
15531        if ((dst = umap(rp, D, TR_ADDR, TR_VLSIZE)) == 0) return(EIO);
15532        phys_copy(vir2phys(&TR_DATA), dst, (phys_bytes) sizeof(long));
15533        TR_DATA = 0;
15534        break;
```



```
15535
15536    case T_SETUSER:                     /* set value in process table */
15537          if ((TR_ADDR & (sizeof(reg_t) - 1)) != 0 ||
15538               TR_ADDR > sizeof(struct stackframe_s) - sizeof(reg_t))
15539                  return(EIO);
15540          i = (int) TR_ADDR;
15541  #if (CHIP == INTEL)
15542          /* Altering segment registers might crash the kernel when it
15543           * tries to load them prior to restarting a process, so do
15544           * not allow it.
15545           */
15546          if (i == (int) &((struct proc *) 0)->p_reg.cs ||
15547              i == (int) &((struct proc *) 0)->p_reg.ds ||
15548              i == (int) &((struct proc *) 0)->p_reg.es ||
15549  #if _WORD_SIZE == 4
15550              i == (int) &((struct proc *) 0)->p_reg.gs ||
15551              i == (int) &((struct proc *) 0)->p_reg.fs ||
15552  #endif
15553              i == (int) &((struct proc *) 0)->p_reg.ss)
15554                  return(EIO);
15555  #endif
15556          if (i == (int) &((struct proc *) 0)->p_reg.psw)
15557                  /* only selected bits are changeable */
15558                  SETPSW(rp, TR_DATA);
15559          else
15560                  *(reg_t *) ((char *) &rp->p_reg + i) = (reg_t) TR_DATA;
15561          TR_DATA = 0;
15562          break;
15563
15564    case T_RESUME:                /* resume execution */
15565          rp->p_flags &= ~P_STOP;
15566          if (rp->p_flags == 0) lock_ready(rp);
15567          TR_DATA = 0;
15568          break;
15569
15570    case T_STEP:                  /* set trace bit */
15571          rp->p_reg.psw |= TRACEBIT;
15572          rp->p_flags &= ~P_STOP;
15573          if (rp->p_flags == 0) lock_ready(rp);
15574          TR_DATA = 0;
15575          break;
15576
15577    default:
15578          return(EIO);
15579    }
15580    return(OK);
15581  }

15583  /*===========================================================================*
15584   *                                cause_sig                                  *
15585   *===========================================================================*/
15586  PUBLIC void cause_sig(proc_nr, sig_nr)
15587  int proc_nr;                  /* process to be signalled */
15588  int sig_nr;                   /* signal to be sent, 1 to _NSIG */
15589  {
15590  /* A task wants to send a signal to a process.   Examples of such tasks are:
15591   *   TTY wanting to cause SIGINT upon getting a DEL
15592   *   CLOCK wanting to cause SIGALRM when timer expires
15593   * FS also uses this to send a signal, via the SYS_KILL message.
15594   * Signals are handled by sending a message to MM.  The tasks don't dare do
```

```
15595     * that directly, for fear of what would happen if MM were busy.  Instead they
15596     * call cause_sig, which sets bits in p_pending, and then carefully checks to
15597     * see if MM is free.  If so, a message is sent to it.  If not, when it becomes
15598     * free, a message is sent.  The process being signaled is blocked while MM
15599     * has not seen or finished with all signals for it.  These signals are
15600     * counted in p_pendcount, and the SIG_PENDING flag is kept nonzero while
15601     * there are some.  It is not sufficient to ready the process when MM is
15602     * informed, because MM can block waiting for FS to do a core dump.
15603     */
15604
15605      register struct proc *rp, *mmp;
15606
15607      rp = proc_addr(proc_nr);
15608      if (sigismember(&rp->p_pending, sig_nr))
15609            return;                 /* this signal already pending */
15610      sigaddset(&rp->p_pending, sig_nr);
15611      ++rp->p_pendcount;            /* count new signal pending */
15612      if (rp->p_flags & PENDING)
15613            return;                 /* another signal already pending */
15614      if (rp->p_flags == 0) lock_unready(rp);
15615      rp->p_flags |= PENDING | SIG_PENDING;
15616      ++sig_procs;                  /* count new process pending */
15617
15618      mmp = proc_addr(MM_PROC_NR);
15619      if ( ((mmp->p_flags & RECEIVING) == 0) || mmp->p_getfrom != ANY) return;
15620      inform();
15621  }


15624  /*==========================================================================*
15625   *                              inform                                      *
15626   *==========================================================================*/
15627  PUBLIC void inform()
15628  {
15629  /* When a signal is detected by the kernel (e.g., DEL), or generated by a task
15630   * (e.g. clock task for SIGALRM), cause_sig() is called to set a bit in the
15631   * p_pending field of the process to signal.  Then inform() is called to see
15632   * if MM is idle and can be told about it.  Whenever MM blocks, a check is
15633   * made to see if 'sig_procs' is nonzero; if so, inform() is called.
15634   */
15635
15636    register struct proc *rp;
15637
15638    /* MM is waiting for new input.  Find a process with pending signals. */
15639    for (rp = BEG_SERV_ADDR; rp < END_PROC_ADDR; rp++)
15640          if (rp->p_flags & PENDING) {
15641                m.m_type = KSIG;
15642                m.SIG_PROC = proc_number(rp);
15643                m.SIG_MAP = rp->p_pending;
15644                sig_procs--;
15645                if (lock_mini_send(proc_addr(HARDWARE), MM_PROC_NR, &m) != OK)
15646                       panic("can't inform MM", NO_NUM);
15647                sigemptyset(&rp->p_pending); /* the ball is now in MM's court */
15648                rp->p_flags &= ~PENDING;/* remains inhibited by SIG_PENDING */
15649                lock_pick_proc();       /* avoid delay in scheduling MM */
15650                return;
15651          }
15652  }
```



```
15655  /*==========================================================================*
15656   *                              umap                                        *
15657   *==========================================================================*/
15658  PUBLIC phys_bytes umap(rp, seg, vir_addr, bytes)
15659  register struct proc *rp;       /* pointer to proc table entry for process */
15660  int seg;                        /* T, D, or S segment */
15661  vir_bytes vir_addr;             /* virtual address in bytes within the seg */
15662  vir_bytes bytes;                /* # of bytes to be copied */
15663  {
15664  /* Calculate the physical memory address for a given virtual address. */
15665
15666    vir_clicks vc;                /* the virtual address in clicks */
15667    phys_bytes pa;                /* intermediate variables as phys_bytes */
15668    phys_bytes seg_base;
15669
15670    /* If 'seg' is D it could really be S and vice versa.  T really means T.
15671     * If the virtual address falls in the gap,  it causes a problem. On the
15672     * 8088 it is probably a legal stack reference, since "stackfaults" are
15673     * not detected by the hardware.  On 8088s, the gap is called S and
15674     * accepted, but on other machines it is called D and rejected.
15675     * The Atari ST behaves like the 8088 in this respect.
15676     */
15677
15678    if (bytes <= 0) return( (phys_bytes) 0);
15679    vc = (vir_addr + bytes - 1) >> CLICK_SHIFT;   /* last click of data */
15680
15681    if (seg != T)
15682          seg = (vc < rp->p_map[D].mem_vir + rp->p_map[D].mem_len ? D : S);
15683
15684    if((vir_addr>>CLICK_SHIFT) >= rp->p_map[seg].mem_vir+ rp->p_map[seg].mem_len)
15685          return( (phys_bytes) 0 );
15686    seg_base = (phys_bytes) rp->p_map[seg].mem_phys;
15687    seg_base = seg_base << CLICK_SHIFT;   /* segment origin in bytes */
15688    pa = (phys_bytes) vir_addr;
15689    pa -= rp->p_map[seg].mem_vir << CLICK_SHIFT;
15690    return(seg_base + pa);
15691  }


15694  /*==========================================================================*
15695   *                              numap                                       *
15696   *==========================================================================*/
15697  PUBLIC phys_bytes numap(proc_nr, vir_addr, bytes)
15698  int proc_nr;                    /* process number to be mapped */
15699  vir_bytes vir_addr;             /* virtual address in bytes within D seg */
15700  vir_bytes bytes;                /* # of bytes required in segment  */
15701  {
15702  /* Do umap() starting from a process number instead of a pointer.  This
15703   * function is used by device drivers, so they need not know about the
15704   * process table.  To save time, there is no 'seg' parameter. The segment
15705   * is always D.
15706   */
15707
15708    return(umap(proc_addr(proc_nr), D, vir_addr, bytes));
15709  }

15711  #if (CHIP == INTEL)
15712  /*==========================================================================*
15713   *                              alloc_segments                              *
15714   *==========================================================================*/
```

```
15715   PUBLIC void alloc_segments(rp)
15716   register struct proc *rp;
15717   {
15718   /* This is called only by do_newmap, but is broken out as a separate function
15719    * because so much is hardware-dependent.
15720    */
15721
15722     phys_bytes code_bytes;
15723     phys_bytes data_bytes;
15724     int privilege;
15725
15726     if (protected_mode) {
15727           data_bytes = (phys_bytes) (rp->p_map[S].mem_vir + rp->p_map[S].mem_len)
15728                       << CLICK_SHIFT;
15729           if (rp->p_map[T].mem_len == 0)
15730                   code_bytes = data_bytes;        /* common I&D, poor protect */
15731           else
15732                   code_bytes = (phys_bytes) rp->p_map[T].mem_len << CLICK_SHIFT;
15733           privilege = istaskp(rp) ? TASK_PRIVILEGE : USER_PRIVILEGE;
15734           init_codeseg(&rp->p_ldt[CS_LDT_INDEX],
15735                       (phys_bytes) rp->p_map[T].mem_phys << CLICK_SHIFT,
15736                       code_bytes, privilege);
15737           init_dataseg(&rp->p_ldt[DS_LDT_INDEX],
15738                       (phys_bytes) rp->p_map[D].mem_phys << CLICK_SHIFT,
15739                       data_bytes, privilege);
15740           rp->p_reg.cs = (CS_LDT_INDEX * DESC_SIZE) | TI | privilege;
15741   #if _WORD_SIZE == 4
15742           rp->p_reg.gs =
15743           rp->p_reg.fs =
15744   #endif
15745           rp->p_reg.ss =
15746           rp->p_reg.es =
15747           rp->p_reg.ds = (DS_LDT_INDEX*DESC_SIZE) | TI | privilege;
15748     } else {
15749           rp->p_reg.cs = click_to_hclick(rp->p_map[T].mem_phys);
15750           rp->p_reg.ss =
15751           rp->p_reg.es =
15752           rp->p_reg.ds = click_to_hclick(rp->p_map[D].mem_phys);
15753     }
15754   }
15755   #endif /* (CHIP == INTEL) */
```

++
src/mm/mm.h
++

```
15800   /* This is the master header for mm.  It includes some other files
15801    * and defines the principal constants.
15802    */
15803   #define _POSIX_SOURCE      1    /* tell headers to include POSIX stuff */
15804   #define _MINIX             1    /* tell headers to include MINIX stuff */
15805   #define _SYSTEM            1    /* tell headers that this is the kernel */
15806
15807   /* The following are so basic, all the *.c files get them automatically. */
15808   #include <minix/config.h>       /* MUST be first */
15809   #include <ansi.h>               /* MUST be second */
```



```
15810   #include <sys/types.h>
15811   #include <minix/const.h>
15812   #include <minix/type.h>
15813
15814   #include <fcntl.h>
15815   #include <unistd.h>
15816   #include <minix/syslib.h>
15817
15818   #include <limits.h>
15819   #include <errno.h>
15820
15821   #include "const.h"
15822   #include "type.h"
15823   #include "proto.h"
15824   #include "glo.h"
```

++
src/mm/const.h
++

```
15900   /* Constants used by the Memory Manager. */
15901
15902   #define NO_MEM ((phys_clicks) 0)  /* returned by alloc_mem() with mem is up */
15903
15904   #if (CHIP == INTEL && _WORD_SIZE == 2)
15905   /* These definitions are used in size_ok and are not needed for 386.
15906    * The 386 segment granularity is 1 for segments smaller than 1M and 4096
15907    * above that.
15908    */
15909   #define PAGE_SIZE         16    /* how many bytes in a page (s.b.HCLICK_SIZE)*/
15910   #define MAX_PAGES       4096    /* how many pages in the virtual addr space */
15911   #endif
15912
15913   #define printf        printk
15914
15915   #define INIT_PID           1    /* init's process id number */
```

++
src/mm/type.h
++

```
16000   /* If there were any type definitions local to the Memory Manager, they would
16001    * be here.  This file is included only for symmetry with the kernel and File
16002    * System, which do have some local type definitions.
16003    */
16004
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/mm/proto.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

16100   /* Function prototypes. */
16101
16102   struct mproc;           /* need types outside of parameter list --kub */
16103   struct stat;
16104
16105   /* alloc.c */
16106   _PROTOTYPE( phys_clicks alloc_mem, (phys_clicks clicks)                );
16107   _PROTOTYPE( void free_mem, (phys_clicks base, phys_clicks clicks)       );
16108   _PROTOTYPE( phys_clicks max_hole, (void)                               );
16109   _PROTOTYPE( void mem_init, (phys_clicks *total, phys_clicks *free)     );
16110   _PROTOTYPE( phys_clicks mem_left, (void)                               );
16111   _PROTOTYPE( int do_brk3, (void)                                        );
16112
16113   /* break.c */
16114   _PROTOTYPE( int adjust, (struct mproc *rmp,
16115                           vir_clicks data_clicks, vir_bytes sp)          );
16116   _PROTOTYPE( int do_brk, (void)                                         );
16117   _PROTOTYPE( int size_ok, (int file_type, vir_clicks tc, vir_clicks dc,
16118                           vir_clicks sc, vir_clicks dvir, vir_clicks s_vir) );
16119
16120   /* exec.c */
16121   _PROTOTYPE( int do_exec, (void)                                        );
16122   _PROTOTYPE( struct mproc *find_share, (struct mproc *mp_ign, Ino_t ino,
16123                           Dev_t dev, time_t ctime)                       );
16124
16125   /* forkexit.c */
16126   _PROTOTYPE( int do_fork, (void)                                        );
16127   _PROTOTYPE( int do_mm_exit, (void)                                     );
16128   _PROTOTYPE( int do_waitpid, (void)                                     );
16129   _PROTOTYPE( void mm_exit, (struct mproc *rmp, int exit_status)         );
16130
16131   /* getset.c */
16132   _PROTOTYPE( int do_getset, (void)                                      );
16133
16134   /* main.c */
16135   _PROTOTYPE( void main, (void)                                          );
16136
16137   #if (MACHINE == MACINTOSH)
16138   _PROTOTYPE( phys_clicks start_click, (void)                            );
16139   #endif
16140
16141   _PROTOTYPE( void reply, (int proc_nr, int result, int res2, char *respt));
16142
16143   /* putk.c */
16144   _PROTOTYPE( void putk, (int c)                                         );
16145
16146   /* signal.c */
16147   _PROTOTYPE( int do_alarm, (void)                                       );
16148   _PROTOTYPE( int do_kill, (void)                                        );
16149   _PROTOTYPE( int do_ksig, (void)                                        );
16150   _PROTOTYPE( int do_pause, (void)                                       );
16151   _PROTOTYPE( int set_alarm, (int proc_nr, int sec)                      );
16152   _PROTOTYPE( int check_sig, (pid_t proc_id, int signo)                  );
16153   _PROTOTYPE( void sig_proc, (struct mproc *rmp, int sig_nr)             );
16154   _PROTOTYPE( int do_sigaction, (void)                                   );
```



```
16155   _PROTOTYPE( int do_sigpending, (void)                                  );
16156   _PROTOTYPE( int do_sigprocmask, (void)                                 );
16157   _PROTOTYPE( int do_sigreturn, (void)                                   );
16158   _PROTOTYPE( int do_sigsuspend, (void)                                  );
16159   _PROTOTYPE( int do_reboot, (void)                                      );
16160
16161   /* trace.c */
16162   _PROTOTYPE( int do_trace, (void)                                       );
16163   _PROTOTYPE( void stop_proc, (struct mproc *rmp, int sig_nr)            );
16164
16165   /* utility.c */
16166   _PROTOTYPE( int allowed, (char *name_buf, struct stat *s_buf, int mask) );
16167   _PROTOTYPE( int no_sys, (void)                                         );
16168   _PROTOTYPE( void panic, (char *format, int num)                        );
16169   _PROTOTYPE( void tell_fs, (int what, int p1, int p2, int p3)           );
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/mm/glo.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

16200   /* EXTERN should be extern except in table.c */
16201   #ifdef _TABLE
16202   #undef EXTERN
16203   #define EXTERN
16204   #endif
16205
16206   /* Global variables. */
16207   EXTERN struct mproc *mp;        /* ptr to 'mproc' slot of current process */
16208   EXTERN int dont_reply;          /* normally 0; set to 1 to inhibit reply */
16209   EXTERN int procs_in_use;        /* how many processes are marked as IN_USE */
16210
16211   /* The parameters of the call are kept here. */
16212   EXTERN message mm_in;           /* the incoming message itself is kept here. */
16213   EXTERN message mm_out;          /* the reply message is built up here. */
16214   EXTERN int who;                 /* caller's proc number */
16215   EXTERN int mm_call;             /* system call number */
16216
16217   /* The following variables are used for returning results to the caller. */
16218   EXTERN int err_code;            /* temporary storage for error number */
16219   EXTERN int result2;             /* secondary result */
16220   EXTERN char *res_ptr;           /* result, if pointer */
16221
16222   extern _PROTOTYPE (int (*call_vec[]), (void) ); /* system call handlers */
16223   extern char core_name[];        /* file name where core images are produced */
16224   EXTERN sigset_t core_sset;      /* which signals cause core images */
```

++
src/mm/mproc.h
++

```
16300   /* This table has one slot per process.  It contains all the memory management
16301    * information for each process.  Among other things, it defines the text, data
16302    * and stack segments, uids and gids, and various flags.  The kernel and file
16303    * systems have tables that are also indexed by process, with the contents
16304    * of corresponding slots referring to the same process in all three.
16305    */
16306
16307   EXTERN struct mproc {
16308     struct mem_map mp_seg[NR_SEGS];/* points to text, data, stack */
16309     char mp_exitstatus;           /* storage for status when process exits */
16310     char mp_sigstatus;            /* storage for signal # for killed procs */
16311     pid_t mp_pid;                 /* process id */
16312     pid_t mp_procgrp;             /* pid of process group (used for signals) */
16313     pid_t mp_wpid;                /* pid this process is waiting for */
16314     int mp_parent;                /* index of parent process */
16315
16316     /* Real and effective uids and gids. */
16317     uid_t mp_realuid;             /* process' real uid */
16318     uid_t mp_effuid;              /* process' effective uid */
16319     gid_t mp_realgid;             /* process' real gid */
16320     gid_t mp_effgid;              /* process' effective gid */
16321
16322     /* File identification for sharing. */
16323     ino_t mp_ino;                 /* inode number of file */
16324     dev_t mp_dev;                 /* device number of file system */
16325     time_t mp_ctime;              /* inode changed time */
16326
16327     /* Signal handling information. */
16328     sigset_t mp_ignore;           /* 1 means ignore the signal, 0 means don't */
16329     sigset_t mp_catch;            /* 1 means catch the signal, 0 means don't */
16330     sigset_t mp_sigmask;          /* signals to be blocked */
16331     sigset_t mp_sigmask2;         /* saved copy of mp_sigmask */
16332     sigset_t mp_sigpending;       /* signals being blocked */
16333     struct sigaction mp_sigact[_NSIG + 1]; /* as in sigaction(2) */
16334     vir_bytes mp_sigreturn;       /* address of C library __sigreturn function */
16335
16336     /* Backwards compatibility for signals. */
16337     sighandler_t mp_func;         /* all sigs vectored to a single user fcn */
16338
16339     unsigned mp_flags;            /* flag bits */
16340     vir_bytes mp_procargs;        /* ptr to proc's initial stack arguments */
16341   } mproc[NR_PROCS];
16342
16343   /* Flag values */
16344   #define IN_USE          001   /* set when 'mproc' slot in use */
16345   #define WAITING         002   /* set by WAIT system call */
16346   #define HANGING         004   /* set by EXIT system call */
16347   #define PAUSED          010   /* set by PAUSE system call */
16348   #define ALARM_ON        020   /* set when SIGALRM timer started */
16349   #define SEPARATE        040   /* set if file is separate I & D space */
16350   #define TRACED         0100   /* set if process is to be traced */
16351   #define STOPPED        0200   /* set if process stopped for tracing */
16352   #define SIGSUSPENDED   0400   /* set by SIGSUSPEND system call */
16353
16354   #define NIL_MPROC ((struct mproc *) 0)
```


++
src/mm/param.h
++

```
16400   /* The following names are synonyms for the variables in the input message. */
16401   #define addr            mm_in.m1_p1
16402   #define exec_name       mm_in.m1_p1
16403   #define exec_len        mm_in.m1_i1
16404   #define func            mm_in.m6_f1
16405   #define grpid           (gid_t) mm_in.m1_i1
16406   #define namelen         mm_in.m1_i1
16407   #define pid             mm_in.m1_i1
16408   #define seconds         mm_in.m1_i1
16409   #define sig             mm_in.m6_i1
16410   #define stack_bytes     mm_in.m1_i2
16411   #define stack_ptr       mm_in.m1_p2
16412   #define status          mm_in.m1_i1
16413   #define usr_id          (uid_t) mm_in.m1_i1
16414   #define request         mm_in.m2_i2
16415   #define taddr           mm_in.m2_l1
16416   #define data            mm_in.m2_l2
16417   #define sig_nr          mm_in.m1_i2
16418   #define sig_nsa         mm_in.m1_p1
16419   #define sig_osa         mm_in.m1_p2
16420   #define sig_ret         mm_in.m1_p3
16421   #define sig_set         mm_in.m2_l1
16422   #define sig_how         mm_in.m2_i1
16423   #define sig_flags       mm_in.m2_i2
16424   #define sig_context     mm_in.m2_p1
16425   #ifdef _SIGMESSAGE
16426   #define sig_msg         mm_in.m1_i1
16427   #endif
16428   #define reboot_flag     mm_in.m1_i1
16429   #define reboot_code     mm_in.m1_p1
16430   #define reboot_size     mm_in.m1_i2
16431
16432   /* The following names are synonyms for the variables in the output message. */
16433   #define reply_type      mm_out.m_type
16434   #define reply_i1        mm_out.m2_i1
16435   #define reply_p1        mm_out.m2_p1
16436   #define ret_mask        mm_out.m2_l1
16437
```

++
src/mm/table.c
++

```
16500   /* This file contains the table used to map system call numbers onto the
16501    * routines that perform them.
16502    */
16503
16504   #define _TABLE
16505
16506   #include "mm.h"
16507   #include <minix/callnr.h>
16508   #include <signal.h>
16509   #include "mproc.h"
```

```
16510   #include "param.h"
16511
16512   /* Miscellaneous */
16513   char core_name[] = "core";      /* file name where core images are produced */
16514
16515   _PROTOTYPE (int (*call_vec[NCALLS]), (void) ) = {
16516           no_sys,         /*  0 = unused  */
16517           do_mm_exit,     /*  1 = exit    */
16518           do_fork,        /*  2 = fork    */
16519           no_sys,         /*  3 = read    */
16520           no_sys,         /*  4 = write   */
16521           no_sys,         /*  5 = open    */
16522           no_sys,         /*  6 = close   */
16523           do_waitpid,     /*  7 = wait    */
16524           no_sys,         /*  8 = creat   */
16525           no_sys,         /*  9 = link    */
16526           no_sys,         /* 10 = unlink  */
16527           do_waitpid,     /* 11 = waitpid */
16528           no_sys,         /* 12 = chdir   */
16529           no_sys,         /* 13 = time    */
16530           no_sys,         /* 14 = mknod   */
16531           no_sys,         /* 15 = chmod   */
16532           no_sys,         /* 16 = chown   */
16533           do_brk,         /* 17 = break   */
16534           no_sys,         /* 18 = stat    */
16535           no_sys,         /* 19 = lseek   */
16536           do_getset,      /* 20 = getpid  */
16537           no_sys,         /* 21 = mount   */
16538           no_sys,         /* 22 = umount  */
16539           do_getset,      /* 23 = setuid  */
16540           do_getset,      /* 24 = getuid  */
16541           no_sys,         /* 25 = stime   */
16542           do_trace,       /* 26 = ptrace  */
16543           do_alarm,       /* 27 = alarm   */
16544           no_sys,         /* 28 = fstat   */
16545           do_pause,       /* 29 = pause   */
16546           no_sys,         /* 30 = utime   */
16547           no_sys,         /* 31 = (stty)  */
16548           no_sys,         /* 32 = (gtty)  */
16549           no_sys,         /* 33 = access  */
16550           no_sys,         /* 34 = (nice)  */
16551           no_sys,         /* 35 = (ftime) */
16552           no_sys,         /* 36 = sync    */
16553           do_kill,        /* 37 = kill    */
16554           no_sys,         /* 38 = rename  */
16555           no_sys,         /* 39 = mkdir   */
16556           no_sys,         /* 40 = rmdir   */
16557           no_sys,         /* 41 = dup     */
16558           no_sys,         /* 42 = pipe    */
16559           no_sys,         /* 43 = times   */
16560           no_sys,         /* 44 = (prof)  */
16561           no_sys,         /* 45 = unused  */
16562           do_getset,      /* 46 = setgid  */
16563           do_getset,      /* 47 = getgid  */
16564           no_sys,         /* 48 = (signal)*/
16565           no_sys,         /* 49 = unused  */
16566           no_sys,         /* 50 = unused  */
16567           no_sys,         /* 51 = (acct)  */
16568           no_sys,         /* 52 = (phys)  */
16569           no_sys,         /* 53 = (lock)  */
```



```
16570           no_sys,         /* 54 = ioctl   */
16571           no_sys,         /* 55 = fcntl   */
16572           no_sys,         /* 56 = (mpx)   */
16573           no_sys,         /* 57 = unused  */
16574           no_sys,         /* 58 = unused  */
16575           do_exec,        /* 59 = execve  */
16576           no_sys,         /* 60 = umask   */
16577           no_sys,         /* 61 = chroot  */
16578           do_getset,      /* 62 = setsid  */
16579           do_getset,      /* 63 = getpgrp */
16580
16581           do_ksig,        /* 64 = KSIG: signals originating in the kernel */
16582           no_sys,         /* 65 = UNPAUSE */
16583           no_sys,         /* 66 = unused  */
16584           no_sys,         /* 67 = REVIVE  */
16585           no_sys,         /* 68 = TASK_REPLY  */
16586           no_sys,         /* 69 = unused  */
16587           no_sys,         /* 70 = unused  */
16588           do_sigaction,   /* 71 = sigaction   */
16589           do_sigsuspend,  /* 72 = sigsuspend  */
16590           do_sigpending,  /* 73 = sigpending  */
16591           do_sigprocmask, /* 74 = sigprocmask */
16592           do_sigreturn,   /* 75 = sigreturn   */
16593           do_reboot,      /* 76 = reboot  */
16594   };
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/mm/main.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

16600   /* This file contains the main program of the memory manager and some related
16601    * procedures.  When MINIX starts up, the kernel runs for a little while,
16602    * initializing itself and its tasks, and then it runs MM and FS.  Both MM
16603    * and FS initialize themselves as far as they can.  FS then makes a call to
16604    * MM, because MM has to wait for FS to acquire a RAM disk.  MM asks the
16605    * kernel for all free memory and starts serving requests.
16606    *
16607    * The entry points into this file are:
16608    *   main:      starts MM running
16609    *   reply:     reply to a process making an MM system call
16610    */
16611
16612   #include "mm.h"
16613   #include <minix/callnr.h>
16614   #include <minix/com.h>
16615   #include <signal.h>
16616   #include <fcntl.h>
16617   #include <sys/ioctl.h>
16618   #include "mproc.h"
16619   #include "param.h"
16620
16621   FORWARD _PROTOTYPE( void get_work, (void)                               );
16622   FORWARD _PROTOTYPE( void mm_init, (void)                                );
16623
16624   /*==========================================================================*
```

```
16625  *                              main                                     *
16626  *===========================================================================*/
16627 PUBLIC void main()
16628 {
16629 /* Main routine of the memory manager. */
16630
16631   int error;
16632
16633   mm_init();                    /* initialize memory manager tables */
16634
16635   /* This is MM's main loop-  get work and do it, forever and forever. */
16636   while (TRUE) {
16637         /* Wait for message. */
16638         get_work();             /* wait for an MM system call */
16639         mp = &mproc[who];
16640
16641         /* Set some flags. */
16642         error = OK;
16643         dont_reply = FALSE;
16644         err_code = -999;
16645
16646         /* If the call number is valid, perform the call. */
16647         if (mm_call < 0 || mm_call >= NCALLS)
16648                 error = EBADCALL;
16649         else
16650                 error = (*call_vec[mm_call])();
16651
16652         /* Send the results back to the user to indicate completion. */
16653         if (dont_reply) continue;       /* no reply for EXIT and WAIT */
16654         if (mm_call == EXEC && error == OK) continue;
16655         reply(who, error, result2, res_ptr);
16656   }
16657 }


16660 /*===========================================================================*
16661  *                              get_work                                     *
16662  *===========================================================================*/
16663 PRIVATE void get_work()
16664 {
16665 /* Wait for the next message and extract useful information from it. */
16666
16667   if (receive(ANY, &mm_in) != OK) panic("MM receive error", NO_NUM);
16668   who = mm_in.m_source;         /* who sent the message */
16669   mm_call = mm_in.m_type;       /* system call number */
16670 }


16673 /*===========================================================================*
16674  *                              reply                                        *
16675  *===========================================================================*/
16676 PUBLIC void reply(proc_nr, result, res2, respt)
16677 int proc_nr;                    /* process to reply to */
16678 int result;                     /* result of the call (usually OK or error #)*/
16679 int res2;                       /* secondary result */
16680 char *respt;                    /* result if pointer */
16681 {
16682 /* Send a reply to a user process. */
16683
16684   register struct mproc *proc_ptr;
```



```
16685
16686   proc_ptr = &mproc[proc_nr];
16687   /*
16688    * To make MM robust, check to see if destination is still alive.  This
16689    * validy check must be skipped if the caller is a task.
16690    */
16691   if ((who >=0) && ((proc_ptr->mp_flags&IN_USE) == 0 ||
16692         (proc_ptr->mp_flags&HANGING))) return;
16693
16694   reply_type = result;
16695   reply_i1 = res2;
16696   reply_p1 = respt;
16697   if (send(proc_nr, &mm_out) != OK) panic("MM can't reply", NO_NUM);
16698 }


16701 /*===========================================================================*
16702  *                              mm_init                                      *
16703  *===========================================================================*/
16704 PRIVATE void mm_init()
16705 {
16706 /* Initialize the memory manager. */
16707
16708   static char core_sigs[] = {
16709         SIGQUIT, SIGILL, SIGTRAP, SIGABRT,
16710         SIGEMT, SIGFPE, SIGUSR1, SIGSEGV,
16711         SIGUSR2, 0 };
16712   register int proc_nr;
16713   register struct mproc *rmp;
16714   register char *sig_ptr;
16715   phys_clicks ram_clicks, total_clicks, minix_clicks, free_clicks, dummy;
16716   message mess;
16717   struct mem_map kernel_map[NR_SEGS];
16718   int mem;
16719
16720   /* Build the set of signals which cause core dumps. Do it the Posix
16721    * way, so no knowledge of bit positions is needed.
16722    */
16723   sigemptyset(&core_sset);
16724   for (sig_ptr = core_sigs; *sig_ptr != 0; sig_ptr++)
16725         sigaddset(&core_sset, *sig_ptr);
16726
16727   /* Get the memory map of the kernel to see how much memory it uses,
16728    * including the gap between address 0 and the start of the kernel.
16729    */
16730   sys_getmap(SYSTASK, kernel_map);
16731   minix_clicks = kernel_map[S].mem_phys + kernel_map[S].mem_len;
16732
16733   /* Initialize MM's tables. */
16734   for (proc_nr = 0; proc_nr <= INIT_PROC_NR; proc_nr++) {
16735         rmp = &mproc[proc_nr];
16736         rmp->mp_flags |= IN_USE;
16737         sys_getmap(proc_nr, rmp->mp_seg);
16738         if (rmp->mp_seg[T].mem_len != 0) rmp->mp_flags |= SEPARATE;
16739         minix_clicks += (rmp->mp_seg[S].mem_phys + rmp->mp_seg[S].mem_len)
16740                                 - rmp->mp_seg[T].mem_phys;
16741   }
16742   mproc[INIT_PROC_NR].mp_pid = INIT_PID;
16743   sigemptyset(&mproc[INIT_PROC_NR].mp_ignore);
16744   sigemptyset(&mproc[INIT_PROC_NR].mp_catch);
```

```
16745      procs_in_use = LOW_USER + 1;
16746
16747      /* Wait for FS to send a message telling the RAM disk size then go "on-line".
16748       */
16749      if (receive(FS_PROC_NR, &mess) != OK)
16750            panic("MM can't obtain RAM disk size from FS", NO_NUM);
16751
16752      ram_clicks = mess.m1_i1;
16753
16754      /* Initialize tables to all physical mem. */
16755      mem_init(&total_clicks, &free_clicks);
16756
16757      /* Print memory information. */
16758      printf("\nMemory size =%5dK   ", click_to_round_k(total_clicks));
16759      printf("MINIX =%4dK   ", click_to_round_k(minix_clicks));
16760      printf("RAM disk =%5dK   ", click_to_round_k(ram_clicks));
16761      printf("Available =%5dK\n\n", click_to_round_k(free_clicks));
16762
16763      /* Tell FS to continue. */
16764      if (send(FS_PROC_NR, &mess) != OK)
16765            panic("MM can't sync up with FS", NO_NUM);
16766
16767      /* Tell the memory task where my process table is for the sake of ps(1). */
16768      if ((mem = open("/dev/mem", O_RDWR)) != -1) {
16769            ioctl(mem, MIOCSPSINFO, (void *) mproc);
16770            close(mem);
16771      }
16772    }
```

++

src/mm/forkexit.c

++

```
16800    /* This file deals with creating processes (via FORK) and deleting them (via
16801     * EXIT/WAIT).  When a process forks, a new slot in the 'mproc' table is
16802     * allocated for it, and a copy of the parent's core image is made for the
16803     * child.  Then the kernel and file system are informed.  A process is removed
16804     * from the 'mproc' table when two events have occurred: (1) it has exited or
16805     * been killed by a signal, and (2) the parent has done a WAIT.  If the process
16806     * exits first, it continues to occupy a slot until the parent does a WAIT.
16807     *
16808     * The entry points into this file are:
16809     *   do_fork:    perform the FORK system call
16810     *   do_mm_exit: perform the EXIT system call (by calling mm_exit())
16811     *   mm_exit:    actually do the exiting
16812     *   do_wait:    perform the WAITPID or WAIT system call
16813     */
16814
16815
16816    #include "mm.h"
16817    #include <sys/wait.h>
16818    #include <minix/callnr.h>
16819    #include <signal.h>
16820    #include "mproc.h"
16821    #include "param.h"
16822
16823    #define LAST_FEW            2   /* last few slots reserved for superuser */
16824
```



```
16825    PRIVATE pid_t next_pid = INIT_PID+1;    /* next pid to be assigned */
16826
16827    FORWARD _PROTOTYPE (void cleanup, (register struct mproc *child) );
16828
16829    /*===========================================================================*
16830     *                                do_fork                                    *
16831     *===========================================================================*/
16832    PUBLIC int do_fork()
16833    {
16834    /* The process pointed to by 'mp' has forked.  Create a child process. */
16835
16836      register struct mproc *rmp;   /* pointer to parent */
16837      register struct mproc *rmc;   /* pointer to child */
16838      int i, child_nr, t;
16839      phys_clicks prog_clicks, child_base = 0;
16840      phys_bytes prog_bytes, parent_abs, child_abs; /* Intel only */
16841
16842     /* If tables might fill up during FORK, don't even start since recovery half
16843      * way through is such a nuisance.
16844      */
16845      rmp = mp;
16846      if (procs_in_use == NR_PROCS) return(EAGAIN);
16847      if (procs_in_use >= NR_PROCS-LAST_FEW && rmp->mp_effuid != 0)return(EAGAIN);
16848
16849      /* Determine how much memory to allocate.  Only the data and stack need to
16850       * be copied, because the text segment is either shared or of zero length.
16851       */
16852      prog_clicks = (phys_clicks) rmp->mp_seg[S].mem_len;
16853      prog_clicks += (rmp->mp_seg[S].mem_vir - rmp->mp_seg[D].mem_vir);
16854      prog_bytes = (phys_bytes) prog_clicks << CLICK_SHIFT;
16855      if ( (child_base = alloc_mem(prog_clicks)) == NO_MEM) return(EAGAIN);
16856
16857      /* Create a copy of the parent's core image for the child. */
16858      child_abs = (phys_bytes) child_base << CLICK_SHIFT;
16859      parent_abs = (phys_bytes) rmp->mp_seg[D].mem_phys << CLICK_SHIFT;
16860      i = sys_copy(ABS, 0, parent_abs, ABS, 0, child_abs, prog_bytes);
16861      if (i < 0) panic("do_fork can't copy", i);
16862
16863      /* Find a slot in 'mproc' for the child process.  A slot must exist. */
16864      for (rmc = &mproc[0]; rmc < &mproc[NR_PROCS]; rmc++)
16865            if ( (rmc->mp_flags & IN_USE) == 0) break;
16866
16867      /* Set up the child and its memory map; copy its 'mproc' slot from parent. */
16868      child_nr = (int)(rmc - mproc);        /* slot number of the child */
16869      procs_in_use++;
16870      *rmc = *rmp;                  /* copy parent's process slot to child's */
16871
16872      rmc->mp_parent = who;         /* record child's parent */
16873      rmc->mp_flags &= ~TRACED;     /* child does not inherit trace status */
16874      /* A separate I&D child keeps the parents text segment.  The data and stack
16875       * segments must refer to the new copy.
16876       */
16877      if (!(rmc->mp_flags & SEPARATE)) rmc->mp_seg[T].mem_phys = child_base;
16878      rmc->mp_seg[D].mem_phys = child_base;
16879      rmc->mp_seg[S].mem_phys = rmc->mp_seg[D].mem_phys +
16880                              (rmp->mp_seg[S].mem_vir - rmp->mp_seg[D].mem_vir);
16881      rmc->mp_exitstatus = 0;
16882      rmc->mp_sigstatus = 0;
16883
16884      /* Find a free pid for the child and put it in the table. */
```

```
16885     do {
16886           t = 0;                  /* 't' = 0 means pid still free */
16887           next_pid = (next_pid < 30000 ? next_pid + 1 : INIT_PID + 1);
16888           for (rmp = &mproc[0]; rmp < &mproc[NR_PROCS]; rmp++)
16889                   if (rmp->mp_pid == next_pid || rmp->mp_procgrp == next_pid) {
16890                           t = 1;
16891                           break;
16892                   }
16893           rmc->mp_pid = next_pid; /* assign pid to child */
16894     } while (t);
16895
16896     /* Tell kernel and file system about the (now successful) FORK. */
16897     sys_fork(who, child_nr, rmc->mp_pid, child_base); /* child_base is 68K only*/
16898     tell_fs(FORK, who, child_nr, rmc->mp_pid);
16899
16900     /* Report child's memory map to kernel. */
16901     sys_newmap(child_nr, rmc->mp_seg);
16902
16903     /* Reply to child to wake it up. */
16904     reply(child_nr, 0, 0, NIL_PTR);
16905     return(next_pid);              /* child's pid */
16906   }


16909   /*===========================================================================*
16910    *                                do_mm_exit                                 *
16911    *===========================================================================*/
16912   PUBLIC int do_mm_exit()
16913   {
16914   /* Perform the exit(status) system call. The real work is done by mm_exit(),
16915    * which is also called when a process is killed by a signal.
16916    */
16917
16918     mm_exit(mp, status);
16919     dont_reply = TRUE;            /* don't reply to newly terminated process */
16920     return(OK);                   /* pro forma return code */
16921   }


16924   /*===========================================================================*
16925    *                                mm_exit                                    *
16926    *===========================================================================*/
16927   PUBLIC void mm_exit(rmp, exit_status)
16928   register struct mproc *rmp;     /* pointer to the process to be terminated */
16929   int exit_status;                /* the process' exit status (for parent) */
16930   {
16931   /* A process is done.  Release most of the process' possessions.  If its
16932    * parent is waiting, release the rest, else hang.
16933    */
16934
16935     register int proc_nr;
16936     int parent_waiting, right_child;
16937     pid_t pidarg, procgrp;
16938     phys_clicks base, size, s;            /* base and size used on 68000 only */
16939
16940     proc_nr = (int) (rmp - mproc);        /* get process slot number */
16941
16942     /* Remember a session leader's process group. */
16943     procgrp = (rmp->mp_pid == mp->mp_procgrp) ? mp->mp_procgrp : 0;
16944
```



```
16945     /* If the exited process has a timer pending, kill it. */
16946     if (rmp->mp_flags & ALARM_ON) set_alarm(proc_nr, (unsigned) 0);
16947
16948     /* Tell the kernel and FS that the process is no longer runnable. */
16949     tell_fs(EXIT, proc_nr, 0, 0);  /* file system can free the proc slot */
16950     sys_xit(rmp->mp_parent, proc_nr, &base, &size);
16951
16952     /* Release the memory occupied by the child. */
16953     if (find_share(rmp, rmp->mp_ino, rmp->mp_dev, rmp->mp_ctime) == NULL) {
16954           /* No other process shares the text segment, so free it. */
16955           free_mem(rmp->mp_seg[T].mem_phys, rmp->mp_seg[T].mem_len);
16956     }
16957     /* Free the data and stack segments. */
16958     free_mem(rmp->mp_seg[D].mem_phys,
16959         rmp->mp_seg[S].mem_vir + rmp->mp_seg[S].mem_len - rmp->mp_seg[D].mem_vir);
16960
16961     /* The process slot can only be freed if the parent has done a WAIT. */
16962     rmp->mp_exitstatus = (char) exit_status;
16963     pidarg = mproc[rmp->mp_parent].mp_wpid;       /* who's being waited for? */
16964     parent_waiting = mproc[rmp->mp_parent].mp_flags & WAITING;
16965     if (pidarg == -1 || pidarg == rmp->mp_pid || -pidarg == rmp->mp_procgrp)
16966           right_child = TRUE;             /* child meets one of the 3 tests */
16967     else
16968           right_child = FALSE;            /* child fails all 3 tests */
16969     if (parent_waiting && right_child)
16970           cleanup(rmp);                   /* tell parent and release child slot */
16971     else
16972           rmp->mp_flags |= HANGING;       /* parent not waiting, suspend child */
16973
16974     /* If the process has children, disinherit them.  INIT is the new parent. */
16975     for (rmp = &mproc[0]; rmp < &mproc[NR_PROCS]; rmp++) {
16976           if (rmp->mp_flags & IN_USE && rmp->mp_parent == proc_nr) {
16977                   /* 'rmp' now points to a child to be disinherited. */
16978                   rmp->mp_parent = INIT_PROC_NR;
16979                   parent_waiting = mproc[INIT_PROC_NR].mp_flags & WAITING;
16980                   if (parent_waiting && (rmp->mp_flags & HANGING)) cleanup(rmp);
16981           }
16982     }
16983
16984     /* Send a hangup to the process' process group if it was a session leader. */
16985     if (procgrp != 0) check_sig(-procgrp, SIGHUP);
16986   }


16989   /*===========================================================================*
16990    *                                do_waitpid                                 *
16991    *===========================================================================*/
16992   PUBLIC int do_waitpid()
16993   {
16994   /* A process wants to wait for a child to terminate. If one is already waiting,
16995    * go clean it up and let this WAIT call terminate.  Otherwise, really wait.
16996    * Both WAIT and WAITPID are handled by this code.
16997    */
16998
16999     register struct mproc *rp;
17000     int pidarg, options, children, res2;
17001
17002     /* A process calling WAIT never gets a reply in the usual way via the
17003      * reply() in the main loop (unless WNOHANG is set or no qualifying child
17004      * exists).  If a child has already exited, the routine cleanup() sends
```

```
17005       * the reply to awaken the caller.
17006       */
17007
17008      /* Set internal variables, depending on whether this is WAIT or WAITPID. */
17009      pidarg  = (mm_call == WAIT ? -1 : pid);       /* first param of waitpid */
17010      options = (mm_call == WAIT ?  0 : sig_nr);    /* third param of waitpid */
17011      if (pidarg == 0) pidarg = -mp->mp_procgrp;    /* pidarg < 0 ==> proc grp */
17012
17013      /* Is there a child waiting to be collected? At this point, pidarg != 0:
17014       *    pidarg  >  0 means pidarg is pid of a specific process to wait for
17015       *    pidarg == -1 means wait for any child
17016       *    pidarg  < -1 means wait for any child whose process group = -pidarg
17017       */
17018      children = 0;
17019      for (rp = &mproc[0]; rp < &mproc[NR_PROCS]; rp++) {
17020            if ( (rp->mp_flags & IN_USE) && rp->mp_parent == who) {
17021                  /* The value of pidarg determines which children qualify. */
17022                  if (pidarg  > 0 && pidarg != rp->mp_pid) continue;
17023                  if (pidarg < -1 && -pidarg != rp->mp_procgrp) continue;
17024
17025                  children++;             /* this child is acceptable */
17026                  if (rp->mp_flags & HANGING) {
17027                        /* This child meets the pid test and has exited. */
17028                        cleanup(rp);    /* this child has already exited */
17029                        dont_reply = TRUE;
17030                        return(OK);
17031                  }
17032                  if ((rp->mp_flags & STOPPED) && rp->mp_sigstatus) {
17033                        /* This child meets the pid test and is being traced.*/
17034                        res2 =  0177 | (rp->mp_sigstatus << 8);
17035                        reply(who, rp->mp_pid, res2, NIL_PTR);
17036                        dont_reply = TRUE;
17037                        rp->mp_sigstatus = 0;
17038                        return(OK);
17039                  }
17040            }
17041      }
17042
17043      /* No qualifying child has exited.  Wait for one, unless none exists. */
17044      if (children > 0) {
17045            /* At least 1 child meets the pid test exists, but has not exited. */
17046            if (options & WNOHANG) return(0);     /* parent does not want to wait */
17047            mp->mp_flags |= WAITING;              /* parent wants to wait */
17048            mp->mp_wpid = (pid_t) pidarg;         /* save pid for later */
17049            dont_reply = TRUE;                    /* do not reply now though */
17050            return(OK);                           /* yes - wait for one to exit */
17051      } else {
17052            /* No child even meets the pid test.  Return error immediately. */
17053            return(ECHILD);                       /* no - parent has no children */
17054      }
17055  }


17058  /*===========================================================================*
17059   *                                cleanup                                    *
17060   *===========================================================================*/
17061  PRIVATE void cleanup(child)
17062  register struct mproc *child;   /* tells which process is exiting */
17063  {
17064  /* Finish off the exit of a process.  The process has exited or been killed
```



```
17065   * by a signal, and its parent is waiting.
17066   */
17067
17068    int exitstatus;
17069
17070    /* Wake up the parent. */
17071    exitstatus = (child->mp_exitstatus << 8) | (child->mp_sigstatus & 0377);
17072    reply(child->mp_parent, child->mp_pid, exitstatus, NIL_PTR);
17073    mproc[child->mp_parent].mp_flags &= ~WAITING; /* parent no longer waiting */
17074
17075    /* Release the process table entry. */
17076    child->mp_flags = 0;
17077    procs_in_use--;
17078  }


++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/mm/exec.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

17100  /* This file handles the EXEC system call.  It performs the work as follows:
17101   *    - see if the permissions allow the file to be executed
17102   *    - read the header and extract the sizes
17103   *    - fetch the initial args and environment from the user space
17104   *    - allocate the memory for the new process
17105   *    - copy the initial stack from MM to the process
17106   *    - read in the text and data segments and copy to the process
17107   *    - take care of setuid and setgid bits
17108   *    - fix up 'mproc' table
17109   *    - tell kernel about EXEC
17110   *    - save offset to initial argc (for ps)
17111   *
17112   * The entry points into this file are:
17113   *   do_exec:    perform the EXEC system call
17114   *   find_share: find a process whose text segment can be shared
17115   */
17116
17117  #include "mm.h"
17118  #include <sys/stat.h>
17119  #include <minix/callnr.h>
17120  #include <a.out.h>
17121  #include <signal.h>
17122  #include <string.h>
17123  #include "mproc.h"
17124  #include "param.h"
17125
17126  FORWARD _PROTOTYPE( void load_seg, (int fd, int seg, vir_bytes seg_bytes) );
17127  FORWARD _PROTOTYPE( int new_mem, (struct mproc *sh_mp, vir_bytes text_bytes,
17128                  vir_bytes data_bytes, vir_bytes bss_bytes,
17129                  vir_bytes stk_bytes, phys_bytes tot_bytes)            );
17130  FORWARD _PROTOTYPE( void patch_ptr, (char stack [ARG_MAX ], vir_bytes base) );
17131  FORWARD _PROTOTYPE( int read_header, (int fd, int *ft, vir_bytes *text_bytes,
17132                  vir_bytes *data_bytes, vir_bytes *bss_bytes,
17133                  phys_bytes *tot_bytes, long *sym_bytes, vir_clicks sc,
17134                  vir_bytes *pc)                                        );
17135
17136
17137  /*===========================================================================*
17138   *                                do_exec                                    *
17139   *===========================================================================*/
```

```
17140   PUBLIC int do_exec()
17141   {
17142   /* Perform the execve(name, argv, envp) call.  The user library builds a
17143    * complete stack image, including pointers, args, environ, etc.  The stack
17144    * is copied to a buffer inside MM, and then to the new core image.
17145    */
17146
17147     register struct mproc *rmp;
17148     struct mproc *sh_mp;
17149     int m, r, fd, ft, sn;
17150     static char mbuf[ARG_MAX];    /* buffer for stack and zeroes */
17151     static char name_buf[PATH_MAX]; /* the name of the file to exec */
17152     char *new_sp, *basename;
17153     vir_bytes src, dst, text_bytes, data_bytes, bss_bytes, stk_bytes, vsp;
17154     phys_bytes tot_bytes;         /* total space for program, including gap */
17155     long sym_bytes;
17156     vir_clicks sc;
17157     struct stat s_buf;
17158     vir_bytes pc;
17159
17160     /* Do some validity checks. */
17161     rmp = mp;
17162     stk_bytes = (vir_bytes) stack_bytes;
17163     if (stk_bytes > ARG_MAX) return(ENOMEM);      /* stack too big */
17164     if (exec_len <= 0 || exec_len > PATH_MAX) return(EINVAL);
17165
17166     /* Get the exec file name and see if the file is executable. */
17167     src = (vir_bytes) exec_name;
17168     dst = (vir_bytes) name_buf;
17169     r = sys_copy(who, D, (phys_bytes) src,
17170                  MM_PROC_NR, D, (phys_bytes) dst, (phys_bytes) exec_len);
17171     if (r != OK) return(r);       /* file name not in user data segment */
17172     tell_fs(CHDIR, who, FALSE, 0);        /* switch to the user's FS environ. */
17173     fd = allowed(name_buf, &s_buf, X_BIT);        /* is file executable? */
17174     if (fd < 0) return(fd);       /* file was not executable */
17175
17176     /* Read the file header and extract the segment sizes. */
17177     sc = (stk_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
17178     m = read_header(fd, &ft, &text_bytes, &data_bytes, &bss_bytes,
17179                                         &tot_bytes, &sym_bytes, sc, &pc);
17180     if (m < 0) {
17181           close(fd);              /* something wrong with header */
17182           return(ENOEXEC);
17183     }
17184
17185     /* Fetch the stack from the user before destroying the old core image. */
17186     src = (vir_bytes) stack_ptr;
17187     dst = (vir_bytes) mbuf;
17188     r = sys_copy(who, D, (phys_bytes) src,
17189                         MM_PROC_NR, D, (phys_bytes) dst, (phys_bytes)stk_bytes);
17190     if (r != OK) {
17191           close(fd);              /* can't fetch stack (e.g. bad virtual addr) */
17192           return(EACCES);
17193     }
17194
17195     /* Can the process' text be shared with that of one already running? */
17196     sh_mp = find_share(rmp, s_buf.st_ino, s_buf.st_dev, s_buf.st_ctime);
17197
17198     /* Allocate new memory and release old memory.  Fix map and tell kernel. */
17199     r = new_mem(sh_mp, text_bytes, data_bytes, bss_bytes, stk_bytes, tot_bytes);
```



```
17200     if (r != OK) {
17201           close(fd);              /* insufficient core or program too big */
17202           return(r);
17203     }
17204
17205     /* Save file identification to allow it to be shared. */
17206     rmp->mp_ino = s_buf.st_ino;
17207     rmp->mp_dev = s_buf.st_dev;
17208     rmp->mp_ctime = s_buf.st_ctime;
17209
17210     /* Patch up stack and copy it from MM to new core image. */
17211     vsp = (vir_bytes) rmp->mp_seg[S].mem_vir << CLICK_SHIFT;
17212     vsp += (vir_bytes) rmp->mp_seg[S].mem_len << CLICK_SHIFT;
17213     vsp -= stk_bytes;
17214     patch_ptr(mbuf, vsp);
17215     src = (vir_bytes) mbuf;
17216     r = sys_copy(MM_PROC_NR, D, (phys_bytes) src,
17217                           who, D, (phys_bytes) vsp, (phys_bytes)stk_bytes);
17218     if (r != OK) panic("do_exec stack copy err", NO_NUM);
17219
17220     /* Read in text and data segments. */
17221     if (sh_mp != NULL) {
17222           lseek(fd, (off_t) text_bytes, SEEK_CUR);  /* shared: skip text */
17223     } else {
17224           load_seg(fd, T, text_bytes);
17225     }
17226     load_seg(fd, D, data_bytes);
17227
17228
17229     close(fd);                    /* don't need exec file any more */
17230
17231     /* Take care of setuid/setgid bits. */
17232     if ((rmp->mp_flags & TRACED) == 0) { /* suppress if tracing */
17233           if (s_buf.st_mode & I_SET_UID_BIT) {
17234                 rmp->mp_effuid = s_buf.st_uid;
17235                 tell_fs(SETUID,who, (int)rmp->mp_realuid, (int)rmp->mp_effuid);
17236           }
17237           if (s_buf.st_mode & I_SET_GID_BIT) {
17238                 rmp->mp_effgid = s_buf.st_gid;
17239                 tell_fs(SETGID,who, (int)rmp->mp_realgid, (int)rmp->mp_effgid);
17240           }
17241     }
17242
17243     /* Save offset to initial argc (for ps) */
17244     rmp->mp_procargs = vsp;
17245
17246     /* Fix 'mproc' fields, tell kernel that exec is done,  reset caught sigs. */
17247     for (sn = 1; sn <= _NSIG; sn++) {
17248           if (sigismember(&rmp->mp_catch, sn)) {
17249                 sigdelset(&rmp->mp_catch, sn);
17250                 rmp->mp_sigact[sn].sa_handler = SIG_DFL;
17251                 sigemptyset(&rmp->mp_sigact[sn].sa_mask);
17252           }
17253     }
17254
17255     rmp->mp_flags &= ~SEPARATE;   /* turn off SEPARATE bit */
17256     rmp->mp_flags |= ft;          /* turn it on for separate I & D files */
17257     new_sp = (char *) vsp;
17258
17259     tell_fs(EXEC, who, 0, 0);     /* allow FS to handle FD_CLOEXEC files */
```

```
17260
17261     /* System will save command line for debugging, ps(1) output, etc. */
17262     basename = strrchr(name_buf, '/');
17263     if (basename == NULL) basename = name_buf; else basename++;
17264     sys_exec(who, new_sp, rmp->mp_flags & TRACED, basename, pc);
17265     return(OK);
17266   }


17269   /*===========================================================================*
17270    *                              read_header                                  *
17271    *===========================================================================*/
17272   PRIVATE int read_header(fd, ft, text_bytes, data_bytes, bss_bytes,
17273                                           tot_bytes, sym_bytes, sc, pc)
17274   int fd;                       /* file descriptor for reading exec file */
17275   int *ft;                      /* place to return ft number */
17276   vir_bytes *text_bytes;        /* place to return text size */
17277   vir_bytes *data_bytes;        /* place to return initialized data size */
17278   vir_bytes *bss_bytes;         /* place to return bss size */
17279   phys_bytes *tot_bytes;        /* place to return total size */
17280   long *sym_bytes;              /* place to return symbol table size */
17281   vir_clicks sc;                /* stack size in clicks */
17282   vir_bytes *pc;                /* program entry point (initial PC) */
17283   {
17284   /* Read the header and extract the text, data, bss and total sizes from it. */
17285
17286     int m, ct;
17287     vir_clicks tc, dc, s_vir, dvir;
17288     phys_clicks totc;
17289     struct exec hdr;            /* a.out header is read in here */
17290
17291     /* Read the header and check the magic number.  The standard MINIX header
17292      * is defined in <a.out.h>.  It consists of 8 chars followed by 6 longs.
17293      * Then come 4 more longs that are not used here.
17294      *    Byte 0: magic number 0x01
17295      *    Byte 1: magic number 0x03
17296      *    Byte 2: normal = 0x10 (not checked, 0 is OK), separate I/D = 0x20
17297      *    Byte 3: CPU type, Intel 16 bit = 0x04, Intel 32 bit = 0x10,
17298      *            Motorola = 0x0B, Sun SPARC = 0x17
17299      *    Byte 4: Header length = 0x20
17300      *    Bytes 5-7 are not used.
17301      *
17302      *    Now come the 6 longs
17303      *    Bytes  8-11: size of text segments in bytes
17304      *    Bytes 12-15: size of initialized data segment in bytes
17305      *    Bytes 16-19: size of bss in bytes
17306      *    Bytes 20-23: program entry point
17307      *    Bytes 24-27: total memory allocated to program (text, data + stack)
17308      *    Bytes 28-31: size of symbol table in bytes
17309      * The longs are represented in a machine dependent order,
17310      * little-endian on the 8088, big-endian on the 68000.
17311      * The header is followed directly by the text and data segments, and the
17312      * symbol table (if any). The sizes are given in the header. Only the
17313      * text and data segments are copied into memory by exec. The header is
17314      * used here only. The symbol table is for the benefit of a debugger and
17315      * is ignored here.
17316      */
17317
17318     if (read(fd, (char *) &hdr, A_MINHDR) != A_MINHDR) return(ENOEXEC);
17319
```



```
17320     /* Check magic number, cpu type, and flags. */
17321     if (BADMAG(hdr)) return(ENOEXEC);
17322   #if (CHIP == INTEL && _WORD_SIZE == 2)
17323     if (hdr.a_cpu != A_I8086) return(ENOEXEC);
17324   #endif
17325   #if (CHIP == INTEL && _WORD_SIZE == 4)
17326     if (hdr.a_cpu != A_I80386) return(ENOEXEC);
17327   #endif
17328     if ((hdr.a_flags & ~(A_NSYM | A_EXEC | A_SEP)) != 0) return(ENOEXEC);
17329
17330     *ft = ( (hdr.a_flags & A_SEP) ? SEPARATE : 0);    /* separate I & D or not */
17331
17332     /* Get text and data sizes. */
17333     *text_bytes = (vir_bytes) hdr.a_text; /* text size in bytes */
17334     *data_bytes = (vir_bytes) hdr.a_data; /* data size in bytes */
17335     *bss_bytes  = (vir_bytes) hdr.a_bss;  /* bss size in bytes */
17336     *tot_bytes  = hdr.a_total;            /* total bytes to allocate for prog */
17337     *sym_bytes  = hdr.a_syms;             /* symbol table size in bytes */
17338     if (*tot_bytes == 0) return(ENOEXEC);
17339
17340     if (*ft != SEPARATE) {
17341
17342           /* If I & D space is not separated, it is all considered data. Text=0*/
17343           *data_bytes += *text_bytes;
17344           *text_bytes = 0;
17345
17346     }
17347     *pc = hdr.a_entry;    /* initial address to start execution */
17348
17349     /* Check to see if segment sizes are feasible. */
17350     tc = ((unsigned long) *text_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
17351     dc = (*data_bytes + *bss_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
17352     totc = (*tot_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
17353     if (dc >= totc) return(ENOEXEC);      /* stack must be at least 1 click */
17354     dvir = (*ft == SEPARATE ? 0 : tc);
17355     s_vir = dvir + (totc - sc);
17356     m = size_ok(*ft, tc, dc, sc, dvir, s_vir);
17357     ct = hdr.a_hdrlen & BYTE;             /* header length */
17358     if (ct > A_MINHDR) lseek(fd, (off_t) ct, SEEK_SET); /* skip unused hdr */
17359     return(m);
17360   }


17363   /*===========================================================================*
17364    *                              new_mem                                      *
17365    *===========================================================================*/
17366   PRIVATE int new_mem(sh_mp, text_bytes, data_bytes,bss_bytes,stk_bytes,tot_bytes)
17367   struct mproc *sh_mp;          /* text can be shared with this process */
17368   vir_bytes text_bytes;         /* text segment size in bytes */
17369   vir_bytes data_bytes;         /* size of initialized data in bytes */
17370   vir_bytes bss_bytes;          /* size of bss in bytes */
17371   vir_bytes stk_bytes;          /* size of initial stack segment in bytes */
17372   phys_bytes tot_bytes;         /* total memory to allocate, including gap */
17373   {
17374   /* Allocate new memory and release the old memory.  Change the map and report
17375    * the new map to the kernel.  Zero the new core image's bss, gap and stack.
17376    */
17377
17378     register struct mproc *rmp;
17379     vir_clicks text_clicks, data_clicks, gap_clicks, stack_clicks, tot_clicks;
```

```
17380     phys_clicks new_base;
17381
17382     static char zero[1024];               /* used to zero bss */
17383     phys_bytes bytes, base, count, bss_offset;
17384
17385     /* No need to allocate text if it can be shared. */
17386     if (sh_mp != NULL) text_bytes = 0;
17387
17388     /* Acquire the new memory.  Each of the 4 parts: text, (data+bss), gap,
17389      * and stack occupies an integral number of clicks, starting at click
17390      * boundary.  The data and bss parts are run together with no space.
17391      */
17392
17393     text_clicks = ((unsigned long) text_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
17394     data_clicks = (data_bytes + bss_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
17395     stack_clicks = (stk_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
17396     tot_clicks = (tot_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
17397     gap_clicks = tot_clicks - data_clicks - stack_clicks;
17398     if ( (int) gap_clicks < 0) return(ENOMEM);
17399
17400     /* Check to see if there is a hole big enough.  If so, we can risk first
17401      * releasing the old core image before allocating the new one, since we
17402      * know it will succeed.  If there is not enough, return failure.
17403      */
17404     if (text_clicks + tot_clicks > max_hole()) return(EAGAIN);
17405
17406     /* There is enough memory for the new core image.  Release the old one. */
17407     rmp = mp;
17408
17409     if (find_share(rmp, rmp->mp_ino, rmp->mp_dev, rmp->mp_ctime) == NULL) {
17410           /* No other process shares the text segment, so free it. */
17411           free_mem(rmp->mp_seg[T].mem_phys, rmp->mp_seg[T].mem_len);
17412     }
17413     /* Free the data and stack segments. */
17414     free_mem(rmp->mp_seg[D].mem_phys,
17415         rmp->mp_seg[S].mem_vir + rmp->mp_seg[S].mem_len - rmp->mp_seg[D].mem_vir);
17416
17417     /* We have now passed the point of no return.  The old core image has been
17418      * forever lost.  The call must go through now.  Set up and report new map.
17419      */
17420     new_base = alloc_mem(text_clicks + tot_clicks);       /* new core image */
17421     if (new_base == NO_MEM) panic("MM hole list is inconsistent", NO_NUM);
17422
17423     if (sh_mp != NULL) {
17424           /* Share the text segment. */
17425           rmp->mp_seg[T] = sh_mp->mp_seg[T];
17426     } else {
17427           rmp->mp_seg[T].mem_phys = new_base;
17428           rmp->mp_seg[T].mem_vir = 0;
17429           rmp->mp_seg[T].mem_len = text_clicks;
17430     }
17431     rmp->mp_seg[D].mem_phys = new_base + text_clicks;
17432     rmp->mp_seg[D].mem_vir = 0;
17433     rmp->mp_seg[D].mem_len = data_clicks;
17434     rmp->mp_seg[S].mem_phys = rmp->mp_seg[D].mem_phys + data_clicks + gap_clicks;
17435     rmp->mp_seg[S].mem_vir = rmp->mp_seg[D].mem_vir + data_clicks + gap_clicks;
17436     rmp->mp_seg[S].mem_len = stack_clicks;
17437
17438
17439     sys_newmap(who, rmp->mp_seg);   /* report new map to the kernel */
```



```
17440
17441     /* Zero the bss, gap, and stack segment. */
17442     bytes = (phys_bytes)(data_clicks + gap_clicks + stack_clicks) << CLICK_SHIFT;
17443     base = (phys_bytes) rmp->mp_seg[D].mem_phys << CLICK_SHIFT;
17444     bss_offset = (data_bytes >> CLICK_SHIFT) << CLICK_SHIFT;
17445     base += bss_offset;
17446     bytes -= bss_offset;
17447
17448     while (bytes > 0) {
17449           count = MIN(bytes, (phys_bytes) sizeof(zero));
17450           if (sys_copy(MM_PROC_NR, D, (phys_bytes) zero,
17451                                                 ABS, 0, base, count) != OK) {
17452                 panic("new_mem can't zero", NO_NUM);
17453           }
17454           base += count;
17455           bytes -= count;
17456     }
17457
17458     return(OK);
17459   }


17462   /*============================================================================*
17463    *                              patch_ptr                                     *
17464    *============================================================================*/
17465   PRIVATE void patch_ptr(stack, base)
17466   char stack[ARG_MAX];    /* pointer to stack image within MM */
17467   vir_bytes base;               /* virtual address of stack base inside user */
17468   {
17469   /* When doing an exec(name, argv, envp) call, the user builds up a stack
17470    * image with arg and env pointers relative to the start of the stack.  Now
17471    * these pointers must be relocated, since the stack is not positioned at
17472    * address 0 in the user's address space.
17473    */
17474
17475     char **ap, flag;
17476     vir_bytes v;
17477
17478     flag = 0;                     /* counts number of 0-pointers seen */
17479     ap = (char **) stack;         /* points initially to 'nargs' */
17480     ap++;                         /* now points to argv[0] */
17481     while (flag < 2) {
17482           if (ap >= (char **) &stack[ARG_MAX]) return;    /* too bad */
17483           if (*ap != NIL_PTR) {
17484                 v = (vir_bytes) *ap;    /* v is relative pointer */
17485                 v += base;              /* relocate it */
17486                 *ap = (char *) v;       /* put it back */
17487           } else {
17488                 flag++;
17489           }
17490           ap++;
17491     }
17492   }


17495   /*============================================================================*
17496    *                              load_seg                                      *
17497    *============================================================================*/
17498   PRIVATE void load_seg(fd, seg, seg_bytes)
17499   int fd;                       /* file descriptor to read from */
```

```
17500   int seg;                        /* T or D */
17501   vir_bytes seg_bytes;            /* how big is the segment */
17502   {
17503   /* Read in text or data from the exec file and copy to the new core image.
17504    * This procedure is a little bit tricky.  The logical way to load a segment
17505    * would be to read it block by block and copy each block to the user space
17506    * one at a time.  This is too slow, so we do something dirty here, namely
17507    * send the user space and virtual address to the file system in the upper
17508    * 10 bits of the file descriptor, and pass it the user virtual address
17509    * instead of a MM address.  The file system extracts these parameters when
17510    * gets a read call from the memory manager, which is the only process that
17511    * is permitted to use this trick.  The file system then copies the whole
17512    * segment directly to user space, bypassing MM completely.
17513    */
17514
17515     int new_fd, bytes;
17516     char *ubuf_ptr;
17517
17518     new_fd = (who << 8) | (seg << 6) | fd;
17519     ubuf_ptr = (char *) ((vir_bytes)mp->mp_seg[seg].mem_vir << CLICK_SHIFT);
17520     while (seg_bytes != 0) {
17521           bytes = (INT_MAX / BLOCK_SIZE) * BLOCK_SIZE;
17522           if (seg_bytes < bytes)
17523                   bytes = (int)seg_bytes;
17524           if (read(new_fd, ubuf_ptr, bytes) != bytes)
17525                   break;          /* error */
17526           ubuf_ptr += bytes;
17527           seg_bytes -= bytes;
17528     }
17529   }


17532   /*===========================================================================*
17533    *                               find_share                                  *
17534    *===========================================================================*/
17535   PUBLIC struct mproc *find_share(mp_ign, ino, dev, ctime)
17536   struct mproc *mp_ign;           /* process that should not be looked at */
17537   ino_t ino;                      /* parameters that uniquely identify a file */
17538   dev_t dev;
17539   time_t ctime;
17540   {
17541   /* Look for a process that is the file <ino, dev, ctime> in execution.  Don't
17542    * accidentally "find" mp_ign, because it is the process on whose behalf this
17543    * call is made.
17544    */
17545     struct mproc *sh_mp;
17546
17547     for (sh_mp = &mproc[INIT_PROC_NR]; sh_mp < &mproc[NR_PROCS]; sh_mp++) {
17548           if ((sh_mp->mp_flags & (IN_USE | HANGING | SEPARATE))
17549                                       != (IN_USE | SEPARATE)) continue;
17550           if (sh_mp == mp_ign) continue;
17551           if (sh_mp->mp_ino != ino) continue;
17552           if (sh_mp->mp_dev != dev) continue;
17553           if (sh_mp->mp_ctime != ctime) continue;
17554           return sh_mp;
17555     }
17556     return(NULL);
17557   }
```



```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/mm/break.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

17600   /* The MINIX model of memory allocation reserves a fixed amount of memory for
17601    * the combined text, data, and stack segments.  The amount used for a child
17602    * process created by FORK is the same as the parent had.  If the child does
17603    * an EXEC later, the new size is taken from the header of the file EXEC'ed.
17604    *
17605    * The layout in memory consists of the text segment, followed by the data
17606    * segment, followed by a gap (unused memory), followed by the stack segment.
17607    * The data segment grows upward and the stack grows downward, so each can
17608    * take memory from the gap.  If they meet, the process must be killed.  The
17609    * procedures in this file deal with the growth of the data and stack segments.
17610    *
17611    * The entry points into this file are:
17612    *   do_brk:      BRK/SBRK system calls to grow or shrink the data segment
17613    *   adjust:      see if a proposed segment adjustment is allowed
17614    *   size_ok:     see if the segment sizes are feasible
17615    */
17616
17617   #include "mm.h"
17618   #include <signal.h>
17619   #include "mproc.h"
17620   #include "param.h"
17621
17622   #define DATA_CHANGED       1    /* flag value when data segment size changed */
17623   #define STACK_CHANGED      2    /* flag value when stack size changed */
17624
17625   /*===========================================================================*
17626    *                                do_brk                                     *
17627    *===========================================================================*/
17628   PUBLIC int do_brk()
17629   {
17630   /* Perform the brk(addr) system call.
17631    *
17632    * The call is complicated by the fact that on some machines (e.g., 8088),
17633    * the stack pointer can grow beyond the base of the stack segment without
17634    * anybody noticing it.
17635    * The parameter, 'addr' is the new virtual address in D space.
17636    */
17637
17638     register struct mproc *rmp;
17639     int r;
17640     vir_bytes v, new_sp;
17641     vir_clicks new_clicks;
17642
17643     rmp = mp;
17644     v = (vir_bytes) addr;
17645     new_clicks = (vir_clicks) ( ((long) v + CLICK_SIZE - 1) >> CLICK_SHIFT);
17646     if (new_clicks < rmp->mp_seg[D].mem_vir) {
17647           res_ptr = (char *) -1;
17648           return(ENOMEM);
17649     }
17650     new_clicks -= rmp->mp_seg[D].mem_vir;
17651     sys_getsp(who, &new_sp);      /* ask kernel for current sp value */
17652     r = adjust(rmp, new_clicks, new_sp);
17653     res_ptr = (r == OK ? addr : (char *) -1);
17654     return(r);                    /* return new address or -1 */
```

```
17655   }


17658   /*===========================================================================*
17659    *                              adjust                                       *
17660    *===========================================================================*/
17661   PUBLIC int adjust(rmp, data_clicks, sp)
17662   register struct mproc *rmp;     /* whose memory is being adjusted? */
17663   vir_clicks data_clicks;         /* how big is data segment to become? */
17664   vir_bytes sp;                   /* new value of sp */
17665   {
17666   /* See if data and stack segments can coexist, adjusting them if need be.
17667    * Memory is never allocated or freed.  Instead it is added or removed from the
17668    * gap between data segment and stack segment.  If the gap size becomes
17669    * negative, the adjustment of data or stack fails and ENOMEM is returned.
17670    */
17671
17672     register struct mem_map *mem_sp, *mem_dp;
17673     vir_clicks sp_click, gap_base, lower, old_clicks;
17674     int changed, r, ft;
17675     long base_of_stack, delta;    /* longs avoid certain problems */
17676
17677     mem_dp = &rmp->mp_seg[D];     /* pointer to data segment map */
17678     mem_sp = &rmp->mp_seg[S];     /* pointer to stack segment map */
17679     changed = 0;                  /* set when either segment changed */
17680
17681     if (mem_sp->mem_len == 0) return(OK); /* don't bother init */
17682
17683     /* See if stack size has gone negative (i.e., sp too close to 0xFFFF...) */
17684     base_of_stack = (long) mem_sp->mem_vir + (long) mem_sp->mem_len;
17685     sp_click = sp >> CLICK_SHIFT; /* click containing sp */
17686     if (sp_click >= base_of_stack) return(ENOMEM);        /* sp too high */
17687
17688     /* Compute size of gap between stack and data segments. */
17689     delta = (long) mem_sp->mem_vir - (long) sp_click;
17690     lower = (delta > 0 ? sp_click : mem_sp->mem_vir);
17691
17692     /* Add a safety margin for future stack growth. Impossible to do right. */
17693   #define SAFETY_BYTES  (384 * sizeof(char *))
17694   #define SAFETY_CLICKS ((SAFETY_BYTES + CLICK_SIZE - 1) / CLICK_SIZE)
17695     gap_base = mem_dp->mem_vir + data_clicks + SAFETY_CLICKS;
17696     if (lower < gap_base) return(ENOMEM); /* data and stack collided */
17697
17698     /* Update data length (but not data orgin) on behalf of brk() system call. */
17699     old_clicks = mem_dp->mem_len;
17700     if (data_clicks != mem_dp->mem_len) {
17701           mem_dp->mem_len = data_clicks;
17702           changed |= DATA_CHANGED;
17703     }
17704
17705     /* Update stack length and origin due to change in stack pointer. */
17706     if (delta > 0) {
17707           mem_sp->mem_vir -= delta;
17708           mem_sp->mem_phys -= delta;
17709           mem_sp->mem_len += delta;
17710           changed |= STACK_CHANGED;
17711     }
17712
17713     /* Do the new data and stack segment sizes fit in the address space? */
17714     ft = (rmp->mp_flags & SEPARATE);
```



```
17715     r = size_ok(ft, rmp->mp_seg[T].mem_len, rmp->mp_seg[D].mem_len,
17716           rmp->mp_seg[S].mem_len, rmp->mp_seg[D].mem_vir, rmp->mp_seg[S].mem_vir);
17717     if (r == OK) {
17718           if (changed) sys_newmap((int)(rmp - mproc), rmp->mp_seg);
17719           return(OK);
17720     }
17721
17722     /* New sizes don't fit or require too many page/segment registers. Restore.*/
17723     if (changed & DATA_CHANGED) mem_dp->mem_len = old_clicks;
17724     if (changed & STACK_CHANGED) {
17725           mem_sp->mem_vir += delta;
17726           mem_sp->mem_phys += delta;
17727           mem_sp->mem_len -= delta;
17728     }
17729     return(ENOMEM);
17730   }


17733   /*===========================================================================*
17734    *                              size_ok                                      *
17735    *===========================================================================*/
17736   PUBLIC int size_ok(file_type, tc, dc, sc, dvir, s_vir)
17737   int file_type;                  /* SEPARATE or 0 */
17738   vir_clicks tc;                  /* text size in clicks */
17739   vir_clicks dc;                  /* data size in clicks */
17740   vir_clicks sc;                  /* stack size in clicks */
17741   vir_clicks dvir;                /* virtual address for start of data seg */
17742   vir_clicks s_vir;               /* virtual address for start of stack seg */
17743   {
17744   /* Check to see if the sizes are feasible and enough segmentation registers
17745    * exist.  On a machine with eight 8K pages, text, data, stack sizes of
17746    * (32K, 16K, 16K) will fit, but (33K, 17K, 13K) will not, even though the
17747    * former is bigger (64K) than the latter (63K).  Even on the 8088 this test
17748    * is needed, since the data and stack may not exceed 4096 clicks.
17749    */
17750
17751   #if (CHIP == INTEL && _WORD_SIZE == 2)
17752     int pt, pd, ps;               /* segment sizes in pages */
17753
17754     pt = ( (tc << CLICK_SHIFT) + PAGE_SIZE - 1)/PAGE_SIZE;
17755     pd = ( (dc << CLICK_SHIFT) + PAGE_SIZE - 1)/PAGE_SIZE;
17756     ps = ( (sc << CLICK_SHIFT) + PAGE_SIZE - 1)/PAGE_SIZE;
17757
17758     if (file_type == SEPARATE) {
17759           if (pt > MAX_PAGES || pd + ps > MAX_PAGES) return(ENOMEM);
17760     } else {
17761           if (pt + pd + ps > MAX_PAGES) return(ENOMEM);
17762     }
17763   #endif
17764
17765     if (dvir + dc > s_vir) return(ENOMEM);
17766
17767     return(OK);
17768   }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                              src/mm/signal.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

17800   /* This file handles signals, which are asynchronous events and are generally
17801    * a messy and unpleasant business.  Signals can be generated by the KILL
17802    * system call, or from the keyboard (SIGINT) or from the clock (SIGALRM).
17803    * In all cases control eventually passes to check_sig() to see which processes
17804    * can be signaled.  The actual signaling is done by sig_proc().
17805    *
17806    * The entry points into this file are:
17807    *   do_sigaction:   perform the SIGACTION system call
17808    *   do_sigpending:  perform the SIGPENDING system call
17809    *   do_sigprocmask: perform the SIGPROCMASK system call
17810    *   do_sigreturn:   perform the SIGRETURN system call
17811    *   do_sigsuspend:  perform the SIGSUSPEND system call
17812    *   do_kill:   perform the KILL system call
17813    *   do_ksig:   accept a signal originating in the kernel (e.g., SIGINT)
17814    *   do_alarm:  perform the ALARM system call by calling set_alarm()
17815    *   set_alarm: tell the clock task to start or stop a timer
17816    *   do_pause:  perform the PAUSE system call
17817    *   do_reboot: kill all processes, then reboot system
17818    *   sig_proc:  interrupt or terminate a signaled process
17819    *   check_sig: check which processes to signal with sig_proc()
17820    */
17821
17822   #include "mm.h"
17823   #include <sys/stat.h>
17824   #include <minix/callnr.h>
17825   #include <minix/com.h>
17826   #include <signal.h>
17827   #include <sys/sigcontext.h>
17828   #include <string.h>
17829   #include "mproc.h"
17830   #include "param.h"
17831
17832   #define CORE_MODE       0777    /* mode to use on core image files */
17833   #define DUMPED          0200    /* bit set in status when core dumped */
17834   #define DUMP_SIZE       ((INT_MAX / BLOCK_SIZE) * BLOCK_SIZE)
17835                                   /* buffer size for core dumps */
17836
17837   FORWARD _PROTOTYPE( void check_pending, (void)                          );
17838   FORWARD _PROTOTYPE( void dump_core, (struct mproc *rmp)                 );
17839   FORWARD _PROTOTYPE( void unpause, (int pro)                             );
17840
17841
17842   /*===========================================================================*
17843    *                              do_sigaction                                 *
17844    *===========================================================================*/
17845   PUBLIC int do_sigaction()
17846   {
17847     int r;
17848     struct sigaction svec;
17849     struct sigaction *svp;
17850
17851     if (sig_nr == SIGKILL) return(OK);
17852     if (sig_nr < 1 || sig_nr > _NSIG) return (EINVAL);
17853     svp = &mp->mp_sigact[sig_nr];
17854     if ((struct sigaction *) sig_osa != (struct sigaction *) NULL) {
```



```
17855           r = sys_copy(MM_PROC_NR,D, (phys_bytes) svp,
17856                   who, D, (phys_bytes) sig_osa, (phys_bytes) sizeof(svec));
17857           if (r != OK) return(r);
17858       }
17859
17860       if ((struct sigaction *) sig_nsa == (struct sigaction *) NULL) return(OK);
17861
17862       /* Read in the sigaction structure. */
17863       r = sys_copy(who, D, (phys_bytes) sig_nsa,
17864                   MM_PROC_NR, D, (phys_bytes) &svec, (phys_bytes) sizeof(svec));
17865       if (r != OK) return(r);
17866
17867       if (svec.sa_handler == SIG_IGN) {
17868           sigaddset(&mp->mp_ignore, sig_nr);
17869           sigdelset(&mp->mp_sigpending, sig_nr);
17870           sigdelset(&mp->mp_catch, sig_nr);
17871       } else {
17872           sigdelset(&mp->mp_ignore, sig_nr);
17873           if (svec.sa_handler == SIG_DFL)
17874                   sigdelset(&mp->mp_catch, sig_nr);
17875           else
17876                   sigaddset(&mp->mp_catch, sig_nr);
17877       }
17878       mp->mp_sigact[sig_nr].sa_handler = svec.sa_handler;
17879       sigdelset(&svec.sa_mask, SIGKILL);
17880       mp->mp_sigact[sig_nr].sa_mask = svec.sa_mask;
17881       mp->mp_sigact[sig_nr].sa_flags = svec.sa_flags;
17882       mp->mp_sigreturn = (vir_bytes) sig_ret;
17883       return(OK);
17884   }

17886   /*===========================================================================*
17887    *                              do_sigpending                                *
17888    *===========================================================================*/
17889   PUBLIC int do_sigpending()
17890   {
17891     ret_mask = (long) mp->mp_sigpending;
17892     return OK;
17893   }

17895   /*===========================================================================*
17896    *                              do_sigprocmask                               *
17897    *===========================================================================*/
17898   PUBLIC int do_sigprocmask()
17899   {
17900   /* Note that the library interface passes the actual mask in sigmask_set,
17901    * not a pointer to the mask, in order to save a sys_copy.  Similarly,
17902    * the old mask is placed in the return message which the library
17903    * interface copies (if requested) to the user specified address.
17904    *
17905    * The library interface must set SIG_INQUIRE if the 'act' argument
17906    * is NULL.
17907    */
17908
17909     int i;
17910
17911     ret_mask = (long) mp->mp_sigmask;
17912
17913     switch (sig_how) {
17914         case SIG_BLOCK:
```

```
17915           sigdelset((sigset_t *)&sig_set, SIGKILL);
17916           for (i = 1; i < _NSIG; i++) {
17917                   if (sigismember((sigset_t *)&sig_set, i))
17918                           sigaddset(&mp->mp_sigmask, i);
17919           }
17920           break;
17921
17922         case SIG_UNBLOCK:
17923           for (i = 1; i < _NSIG; i++) {
17924                   if (sigismember((sigset_t *)&sig_set, i))
17925                           sigdelset(&mp->mp_sigmask, i);
17926           }
17927           check_pending();
17928           break;
17929
17930         case SIG_SETMASK:
17931           sigdelset((sigset_t *)&sig_set, SIGKILL);
17932           mp->mp_sigmask = (sigset_t)sig_set;
17933           check_pending();
17934           break;
17935
17936         case SIG_INQUIRE:
17937           break;
17938
17939         default:
17940           return(EINVAL);
17941           break;
17942     }
17943     return OK;
17944   }

17946   /*===========================================================================*
17947    *                              do_sigsuspend                                *
17948    *===========================================================================*/
17949   PUBLIC int do_sigsuspend()
17950   {
17951     mp->mp_sigmask2 = mp->mp_sigmask;     /* save the old mask */
17952     mp->mp_sigmask = (sigset_t) sig_set;
17953     sigdelset(&mp->mp_sigmask, SIGKILL);
17954     mp->mp_flags |= SIGSUSPENDED;
17955     dont_reply = TRUE;
17956     check_pending();
17957     return OK;
17958   }


17961   /*===========================================================================*
17962    *                              do_sigreturn                                 *
17963    *===========================================================================*/
17964   PUBLIC int do_sigreturn()
17965   {
17966   /* A user signal handler is done.  Restore context and check for
17967    * pending unblocked signals.
17968    */
17969
17970     int r;
17971
17972     mp->mp_sigmask = (sigset_t) sig_set;
17973     sigdelset(&mp->mp_sigmask, SIGKILL);
17974
```



```
17975     r = sys_sigreturn(who, (vir_bytes)sig_context, sig_flags);
17976     check_pending();
17977     return(r);
17978   }

17980   /*===========================================================================*
17981    *                              do_kill                                      *
17982    *===========================================================================*/
17983   PUBLIC int do_kill()
17984   {
17985   /* Perform the kill(pid, signo) system call. */
17986
17987     return check_sig(pid, sig_nr);
17988   }


17991   /*===========================================================================*
17992    *                              do_ksig                                      *
17993    *===========================================================================*/
17994   PUBLIC int do_ksig()
17995   {
17996   /* Certain signals, such as segmentation violations and DEL, originate in the
17997    * kernel.  When the kernel detects such signals, it sets bits in a bit map.
17998    * As soon as MM is awaiting new work, the kernel sends MM a message containing
17999    * the process slot and bit map.  That message comes here.  The File System
18000    * also uses this mechanism to signal writing on broken pipes (SIGPIPE).
18001    */
18002
18003     register struct mproc *rmp;
18004     int i, proc_nr;
18005     pid_t proc_id, id;
18006     sigset_t sig_map;
18007
18008     /* Only kernel may make this call. */
18009     if (who != HARDWARE) return(EPERM);
18010     dont_reply = TRUE;            /* don't reply to the kernel */
18011     proc_nr = mm_in.SIG_PROC;
18012     rmp = &mproc[proc_nr];
18013     if ( (rmp->mp_flags & IN_USE) == 0 || (rmp->mp_flags & HANGING) ) return(OK);
18014     proc_id = rmp->mp_pid;
18015     sig_map = (sigset_t) mm_in.SIG_MAP;
18016     mp = &mproc[0];               /* pretend kernel signals are from MM */
18017     mp->mp_procgrp = rmp->mp_procgrp;     /* get process group right */
18018
18019     /* Check each bit in turn to see if a signal is to be sent.  Unlike
18020      * kill(), the kernel may collect several unrelated signals for a
18021      * process and pass them to MM in one blow.  Thus loop on the bit
18022      * map. For SIGINT and SIGQUIT, use proc_id 0 to indicate a broadcast
18023      * to the recipient's process group.  For SIGKILL, use proc_id -1 to
18024      * indicate a systemwide broadcast.
18025      */
18026     for (i = 1; i <= _NSIG; i++) {
18027           if (!sigismember(&sig_map, i)) continue;
18028           switch (i) {
18029               case SIGINT:
18030               case SIGQUIT:
18031                   id = 0; break;  /* broadcast to process group */
18032               case SIGKILL:
18033                   id = -1; break; /* broadcast to all except INIT */
18034               case SIGALRM:
```

```
18035                   /* Disregard SIGALRM when the target process has not
18036                    * requested an alarm.  This only applies for a KERNEL
18037                    * generated signal.
18038                    */
18039                   if ((rmp->mp_flags & ALARM_ON) == 0) continue;
18040                   rmp->mp_flags &= ~ALARM_ON;
18041                   /* fall through */
18042               default:
18043                   id = proc_id;
18044                   break;
18045           }
18046           check_sig(id, i);
18047           sys_endsig(proc_nr);    /* tell kernel it's done */
18048       }
18049       return(OK);
18050   }


18053   /*===========================================================================*
18054    *                              do_alarm                                     *
18055    *===========================================================================*/
18056   PUBLIC int do_alarm()
18057   {
18058   /* Perform the alarm(seconds) system call. */
18059
18060     return(set_alarm(who, seconds));
18061   }


18064   /*===========================================================================*
18065    *                              set_alarm                                    *
18066    *===========================================================================*/
18067   PUBLIC int set_alarm(proc_nr, sec)
18068   int proc_nr;                    /* process that wants the alarm */
18069   int sec;                        /* how many seconds delay before the signal */
18070   {
18071   /* This routine is used by do_alarm() to set the alarm timer.  It is also used
18072    * to turn the timer off when a process exits with the timer still on.
18073    */
18074
18075     message m_sig;
18076     int remaining;
18077
18078     if (sec != 0)
18079           mproc[proc_nr].mp_flags |= ALARM_ON;
18080     else
18081           mproc[proc_nr].mp_flags &= ~ALARM_ON;
18082
18083     /* Tell the clock task to provide a signal message when the time comes.
18084      *
18085      * Large delays cause a lot of problems.  First, the alarm system call
18086      * takes an unsigned seconds count and the library has cast it to an int.
18087      * That probably works, but on return the library will convert "negative"
18088      * unsigneds to errors.  Presumably no one checks for these errors, so
18089      * force this call through.  Second, If unsigned and long have the same
18090      * size, converting from seconds to ticks can easily overflow.  Finally,
18091      * the kernel has similar overflow bugs adding ticks.
18092      *
18093      * Fixing this requires a lot of ugly casts to fit the wrong interface
18094      * types and to avoid overflow traps.  DELTA_TICKS has the right type
```



```
18095      * (clock_t) although it is declared as long.  How can variables like
18096      * this be declared properly without combinatorial explosion of message
18097      * types?
18098      */
18099     m_sig.m_type = SET_ALARM;
18100     m_sig.CLOCK_PROC_NR = proc_nr;
18101     m_sig.DELTA_TICKS = (clock_t) (HZ * (unsigned long) (unsigned) sec);
18102     if ( (unsigned long) m_sig.DELTA_TICKS / HZ != (unsigned) sec)
18103           m_sig.DELTA_TICKS = LONG_MAX;   /* eternity (really CLOCK_T_MAX) */
18104     if (sendrec(CLOCK, &m_sig) != OK) panic("alarm er", NO_NUM);
18105     remaining = (int) m_sig.SECONDS_LEFT;
18106     if (remaining != m_sig.SECONDS_LEFT || remaining < 0)
18107           remaining = INT_MAX;    /* true value is not representable */
18108     return(remaining);
18109   }


18112   /*===========================================================================*
18113    *                              do_pause                                     *
18114    *===========================================================================*/
18115   PUBLIC int do_pause()
18116   {
18117   /* Perform the pause() system call. */
18118
18119     mp->mp_flags |= PAUSED;
18120     dont_reply = TRUE;
18121     return(OK);
18122   }


18125   /*===========================================================================*
18126    *                              do_reboot                                    *
18127    *===========================================================================*/
18128   PUBLIC int do_reboot()
18129   {
18130     register struct mproc *rmp = mp;
18131     char monitor_code[64];
18132
18133     if (rmp->mp_effuid != SUPER_USER)   return EPERM;
18134
18135     switch (reboot_flag) {
18136     case RBT_HALT:
18137     case RBT_REBOOT:
18138     case RBT_PANIC:
18139     case RBT_RESET:
18140           break;
18141     case RBT_MONITOR:
18142           if (reboot_size > sizeof(monitor_code)) return EINVAL;
18143           memset(monitor_code, 0, sizeof(monitor_code));
18144           if (sys_copy(who, D, (phys_bytes) reboot_code,
18145                   MM_PROC_NR, D, (phys_bytes) monitor_code,
18146                   (phys_bytes) reboot_size) != OK) return EFAULT;
18147           if (monitor_code[sizeof(monitor_code)-1] != 0) return EINVAL;
18148           break;
18149     default:
18150           return EINVAL;
18151     }
18152
18153     /* Kill all processes except init. */
18154     check_sig(-1, SIGKILL);
```

```
18155
18156     tell_fs(EXIT, INIT_PROC_NR, 0, 0);    /* cleanup init */
18157
18158     tell_fs(SYNC,0,0,0);
18159
18160     sys_abort(reboot_flag, monitor_code);
18161     /* NOTREACHED */
18162   }


18165   /*===========================================================================*
18166    *                                sig_proc                                   *
18167    *===========================================================================*/
18168   PUBLIC void sig_proc(rmp, signo)
18169   register struct mproc *rmp;     /* pointer to the process to be signaled */
18170   int signo;                      /* signal to send to process (1 to _NSIG) */
18171   {
18172   /* Send a signal to a process.  Check to see if the signal is to be caught,
18173    * ignored, or blocked.  If the signal is to be caught, coordinate with
18174    * KERNEL to push a sigcontext structure and a sigframe structure onto
18175    * the catcher's stack.  Also, KERNEL will reset the program counter and
18176    * stack pointer, so that when the process next runs, it will be executing
18177    * the signal handler.  When the signal handler returns,  sigreturn(2)
18178    * will be called.  Then KERNEL will restore the signal context from the
18179    * sigcontext structure.
18180    *
18181    * If there is insufficient stack space, kill the process.
18182    */
18183
18184     vir_bytes new_sp;
18185     int slot;
18186     int sigflags;
18187     struct sigmsg sm;
18188
18189     slot = (int) (rmp - mproc);
18190     if (!(rmp->mp_flags & IN_USE)) {
18191           printf("MM: signal %d sent to dead process %d\n", signo, slot);
18192           panic("", NO_NUM);
18193     }
18194     if (rmp->mp_flags & HANGING) {
18195           printf("MM: signal %d sent to HANGING process %d\n", signo, slot);
18196           panic("", NO_NUM);
18197     }
18198     if (rmp->mp_flags & TRACED && signo != SIGKILL) {
18199           /* A traced process has special handling. */
18200           unpause(slot);
18201           stop_proc(rmp, signo);  /* a signal causes it to stop */
18202           return;
18203     }
18204     /* Some signals are ignored by default. */
18205     if (sigismember(&rmp->mp_ignore, signo)) return;
18206
18207     if (sigismember(&rmp->mp_sigmask, signo)) {
18208           /* Signal should be blocked. */
18209           sigaddset(&rmp->mp_sigpending, signo);
18210           return;
18211     }
18212     sigflags = rmp->mp_sigact[signo].sa_flags;
18213     if (sigismember(&rmp->mp_catch, signo)) {
18214           if (rmp->mp_flags & SIGSUSPENDED)
```



```
18215                 sm.sm_mask = rmp->mp_sigmask2;
18216           else
18217                 sm.sm_mask = rmp->mp_sigmask;
18218           sm.sm_signo = signo;
18219           sm.sm_sighandler = (vir_bytes) rmp->mp_sigact[signo].sa_handler;
18220           sm.sm_sigreturn = rmp->mp_sigreturn;
18221           sys_getsp(slot, &new_sp);
18222           sm.sm_stkptr = new_sp;
18223
18224           /* Make room for the sigcontext and sigframe struct. */
18225           new_sp -= sizeof(struct sigcontext)
18226                                   + 3 * sizeof(char *) + 2 * sizeof(int);
18227
18228           if (adjust(rmp, rmp->mp_seg[D].mem_len, new_sp) != OK)
18229                 goto doterminate;
18230
18231           rmp->mp_sigmask |= rmp->mp_sigact[signo].sa_mask;
18232           if (sigflags & SA_NODEFER)
18233                 sigdelset(&rmp->mp_sigmask, signo);
18234           else
18235                 sigaddset(&rmp->mp_sigmask, signo);
18236
18237           if (sigflags & SA_RESETHAND) {
18238                 sigdelset(&rmp->mp_catch, signo);
18239                 rmp->mp_sigact[signo].sa_handler = SIG_DFL;
18240           }
18241
18242           sys_sendsig(slot, &sm);
18243           sigdelset(&rmp->mp_sigpending, signo);
18244           /* If process is hanging on PAUSE, WAIT, SIGSUSPEND, tty, pipe, etc.,
18245            * release it.
18246            */
18247           unpause(slot);
18248           return;
18249     }
18250   doterminate:
18251     /* Signal should not or cannot be caught.  Terminate the process. */
18252     rmp->mp_sigstatus = (char) signo;
18253     if (sigismember(&core_sset, signo)) {
18254           /* Switch to the user's FS environment and dump core. */
18255           tell_fs(CHDIR, slot, FALSE, 0);
18256           dump_core(rmp);
18257     }
18258     mm_exit(rmp, 0);              /* terminate process */
18259   }


18262   /*===========================================================================*
18263    *                                check_sig                                  *
18264    *===========================================================================*/
18265   PUBLIC int check_sig(proc_id, signo)
18266   pid_t proc_id;                  /* pid of proc to sig, or 0 or -1, or -pgrp */
18267   int signo;                      /* signal to send to process (0 to _NSIG) */
18268   {
18269   /* Check to see if it is possible to send a signal.  The signal may have to be
18270    * sent to a group of processes.  This routine is invoked by the KILL system
18271    * call, and also when the kernel catches a DEL or other signal.
18272    */
18273
18274     register struct mproc *rmp;
```

```
18275     int count;                    /* count # of signals sent */
18276     int error_code;
18277
18278     if (signo < 0 || signo > _NSIG) return(EINVAL);
18279
18280     /* Return EINVAL for attempts to send SIGKILL to INIT alone. */
18281     if (proc_id == INIT_PID && signo == SIGKILL) return(EINVAL);
18282
18283     /* Search the proc table for processes to signal.  (See forkexit.c about
18284      * pid magic.)
18285      */
18286     count = 0;
18287     error_code = ESRCH;
18288     for (rmp = &mproc[INIT_PROC_NR]; rmp < &mproc[NR_PROCS]; rmp++) {
18289           if ( (rmp->mp_flags & IN_USE) == 0) continue;
18290           if (rmp->mp_flags & HANGING && signo != 0) continue;
18291
18292           /* Check for selection. */
18293           if (proc_id > 0 && proc_id != rmp->mp_pid) continue;
18294           if (proc_id == 0 && mp->mp_procgrp != rmp->mp_procgrp) continue;
18295           if (proc_id == -1 && rmp->mp_pid == INIT_PID) continue;
18296           if (proc_id < -1 && rmp->mp_procgrp != -proc_id) continue;
18297
18298           /* Check for permission. */
18299           if (mp->mp_effuid != SUPER_USER
18300               && mp->mp_realuid != rmp->mp_realuid
18301               && mp->mp_effuid != rmp->mp_realuid
18302               && mp->mp_realuid != rmp->mp_effuid
18303               && mp->mp_effuid != rmp->mp_effuid) {
18304                   error_code = EPERM;
18305                   continue;
18306           }
18307
18308           count++;
18309           if (signo == 0) continue;
18310
18311           /* 'sig_proc' will handle the disposition of the signal.  The
18312            * signal may be caught, blocked, ignored, or cause process
18313            * termination, possibly with core dump.
18314            */
18315           sig_proc(rmp, signo);
18316
18317           if (proc_id > 0) break; /* only one process being signaled */
18318     }
18319
18320     /* If the calling process has killed itself, don't reply. */
18321     if ((mp->mp_flags & IN_USE) == 0 || (mp->mp_flags & HANGING))
18322           dont_reply = TRUE;
18323     return(count > 0 ? OK : error_code);
18324   }


18327   /*===========================================================================*
18328    *                                check_pending                              *
18329    *===========================================================================*/
18330   PRIVATE void check_pending()
18331   {
18332     /* Check to see if any pending signals have been unblocked.  The
18333      * first such signal found is delivered.
18334      *
```



```
18335      * If multiple pending unmasked signals are found, they will be
18336      * delivered sequentially.
18337      *
18338      * There are several places in this file where the signal mask is
18339      * changed.  At each such place, check_pending() should be called to
18340      * check for newly unblocked signals.
18341      */
18342
18343     int i;
18344
18345     for (i = 1; i < _NSIG; i++) {
18346           if (sigismember(&mp->mp_sigpending, i) &&
18347                   !sigismember(&mp->mp_sigmask, i)) {
18348                   sigdelset(&mp->mp_sigpending, i);
18349                   sig_proc(mp, i);
18350                   break;
18351           }
18352     }
18353   }


18356   /*===========================================================================*
18357    *                                unpause                                    *
18358    *===========================================================================*/
18359   PRIVATE void unpause(pro)
18360   int pro;                      /* which process number */
18361   {
18362   /* A signal is to be sent to a process.  If that process is hanging on a
18363    * system call, the system call must be terminated with EINTR.  Possible
18364    * calls are PAUSE, WAIT, READ and WRITE, the latter two for pipes and ttys.
18365    * First check if the process is hanging on an MM call.  If not, tell FS,
18366    * so it can check for READs and WRITEs from pipes, ttys and the like.
18367    */
18368
18369     register struct mproc *rmp;
18370
18371     rmp = &mproc[pro];
18372
18373     /* Check to see if process is hanging on a PAUSE call. */
18374     if ( (rmp->mp_flags & PAUSED) && (rmp->mp_flags & HANGING) == 0) {
18375           rmp->mp_flags &= ~PAUSED;
18376           reply(pro, EINTR, 0, NIL_PTR);
18377           return;
18378     }
18379
18380     /* Check to see if process is hanging on a WAIT call. */
18381     if ( (rmp->mp_flags & WAITING) && (rmp->mp_flags & HANGING) == 0) {
18382           rmp->mp_flags &= ~WAITING;
18383           reply(pro, EINTR, 0, NIL_PTR);
18384           return;
18385     }
18386
18387     /* Check to see if process is hanging on a SIGSUSPEND call. */
18388     if ((rmp->mp_flags & SIGSUSPENDED) && (rmp->mp_flags & HANGING) == 0) {
18389           rmp->mp_flags &= ~SIGSUSPENDED;
18390           reply(pro, EINTR, 0, NIL_PTR);
18391           return;
18392     }
18393
18394     /* Process is not hanging on an MM call.  Ask FS to take a look. */
```

```
18395           tell_fs(UNPAUSE, pro, 0, 0);
18396   }


18399   /*===========================================================================*
18400    *                              dump_core                                    *
18401    *===========================================================================*/
18402   PRIVATE void dump_core(rmp)
18403   register struct mproc *rmp;     /* whose core is to be dumped */
18404   {
18405   /* Make a core dump on the file "core", if possible. */
18406
18407     int fd, fake_fd, nr_written, seg, slot;
18408     char *buf;
18409     vir_bytes current_sp;
18410     phys_bytes left;              /* careful; 64K might overflow vir_bytes */
18411     unsigned nr_to_write;         /* unsigned for arg to write() but < INT_MAX */
18412     long trace_data, trace_off;
18413
18414     slot = (int) (rmp - mproc);
18415
18416     /* Can core file be written?  We are operating in the user's FS environment,
18417      * so no special permission checks are needed.
18418      */
18419     if (rmp->mp_realuid != rmp->mp_effuid) return;
18420     if ( (fd = creat(core_name, CORE_MODE)) < 0) return;
18421     rmp->mp_sigstatus |= DUMPED;
18422
18423     /* Make sure the stack segment is up to date.
18424      * We don't want adjust() to fail unless current_sp is preposterous,
18425      * but it might fail due to safety checking.  Also, we don't really want
18426      * the adjust() for sending a signal to fail due to safety checking.
18427      * Maybe make SAFETY_BYTES a parameter.
18428      */
18429     sys_getsp(slot, &current_sp);
18430     adjust(rmp, rmp->mp_seg[D].mem_len, current_sp);
18431
18432     /* Write the memory map of all segments to begin the core file. */
18433     if (write(fd, (char *) rmp->mp_seg, (unsigned) sizeof rmp->mp_seg)
18434         != (unsigned) sizeof rmp->mp_seg) {
18435           close(fd);
18436           return;
18437     }
18438
18439     /* Write out the whole kernel process table entry to get the regs. */
18440     trace_off = 0;
18441     while (sys_trace(3, slot, trace_off, &trace_data) == OK) {
18442           if (write(fd, (char *) &trace_data, (unsigned) sizeof (long))
18443               != (unsigned) sizeof (long)) {
18444                   close(fd);
18445                   return;
18446           }
18447           trace_off += sizeof (long);
18448     }
18449
18450     /* Loop through segments and write the segments themselves out. */
18451     for (seg = 0; seg < NR_SEGS; seg++) {
18452           buf = (char *) ((vir_bytes) rmp->mp_seg[seg].mem_vir << CLICK_SHIFT);
18453           left = (phys_bytes) rmp->mp_seg[seg].mem_len << CLICK_SHIFT;
18454           fake_fd = (slot << 8) | (seg << 6) | fd;
```



```
18455
18456           /* Loop through a segment, dumping it. */
18457           while (left != 0) {
18458                   nr_to_write = (unsigned) MIN(left, DUMP_SIZE);
18459                   if ( (nr_written = write(fake_fd, buf, nr_to_write)) < 0) {
18460                           close(fd);
18461                           return;
18462                   }
18463                   buf += nr_written;
18464                   left -= nr_written;
18465           }
18466      }
18467      close(fd);
18468   }


++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/mm/getset.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

18500   /* This file handles the 4 system calls that get and set uids and gids.
18501    * It also handles getpid(), setsid(), and getpgrp().  The code for each
18502    * one is so tiny that it hardly seemed worthwhile to make each a separate
18503    * function.
18504    */
18505
18506   #include "mm.h"
18507   #include <minix/callnr.h>
18508   #include <signal.h>
18509   #include "mproc.h"
18510   #include "param.h"
18511
18512   /*===========================================================================*
18513    *                              do_getset                                    *
18514    *===========================================================================*/
18515   PUBLIC int do_getset()
18516   {
18517   /* Handle GETUID, GETGID, GETPID, GETPGRP, SETUID, SETGID, SETSID.  The four
18518    * GETs and SETSID return their primary results in 'r'.  GETUID, GETGID, and
18519    * GETPID also return secondary results (the effective IDs, or the parent
18520    * process ID) in 'result2', which is returned to the user.
18521    */
18522
18523     register struct mproc *rmp = mp;
18524     register int r;
18525
18526     switch(mm_call) {
18527           case GETUID:
18528                   r = rmp->mp_realuid;
18529                   result2 = rmp->mp_effuid;
18530                   break;
18531
18532           case GETGID:
18533                   r = rmp->mp_realgid;
18534                   result2 = rmp->mp_effgid;
18535                   break;
18536
18537           case GETPID:
18538                   r = mproc[who].mp_pid;
18539                   result2 = mproc[rmp->mp_parent].mp_pid;
```

```
18540                   break;
18541
18542           case SETUID:
18543                   if (rmp->mp_realuid != usr_id && rmp->mp_effuid != SUPER_USER)
18544                           return(EPERM);
18545                   rmp->mp_realuid = usr_id;
18546                   rmp->mp_effuid = usr_id;
18547                   tell_fs(SETUID, who, usr_id, usr_id);
18548                   r = OK;
18549                   break;
18550
18551           case SETGID:
18552                   if (rmp->mp_realgid != grpid && rmp->mp_effuid != SUPER_USER)
18553                           return(EPERM);
18554                   rmp->mp_realgid = grpid;
18555                   rmp->mp_effgid = grpid;
18556                   tell_fs(SETGID, who, grpid, grpid);
18557                   r = OK;
18558                   break;
18559
18560           case SETSID:
18561                   if (rmp->mp_procgrp == rmp->mp_pid) return(EPERM);
18562                   rmp->mp_procgrp = rmp->mp_pid;
18563                   tell_fs(SETSID, who, 0, 0);
18564                   /*FALL THROUGH*/
18565
18566           case GETPGRP:
18567                   r = rmp->mp_procgrp;
18568                   break;
18569
18570           default:
18571                   r = EINVAL;
18572                   break;
18573       }
18574       return(r);
18575   }
```

++

src/mm/trace.c

++

```
18600   /* This file handles the memory manager's part of debugging, using the
18601    * ptrace system call. Most of the commands are passed on to the system
18602    * task for completion.
18603    *
18604    * The debugging commands available are:
18605    * T_STOP       stop the process
18606    * T_OK         enable tracing by parent for this process
18607    * T_GETINS     return value from instruction space
18608    * T_GETDATA    return value from data space
18609    * T_GETUSER    return value from user process table
18610    * T_SETINS     set value in instruction space
18611    * T_SETDATA    set value in data space
18612    * T_SETUSER    set value in user process table
18613    * T_RESUME     resume execution
18614    * T_EXIT       exit
```



```
18615    * T_STEP       set trace bit
18616    *
18617    * The T_OK and T_EXIT commands are handled here, and the T_RESUME and
18618    * T_STEP commands are partially handled here and completed by the system
18619    * task. The rest are handled entirely by the system task.
18620    */
18621
18622   #include "mm.h"
18623   #include <sys/ptrace.h>
18624   #include <signal.h>
18625   #include "mproc.h"
18626   #include "param.h"
18627
18628   #define NIL_MPROC       ((struct mproc *) 0)
18629
18630   FORWARD _PROTOTYPE( struct mproc *findproc, (pid_t lpid) );
18631
18632   /*===========================================================================*
18633    *                              do_trace                                     *
18634    *===========================================================================*/
18635   PUBLIC int do_trace()
18636   {
18637     register struct mproc *child;
18638
18639     /* the T_OK call is made by the child fork of the debugger before it execs
18640      * the process to be traced
18641      */
18642     if (request == T_OK) {/* enable tracing by parent for this process */
18643           mp->mp_flags |= TRACED;
18644           mm_out.m2_l2 = 0;
18645           return(OK);
18646     }
18647     if ((child = findproc(pid)) == NIL_MPROC || !(child->mp_flags & STOPPED)) {
18648           return(ESRCH);
18649     }
18650     /* all the other calls are made by the parent fork of the debugger to
18651      * control execution of the child
18652      */
18653     switch (request) {
18654     case T_EXIT:          /* exit */
18655           mm_exit(child, (int)data);
18656           mm_out.m2_l2 = 0;
18657           return(OK);
18658     case T_RESUME:
18659     case T_STEP:          /* resume execution */
18660           if (data < 0 || data > _NSIG) return(EIO);
18661           if (data > 0) {         /* issue signal */
18662                   child->mp_flags &= ~TRACED;  /* so signal is not diverted */
18663                   sig_proc(child, (int) data);
18664                   child->mp_flags |= TRACED;
18665           }
18666           child->mp_flags &= ~STOPPED;
18667           break;
18668     }
18669     if (sys_trace(request, (int) (child - mproc), taddr, &data) != OK)
18670           return(-errno);
18671     mm_out.m2_l2 = data;
18672     return(OK);
18673   }
```

```
18675   /*===========================================================================*
18676    *                              findproc                                     *
18677    *===========================================================================*/
18678   PRIVATE struct mproc *findproc(lpid)
18679   pid_t lpid;
18680   {
18681     register struct mproc *rmp;
18682
18683     for (rmp = &mproc[INIT_PROC_NR + 1]; rmp < &mproc[NR_PROCS]; rmp++)
18684           if (rmp->mp_flags & IN_USE && rmp->mp_pid == lpid) return(rmp);
18685     return(NIL_MPROC);
18686   }

18688   /*===========================================================================*
18689    *                              stop_proc                                    *
18690    *===========================================================================*/
18691   PUBLIC void stop_proc(rmp, signo)
18692   register struct mproc *rmp;
18693   int signo;
18694   {
18695   /* A traced process got a signal so stop it. */
18696
18697     register struct mproc *rpmp = mproc + rmp->mp_parent;
18698
18699     if (sys_trace(-1, (int) (rmp - mproc), 0L, (long *) 0) != OK) return;
18700     rmp->mp_flags |= STOPPED;
18701     if (rpmp->mp_flags & WAITING) {
18702           rpmp->mp_flags &= ~WAITING;     /* parent is no longer waiting */
18703           reply(rmp->mp_parent, rmp->mp_pid, 0177 | (signo << 8), NIL_PTR);
18704     } else {
18705           rmp->mp_sigstatus = signo;
18706     }
18707     return;
18708   }


++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/mm/alloc.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

18800   /* This file is concerned with allocating and freeing arbitrary-size blocks of
18801    * physical memory on behalf of the FORK and EXEC system calls.  The key data
18802    * structure used is the hole table, which maintains a list of holes in memory.
18803    * It is kept sorted in order of increasing memory address. The addresses
18804    * it contains refer to physical memory, starting at absolute address 0
18805    * (i.e., they are not relative to the start of MM).  During system
18806    * initialization, that part of memory containing the interrupt vectors,
18807    * kernel, and MM are "allocated" to mark them as not available and to
18808    * remove them from the hole list.
18809    *
18810    * The entry points into this file are:
18811    *   alloc_mem: allocate a given sized chunk of memory
18812    *   free_mem:  release a previously allocated chunk of memory
18813    *   mem_init:  initialize the tables when MM start up
18814    *   max_hole:  returns the largest hole currently available
18815    */
18816
18817   #include "mm.h"
18818   #include <minix/com.h>
18819
```



```
18820   #define NR_HOLES         128    /* max # entries in hole table */
18821   #define NIL_HOLE (struct hole *) 0
18822
18823   PRIVATE struct hole {
18824     phys_clicks h_base;           /* where does the hole begin? */
18825     phys_clicks h_len;            /* how big is the hole? */
18826     struct hole *h_next;          /* pointer to next entry on the list */
18827   } hole[NR_HOLES];
18828
18829
18830   PRIVATE struct hole *hole_head; /* pointer to first hole */
18831   PRIVATE struct hole *free_slots;        /* ptr to list of unused table slots */
18832
18833   FORWARD _PROTOTYPE( void del_slot, (struct hole *prev_ptr, struct hole *hp) );
18834   FORWARD _PROTOTYPE( void merge, (struct hole *hp)                           );
18835
18836
18837   /*===========================================================================*
18838    *                              alloc_mem                                    *
18839    *===========================================================================*/
18840   PUBLIC phys_clicks alloc_mem(clicks)
18841   phys_clicks clicks;             /* amount of memory requested */
18842   {
18843   /* Allocate a block of memory from the free list using first fit. The block
18844    * consists of a sequence of contiguous bytes, whose length in clicks is
18845    * given by 'clicks'.  A pointer to the block is returned.  The block is
18846    * always on a click boundary.  This procedure is called when memory is
18847    * needed for FORK or EXEC.
18848    */
18849
18850     register struct hole *hp, *prev_ptr;
18851     phys_clicks old_base;
18852
18853     hp = hole_head;
18854     while (hp != NIL_HOLE) {
18855           if (hp->h_len >= clicks) {
18856                 /* We found a hole that is big enough.  Use it. */
18857                 old_base = hp->h_base;  /* remember where it started */
18858                 hp->h_base += clicks;   /* bite a piece off */
18859                 hp->h_len -= clicks;    /* ditto */
18860
18861                 /* If hole is only partly used, reduce size and return. */
18862                 if (hp->h_len != 0) return(old_base);
18863
18864                 /* The entire hole has been used up.  Manipulate free list. */
18865                 del_slot(prev_ptr, hp);
18866                 return(old_base);
18867           }
18868
18869           prev_ptr = hp;
18870           hp = hp->h_next;
18871     }
18872     return(NO_MEM);
18873   }


18876   /*===========================================================================*
18877    *                              free_mem                                     *
18878    *===========================================================================*/
18879   PUBLIC void free_mem(base, clicks)
```

```
18880   phys_clicks base;               /* base address of block to free */
18881   phys_clicks clicks;             /* number of clicks to free */
18882   {
18883   /* Return a block of free memory to the hole list.  The parameters tell where
18884    * the block starts in physical memory and how big it is.  The block is added
18885    * to the hole list.  If it is contiguous with an existing hole on either end,
18886    * it is merged with the hole or holes.
18887    */
18888
18889     register struct hole *hp, *new_ptr, *prev_ptr;
18890
18891     if (clicks == 0) return;
18892     if ( (new_ptr = free_slots) == NIL_HOLE) panic("Hole table full", NO_NUM);
18893     new_ptr->h_base = base;
18894     new_ptr->h_len = clicks;
18895     free_slots = new_ptr->h_next;
18896     hp = hole_head;
18897
18898     /* If this block's address is numerically less than the lowest hole currently
18899      * available, or if no holes are currently available, put this hole on the
18900      * front of the hole list.
18901      */
18902     if (hp == NIL_HOLE || base <= hp->h_base) {
18903           /* Block to be freed goes on front of the hole list. */
18904           new_ptr->h_next = hp;
18905           hole_head = new_ptr;
18906           merge(new_ptr);
18907           return;
18908     }
18909
18910     /* Block to be returned does not go on front of hole list. */
18911     while (hp != NIL_HOLE && base > hp->h_base) {
18912           prev_ptr = hp;
18913           hp = hp->h_next;
18914     }
18915
18916     /* We found where it goes.  Insert block after 'prev_ptr'. */
18917     new_ptr->h_next = prev_ptr->h_next;
18918     prev_ptr->h_next = new_ptr;
18919     merge(prev_ptr);              /* sequence is 'prev_ptr', 'new_ptr', 'hp' */
18920   }


18923   /*==========================================================================*
18924    *                              del_slot                                    *
18925    *==========================================================================*/
18926   PRIVATE void del_slot(prev_ptr, hp)
18927   register struct hole *prev_ptr; /* pointer to hole entry just ahead of 'hp' */
18928   register struct hole *hp;       /* pointer to hole entry to be removed */
18929   {
18930   /* Remove an entry from the hole list.  This procedure is called when a
18931    * request to allocate memory removes a hole in its entirety, thus reducing
18932    * the numbers of holes in memory, and requiring the elimination of one
18933    * entry in the hole list.
18934    */
18935
18936     if (hp == hole_head)
18937           hole_head = hp->h_next;
18938     else
18939           prev_ptr->h_next = hp->h_next;
```



```
18940
18941     hp->h_next = free_slots;
18942     free_slots = hp;
18943   }


18946   /*==========================================================================*
18947    *                              merge                                       *
18948    *==========================================================================*/
18949   PRIVATE void merge(hp)
18950   register struct hole *hp;       /* ptr to hole to merge with its successors */
18951   {
18952   /* Check for contiguous holes and merge any found.  Contiguous holes can occur
18953    * when a block of memory is freed, and it happens to abut another hole on
18954    * either or both ends.  The pointer 'hp' points to the first of a series of
18955    * three holes that can potentially all be merged together.
18956    */
18957
18958     register struct hole *next_ptr;
18959
18960     /* If 'hp' points to the last hole, no merging is possible.  If it does not,
18961      * try to absorb its successor into it and free the successor's table entry.
18962      */
18963     if ( (next_ptr = hp->h_next) == NIL_HOLE) return;
18964     if (hp->h_base + hp->h_len == next_ptr->h_base) {
18965           hp->h_len += next_ptr->h_len;   /* first one gets second one's mem */
18966           del_slot(hp, next_ptr);
18967     } else {
18968           hp = next_ptr;
18969     }
18970
18971     /* If 'hp' now points to the last hole, return; otherwise, try to absorb its
18972      * successor into it.
18973      */
18974     if ( (next_ptr = hp->h_next) == NIL_HOLE) return;
18975     if (hp->h_base + hp->h_len == next_ptr->h_base) {
18976           hp->h_len += next_ptr->h_len;
18977           del_slot(hp, next_ptr);
18978     }
18979   }


18982   /*==========================================================================*
18983    *                              max_hole                                    *
18984    *==========================================================================*/
18985   PUBLIC phys_clicks max_hole()
18986   {
18987   /* Scan the hole list and return the largest hole. */
18988
18989     register struct hole *hp;
18990     register phys_clicks max;
18991
18992     hp = hole_head;
18993     max = 0;
18994     while (hp != NIL_HOLE) {
18995           if (hp->h_len > max) max = hp->h_len;
18996           hp = hp->h_next;
18997     }
18998     return(max);
18999   }
```

```
19002   /*===========================================================================*
19003    *                              mem_init                                     *
19004    *===========================================================================*/
19005   PUBLIC void mem_init(total, free)
19006   phys_clicks *total, *free;              /* memory size summaries */
19007   {
19008   /* Initialize hole lists.  There are two lists: 'hole_head' points to a linked
19009    * list of all the holes (unused memory) in the system; 'free_slots' points to
19010    * a linked list of table entries that are not in use.  Initially, the former
19011    * list has one entry for each chunk of physical memory, and the second
19012    * list links together the remaining table slots.  As memory becomes more
19013    * fragmented in the course of time (i.e., the initial big holes break up into
19014    * smaller holes), new table slots are needed to represent them.  These slots
19015    * are taken from the list headed by 'free_slots'.
19016    */
19017
19018     register struct hole *hp;
19019     phys_clicks base;             /* base address of chunk */
19020     phys_clicks size;             /* size of chunk */
19021     message mess;
19022
19023     /* Put all holes on the free list. */
19024     for (hp = &hole[0]; hp < &hole[NR_HOLES]; hp++) hp->h_next = hp + 1;
19025     hole[NR_HOLES-1].h_next = NIL_HOLE;
19026     hole_head = NIL_HOLE;
19027     free_slots = &hole[0];
19028
19029     /* Ask the kernel for chunks of physical memory and allocate a hole for
19030      * each of them.  The SYS_MEM call responds with the base and size of the
19031      * next chunk and the total amount of memory.
19032      */
19033     *free = 0;
19034     for (;;) {
19035           mess.m_type = SYS_MEM;
19036           if (sendrec(SYSTASK, &mess) != OK) panic("bad SYS_MEM?", NO_NUM);
19037           base = mess.m1_i1;
19038           size = mess.m1_i2;
19039           if (size == 0) break;           /* no more? */
19040
19041           free_mem(base, size);
19042           *total = mess.m1_i3;
19043           *free += size;
19044     }
19045   }
```


++
src/mm/utility.c
++

```
19100   /* This file contains some utility routines for MM.
19101    *
19102    * The entry points are:
19103    *   allowed:   see if an access is permitted
19104    *   no_sys:    this routine is called for invalid system call numbers
19105    *   panic:     MM has run aground of a fatal error and cannot continue
19106    *   tell_fs:   interface to FS
19107    */
19108
19109   #include "mm.h"
19110   #include <sys/stat.h>
19111   #include <minix/callnr.h>
19112   #include <minix/com.h>
19113   #include <fcntl.h>
19114   #include <signal.h>             /* needed only because mproc.h needs it */
19115   #include "mproc.h"
19116
19117   /*===========================================================================*
19118    *                              allowed                                      *
19119    *===========================================================================*/
19120   PUBLIC int allowed(name_buf, s_buf, mask)
19121   char *name_buf;                 /* pointer to file name to be EXECed */
19122   struct stat *s_buf;             /* buffer for doing and returning stat struct*/
19123   int mask;                       /* R_BIT, W_BIT, or X_BIT */
19124   {
19125   /* Check to see if file can be accessed.  Return EACCES or ENOENT if the access
19126    * is prohibited.  If it is legal open the file and return a file descriptor.
19127    */
19128
19129     int fd;
19130     int save_errno;
19131
19132     /* Use the fact that mask for access() is the same as the permissions mask.
19133      * E.g., X_BIT in <minix/const.h> is the same as X_OK in <unistd.h> and
19134      * S_IXOTH in <sys/stat.h>.  tell_fs(DO_CHDIR, ...) has set MM's real ids
19135      * to the user's effective ids, so access() works right for setuid programs.
19136      */
19137     if (access(name_buf, mask) < 0) return(-errno);
19138
19139     /* The file is accessible but might not be readable.  Make it readable. */
19140     tell_fs(SETUID, MM_PROC_NR, (int) SUPER_USER, (int) SUPER_USER);
19141
19142     /* Open the file and fstat it.  Restore the ids early to handle errors. */
19143     fd = open(name_buf, O_RDONLY);
19144     save_errno = errno;             /* open might fail, e.g. from ENFILE */
19145     tell_fs(SETUID, MM_PROC_NR, (int) mp->mp_effuid, (int) mp->mp_effuid);
19146     if (fd < 0) return(-save_errno);
19147     if (fstat(fd, s_buf) < 0) panic("allowed: fstat failed", NO_NUM);
19148
19149     /* Only regular files can be executed. */
19150     if (mask == X_BIT && (s_buf->st_mode & I_TYPE) != I_REGULAR) {
19151           close(fd);
19152           return(EACCES);
19153     }
19154     return(fd);
```

```
19155   }


19158   /*===========================================================================*
19159    *                              no_sys                                       *
19160    *===========================================================================*/
19161   PUBLIC int no_sys()
19162   {
19163   /* A system call number not implemented by MM has been requested. */
19164
19165     return(EINVAL);
19166   }


19169   /*===========================================================================*
19170    *                              panic                                        *
19171    *===========================================================================*/
19172   PUBLIC void panic(format, num)
19173   char *format;                   /* format string */
19174   int num;                        /* number to go with format string */
19175   {
19176   /* Something awful has happened.  Panics are caused when an internal
19177    * inconsistency is detected, e.g., a programming error or illegal value of a
19178    * defined constant.
19179    */
19180
19181     printf("Memory manager panic: %s ", format);
19182     if (num != NO_NUM) printf("%d",num);
19183     printf("\n");
19184     tell_fs(SYNC, 0, 0, 0);       /* flush the cache to the disk */
19185     sys_abort(RBT_PANIC);
19186   }


19189   /*===========================================================================*
19190    *                              tell_fs                                      *
19191    *===========================================================================*/
19192   PUBLIC void tell_fs(what, p1, p2, p3)
19193   int what, p1, p2, p3;
19194   {
19195   /* This routine is only used by MM to inform FS of certain events:
19196    *      tell_fs(CHDIR, slot, dir, 0)
19197    *      tell_fs(EXEC, proc, 0, 0)
19198    *      tell_fs(EXIT, proc, 0, 0)
19199    *      tell_fs(FORK, parent, child, pid)
19200    *      tell_fs(SETGID, proc, realgid, effgid)
19201    *      tell_fs(SETSID, proc, 0, 0)
19202    *      tell_fs(SETUID, proc, realuid, effuid)
19203    *      tell_fs(SYNC, 0, 0, 0)
19204    *      tell_fs(UNPAUSE, proc, signr, 0)
19205    */
19206
19207     message m;
19208
19209     m.m1_i1 = p1;
19210     m.m1_i2 = p2;
19211     m.m1_i3 = p3;
19212     _taskcall(FS_PROC_NR, what, &m);
19213   }
```



```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/mm/putk.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

19300   /* MM must occasionally print some message.  It uses the standard library
19301    * routine printk().  (The name "printf" is really a macro defined as
19302    * "printk"). Printing is done by calling the TTY task directly, not going
19303    * through FS.
19304    */
19305
19306   #include "mm.h"
19307   #include <minix/com.h>
19308
19309   #define BUF_SIZE          100   /* print buffer size */
19310
19311   PRIVATE int buf_count;          /* # characters in the buffer */
19312   PRIVATE char print_buf[BUF_SIZE];       /* output is buffered here */
19313   PRIVATE message putch_msg;      /* used for message to TTY task */
19314
19315   _PROTOTYPE( FORWARD void flush, (void) );
19316
19317   /*===========================================================================*
19318    *                              putk                                         *
19319    *===========================================================================*/
19320   PUBLIC void putk(c)
19321   int c;
19322   {
19323   /* Accumulate another character.  If 0 or buffer full, print it. */
19324
19325     if (c == 0 || buf_count == BUF_SIZE) flush();
19326     if (c == '\n') putk('\r');
19327     if (c != 0) print_buf[buf_count++] = c;
19328   }


19331   /*===========================================================================*
19332    *                              flush                                        *
19333    *===========================================================================*/
19334   PRIVATE void flush()
19335   {
19336   /* Flush the print buffer by calling TTY task. */
19337
19338     if (buf_count == 0) return;
19339     putch_msg.m_type = DEV_WRITE;
19340     putch_msg.PROC_NR  = 0;
19341     putch_msg.TTY_LINE = 0;
19342     putch_msg.ADDRESS  = print_buf;
19343     putch_msg.COUNT = buf_count;
19344     sendrec(TTY, &putch_msg);
19345     buf_count = 0;
19346   }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/fs/fs.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

19400    /* This is the master header for fs.  It includes some other files
19401     * and defines the principal constants.
19402     */
19403    #define _POSIX_SOURCE      1    /* tell headers to include POSIX stuff */
19404    #define _MINIX             1    /* tell headers to include MINIX stuff */
19405    #define _SYSTEM            1    /* tell headers that this is the kernel */
19406
19407    /* The following are so basic, all the *.c files get them automatically. */
19408    #include <minix/config.h>       /* MUST be first */
19409    #include <ansi.h>               /* MUST be second */
19410    #include <sys/types.h>
19411    #include <minix/const.h>
19412    #include <minix/type.h>
19413
19414    #include <limits.h>
19415    #include <errno.h>
19416
19417    #include <minix/syslib.h>
19418
19419    #include "const.h"
19420    #include "type.h"
19421    #include "proto.h"
19422    #include "glo.h"



++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/fs/const.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

19500    /* Tables sizes */
19501    #define V1_NR_DZONES       7    /* # direct zone numbers in a V1 inode */
19502    #define V1_NR_TZONES       9    /* total # zone numbers in a V1 inode */
19503    #define V2_NR_DZONES       7    /* # direct zone numbers in a V2 inode */
19504    #define V2_NR_TZONES      10    /* total # zone numbers in a V2 inode */
19505
19506    #define NR_FILPS         128    /* # slots in filp table */
19507    #define NR_INODES         64    /* # slots in "in core" inode table */
19508    #define NR_SUPERS          8    /* # slots in super block table */
19509    #define NR_LOCKS           8    /* # slots in the file locking table */
19510
19511    /* The type of sizeof may be (unsigned) long.  Use the following macro for
19512     * taking the sizes of small objects so that there are no surprises like
19513     * (small) long constants being passed to routines expecting an int.
19514     */
19515    #define usizeof(t) ((unsigned) sizeof(t))
19516
19517    /* File system types. */
19518    #define SUPER_MAGIC   0x137F    /* magic number contained in super-block */
19519    #define SUPER_REV     0x7F13    /* magic # when 68000 disk read on PC or vv */
19520    #define SUPER_V2      0x2468    /* magic # for V2 file systems */
19521    #define SUPER_V2_REV  0x6824    /* V2 magic written on PC, read on 68K or vv */
19522
19523    #define V1                 1    /* version number of V1 file systems */
19524    #define V2                 2    /* version number of V2 file systems */
```



```
19525
19526    /* Miscellaneous constants */
19527    #define SU_UID   ((uid_t) 0)    /* super_user's uid_t */
19528    #define SYS_UID  ((uid_t) 0)    /* uid_t for processes MM and INIT */
19529    #define SYS_GID  ((gid_t) 0)    /* gid_t for processes MM and INIT */
19530    #define NORMAL             0    /* forces get_block to do disk read */
19531    #define NO_READ            1    /* prevents get_block from doing disk read */
19532    #define PREFETCH           2    /* tells get_block not to read or mark dev */
19533
19534    #define XPIPE  (-NR_TASKS-1)    /* used in fp_task when susp'd on pipe */
19535    #define XOPEN  (-NR_TASKS-2)    /* used in fp_task when susp'd on open */
19536    #define XLOCK  (-NR_TASKS-3)    /* used in fp_task when susp'd on lock */
19537    #define XPOPEN (-NR_TASKS-4)    /* used in fp_task when susp'd on pipe open */
19538
19539    #define NO_BIT   ((bit_t) 0)    /* returned by alloc_bit() to signal failure */
19540
19541    #define DUP_MASK        0100    /* mask to distinguish dup2 from dup */
19542
19543    #define LOOK_UP            0    /* tells search_dir to lookup string */
19544    #define ENTER              1    /* tells search_dir to make dir entry */
19545    #define DELETE             2    /* tells search_dir to delete entry */
19546    #define IS_EMPTY           3    /* tells search_dir to ret. OK or ENOTEMPTY */
19547
19548    #define CLEAN              0    /* disk and memory copies identical */
19549    #define DIRTY              1    /* disk and memory copies differ */
19550    #define ATIME            002    /* set if atime field needs updating */
19551    #define CTIME            004    /* set if ctime field needs updating */
19552    #define MTIME            010    /* set if mtime field needs updating */
19553
19554    #define BYTE_SWAP          0    /* tells conv2/conv4 to swap bytes */
19555    #define DONT_SWAP          1    /* tells conv2/conv4 not to swap bytes */
19556
19557    #define END_OF_FILE   (-104)    /* eof detected */
19558
19559    #define ROOT_INODE         1    /* inode number for root directory */
19560    #define BOOT_BLOCK  ((block_t) 0)       /* block number of boot block */
19561    #define SUPER_BLOCK ((block_t) 1)       /* block number of super block */
19562
19563    #define DIR_ENTRY_SIZE       usizeof (struct direct)  /* # bytes/dir entry   */
19564    #define NR_DIR_ENTRIES   (BLOCK_SIZE/DIR_ENTRY_SIZE)  /* # dir entries/blk   */
19565    #define SUPER_SIZE      usizeof (struct super_block)  /* super_block size    */
19566    #define PIPE_SIZE          (V1_NR_DZONES*BLOCK_SIZE)  /* pipe size in bytes  */
19567    #define BITMAP_CHUNKS (BLOCK_SIZE/usizeof (bitchunk_t))/* # map chunks/blk   */
19568
19569    /* Derived sizes pertaining to the V1 file system. */
19570    #define V1_ZONE_NUM_SIZE           usizeof (zone1_t)  /* # bytes in V1 zone  */
19571    #define V1_INODE_SIZE             usizeof (d1_inode)  /* bytes in V1 dsk ino */
19572    #define V1_INDIRECTS   (BLOCK_SIZE/V1_ZONE_NUM_SIZE)  /* # zones/indir block */
19573    #define V1_INODES_PER_BLOCK (BLOCK_SIZE/V1_INODE_SIZE)/* # V1 dsk inodes/blk */
19574
19575    /* Derived sizes pertaining to the V2 file system. */
19576    #define V2_ZONE_NUM_SIZE            usizeof (zone_t)  /* # bytes in V2 zone  */
19577    #define V2_INODE_SIZE             usizeof (d2_inode)  /* bytes in V2 dsk ino */
19578    #define V2_INDIRECTS   (BLOCK_SIZE/V2_ZONE_NUM_SIZE)  /* # zones/indir block */
19579    #define V2_INODES_PER_BLOCK (BLOCK_SIZE/V2_INODE_SIZE)/* # V2 dsk inodes/blk */
19580
19581    #define printf printk
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/fs/type.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

19600    /* Declaration of the V1 inode as it is on the disk (not in core). */
19601    typedef struct {                /* V1.x disk inode */
19602      mode_t d1_mode;               /* file type, protection, etc. */
19603      uid_t d1_uid;                 /* user id of the file's owner */
19604      off_t d1_size;                /* current file size in bytes */
19605      time_t d1_mtime;              /* when was file data last changed */
19606      gid_t d1_gid;                 /* group number */
19607      nlink_t d1_nlinks;            /* how many links to this file */
19608      u16_t d1_zone[V1_NR_TZONES];  /* block nums for direct, ind, and dbl ind */
19609    } d1_inode;
19610
19611    /* Declaration of the V2 inode as it is on the disk (not in core). */
19612    typedef struct {                /* V2.x disk inode */
19613      mode_t d2_mode;               /* file type, protection, etc. */
19614      u16_t d2_nlinks;              /* how many links to this file. HACK! */
19615      uid_t d2_uid;                 /* user id of the file's owner. */
19616      u16_t d2_gid;                 /* group number HACK! */
19617      off_t d2_size;                /* current file size in bytes */
19618      time_t d2_atime;              /* when was file data last accessed */
19619      time_t d2_mtime;              /* when was file data last changed */
19620      time_t d2_ctime;              /* when was inode data last changed */
19621      zone_t d2_zone[V2_NR_TZONES]; /* block nums for direct, ind, and dbl ind */
19622    } d2_inode;



++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/fs/proto.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

19700    /* Function prototypes. */
19701
19702    /* Structs used in prototypes must be declared as such first. */
19703    struct buf;
19704    struct filp;
19705    struct inode;
19706    struct super_block;
19707
19708    /* cache.c */
19709    _PROTOTYPE( zone_t alloc_zone, (Dev_t dev, zone_t z)                    );
19710    _PROTOTYPE( void flushall, (Dev_t dev)                                  );
19711    _PROTOTYPE( void free_zone, (Dev_t dev, zone_t numb)                    );
19712    _PROTOTYPE( struct buf *get_block, (Dev_t dev, block_t block,int only_search));
19713    _PROTOTYPE( void invalidate, (Dev_t device)                             );
19714    _PROTOTYPE( void put_block, (struct buf *bp, int block_type)            );
19715    _PROTOTYPE( void rw_block, (struct buf *bp, int rw_flag)                );
19716    _PROTOTYPE( void rw_scattered, (Dev_t dev,
19717                              struct buf **bufq, int bufqsize, int rw_flag) );
19718
19719    /* device.c */
19720    _PROTOTYPE( void call_task, (int task_nr, message *mess_ptr)            );
19721    _PROTOTYPE( void dev_opcl, (int task_nr, message *mess_ptr)             );
19722    _PROTOTYPE( int dev_io, (int rw_flag, int nonblock, Dev_t dev,
19723                              off_t pos, int bytes, int proc, char *buff)   );
19724    _PROTOTYPE( int do_ioctl, (void)                                        );
```



```
19725    _PROTOTYPE( void no_dev, (int task_nr, message *m_ptr)                  );
19726    _PROTOTYPE( void call_ctty, (int task_nr, message *mess_ptr)            );
19727    _PROTOTYPE( void tty_open, (int task_nr, message *mess_ptr)             );
19728    _PROTOTYPE( void ctty_close, (int task_nr, message *mess_ptr)           );
19729    _PROTOTYPE( void ctty_open, (int task_nr, message *mess_ptr)            );
19730    _PROTOTYPE( int do_setsid, (void)                                       );
19731    #if ENABLE_NETWORKING
19732    _PROTOTYPE( void net_open, (int task_nr, message *mess_ptr)             );
19733    #else
19734    #define net_open  0
19735    #endif
19736
19737    /* filedes.c */
19738    _PROTOTYPE( struct filp *find_filp, (struct inode *rip, Mode_t bits)    );
19739    _PROTOTYPE( int get_fd, (int start, Mode_t bits, int *k, struct filp **fpt) );
19740    _PROTOTYPE( struct filp *get_filp, (int fild)                           );
19741
19742    /* inode.c */
19743    _PROTOTYPE( struct inode *alloc_inode, (Dev_t dev, Mode_t bits)         );
19744    _PROTOTYPE( void dup_inode, (struct inode *ip)                          );
19745    _PROTOTYPE( void free_inode, (Dev_t dev, Ino_t numb)                    );
19746    _PROTOTYPE( struct inode *get_inode, (Dev_t dev, int numb)              );
19747    _PROTOTYPE( void put_inode, (struct inode *rip)                         );
19748    _PROTOTYPE( void update_times, (struct inode *rip)                      );
19749    _PROTOTYPE( void rw_inode, (struct inode *rip, int rw_flag)             );
19750    _PROTOTYPE( void wipe_inode, (struct inode *rip)                        );
19751
19752    /* link.c */
19753    _PROTOTYPE( int do_link, (void)                                         );
19754    _PROTOTYPE( int do_unlink, (void)                                       );
19755    _PROTOTYPE( int do_rename, (void)                                       );
19756    _PROTOTYPE( void truncate, (struct inode *rip)                          );
19757
19758    /* lock.c */
19759    _PROTOTYPE( int lock_op, (struct filp *f, int req)                      );
19760    _PROTOTYPE( void lock_revive, (void)                                    );
19761
19762    /* main.c */
19763    _PROTOTYPE( void main, (void)                                           );
19764    _PROTOTYPE( void reply, (int whom, int result)                          );
19765
19766    /* misc.c */
19767    _PROTOTYPE( int do_dup, (void)                                          );
19768    _PROTOTYPE( int do_exit, (void)                                         );
19769    _PROTOTYPE( int do_fcntl, (void)                                        );
19770    _PROTOTYPE( int do_fork, (void)                                         );
19771    _PROTOTYPE( int do_exec, (void)                                         );
19772    _PROTOTYPE( int do_revive, (void)                                       );
19773    _PROTOTYPE( int do_set, (void)                                          );
19774    _PROTOTYPE( int do_sync, (void)                                         );
19775
19776    /* mount.c */
19777    _PROTOTYPE( int do_mount, (void)                                        );
19778    _PROTOTYPE( int do_umount, (void)                                       );
19779
19780    /* open.c */
19781    _PROTOTYPE( int do_close, (void)                                        );
19782    _PROTOTYPE( int do_creat, (void)                                        );
19783    _PROTOTYPE( int do_lseek, (void)                                        );
19784    _PROTOTYPE( int do_mknod, (void)                                        );
```

```
19785   _PROTOTYPE( int do_mkdir, (void)                                        );
19786   _PROTOTYPE( int do_open, (void)                                         );
19787
19788   /* path.c */
19789   _PROTOTYPE( struct inode *advance,(struct inode *dirp, char string[NAME_MAX]));
19790   _PROTOTYPE( int search_dir, (struct inode *ldir_ptr,
19791                           char string [NAME_MAX], ino_t *numb, int flag)  );
19792   _PROTOTYPE( struct inode *eat_path, (char *path)                        );
19793   _PROTOTYPE( struct inode *last_dir, (char *path, char string [NAME_MAX]));
19794
19795   /* pipe.c */
19796   _PROTOTYPE( int do_pipe, (void)                                         );
19797   _PROTOTYPE( int do_unpause, (void)                                      );
19798   _PROTOTYPE( int pipe_check, (struct inode *rip, int rw_flag,
19799                           int oflags, int bytes, off_t position, int *canwrite));
19800   _PROTOTYPE( void release, (struct inode *ip, int call_nr, int count)    );
19801   _PROTOTYPE( void revive, (int proc_nr, int bytes)                       );
19802   _PROTOTYPE( void suspend, (int task)                                    );
19803
19804   /* protect.c */
19805   _PROTOTYPE( int do_access, (void)                                       );
19806   _PROTOTYPE( int do_chmod, (void)                                        );
19807   _PROTOTYPE( int do_chown, (void)                                        );
19808   _PROTOTYPE( int do_umask, (void)                                        );
19809   _PROTOTYPE( int forbidden, (struct inode *rip, Mode_t access_desired)   );
19810   _PROTOTYPE( int read_only, (struct inode *ip)                           );
19811
19812   /* putk.c */
19813   _PROTOTYPE( void putk, (int c)                                          );
19814
19815   /* read.c */
19816   _PROTOTYPE( int do_read, (void)                                         );
19817   _PROTOTYPE( struct buf *rahead, (struct inode *rip, block_t baseblock,
19818                           off_t position, unsigned bytes_ahead)           );
19819   _PROTOTYPE( void read_ahead, (void)                                     );
19820   _PROTOTYPE( block_t read_map, (struct inode *rip, off_t position)       );
19821   _PROTOTYPE( int read_write, (int rw_flag)                               );
19822   _PROTOTYPE( zone_t rd_indir, (struct buf *bp, int index)                );
19823
19824   /* stadir.c */
19825   _PROTOTYPE( int do_chdir, (void)                                        );
19826   _PROTOTYPE( int do_chroot, (void)                                       );
19827   _PROTOTYPE( int do_fstat, (void)                                        );
19828   _PROTOTYPE( int do_stat, (void)                                         );
19829
19830   /* super.c */
19831   _PROTOTYPE( bit_t alloc_bit, (struct super_block *sp, int map, bit_t origin));
19832   _PROTOTYPE( void free_bit, (struct super_block *sp, int map,
19833                                           bit_t bit_returned)             );
19834   _PROTOTYPE( struct super_block *get_super, (Dev_t dev)                  );
19835   _PROTOTYPE( int mounted, (struct inode *rip)                            );
19836   _PROTOTYPE( int read_super, (struct super_block *sp)                    );
19837
19838   /* time.c */
19839   _PROTOTYPE( int do_stime, (void)                                        );
19840   _PROTOTYPE( int do_time, (void)                                         );
19841   _PROTOTYPE( int do_tims, (void)                                         );
19842   _PROTOTYPE( int do_utime, (void)                                        );
19843
19844   /* utility.c */
```



```
19845   _PROTOTYPE( time_t clock_time, (void)                                   );
19846   _PROTOTYPE( unsigned conv2, (int norm, int w)                           );
19847   _PROTOTYPE( long conv4, (int norm, long x)                              );
19848   _PROTOTYPE( int fetch_name, (char *path, int len, int flag)             );
19849   _PROTOTYPE( int no_sys, (void)                                          );
19850   _PROTOTYPE( void panic, (char *format, int num)                         );
19851
19852   /* write.c */
19853   _PROTOTYPE( void clear_zone, (struct inode *rip, off_t pos, int flag)   );
19854   _PROTOTYPE( int do_write, (void)                                        );
19855   _PROTOTYPE( struct buf *new_block, (struct inode *rip, off_t position)  );
19856   _PROTOTYPE( void zero_block, (struct buf *bp)                           );
```

++
src/fs/glo.h
++

```
19900   /* EXTERN should be extern except for the table file */
19901   #ifdef _TABLE
19902   #undef EXTERN
19903   #define EXTERN
19904   #endif
19905
19906   /* File System global variables */
19907   EXTERN struct fproc *fp;        /* pointer to caller's fproc struct */
19908   EXTERN int super_user;          /* 1 if caller is super_user, else 0 */
19909   EXTERN int dont_reply;          /* normally 0; set to 1 to inhibit reply */
19910   EXTERN int susp_count;          /* number of procs suspended on pipe */
19911   EXTERN int nr_locks;            /* number of locks currently in place */
19912   EXTERN int reviving;            /* number of pipe processes to be revived */
19913   EXTERN off_t rdahedpos;         /* position to read ahead */
19914   EXTERN struct inode *rdahed_inode;      /* pointer to inode to read ahead */
19915
19916   /* The parameters of the call are kept here. */
19917   EXTERN message m;               /* the input message itself */
19918   EXTERN message m1;              /* the output message used for reply */
19919   EXTERN int who;                 /* caller's proc number */
19920   EXTERN int fs_call;             /* system call number */
19921   EXTERN char user_path[PATH_MAX];/* storage for user path name */
19922
19923   /* The following variables are used for returning results to the caller. */
19924   EXTERN int err_code;            /* temporary storage for error number */
19925   EXTERN int rdwt_err;            /* status of last disk i/o request */
19926
19927   /* Data which need initialization. */
19928   extern _PROTOTYPE (int (*call_vector[]), (void) ); /* sys call table */
19929   extern int max_major;  /* maximum major device (+ 1) */
19930   extern char dot1[2];   /* dot1 (&dot1[0]) and dot2 (&dot2[0]) have a special */
19931   extern char dot2[3];   /* meaning to search_dir: no access permission check. */
```

++
src/fs/fproc.h
++

```
20000    /* This is the per-process information.  A slot is reserved for each potential
20001     * process. Thus NR_PROCS must be the same as in the kernel. It is not possible
20002     * or even necessary to tell when a slot is free here.
20003     */
20004
20005
20006    EXTERN struct fproc {
20007      mode_t fp_umask;              /* mask set by umask system call */
20008      struct inode *fp_workdir;     /* pointer to working directory's inode */
20009      struct inode *fp_rootdir;     /* pointer to current root dir (see chroot) */
20010      struct filp *fp_filp[OPEN_MAX];/* the file descriptor table */
20011      uid_t fp_realuid;             /* real user id */
20012      uid_t fp_effuid;              /* effective user id */
20013      gid_t fp_realgid;             /* real group id */
20014      gid_t fp_effgid;              /* effective group id */
20015      dev_t fp_tty;                 /* major/minor of controlling tty */
20016      int fp_fd;                    /* place to save fd if rd/wr can't finish */
20017      char *fp_buffer;              /* place to save buffer if rd/wr can't finish*/
20018      int  fp_nbytes;               /* place to save bytes if rd/wr can't finish */
20019      int  fp_cum_io_partial;       /* partial byte count if rd/wr can't finish */
20020      char fp_suspended;            /* set to indicate process hanging */
20021      char fp_revived;              /* set to indicate process being revived */
20022      char fp_task;                 /* which task is proc suspended on */
20023      char fp_sesldr;               /* true if proc is a session leader */
20024      pid_t fp_pid;                 /* process id */
20025      long fp_cloexec;              /* bit map for POSIX Table 6-2 FD_CLOEXEC */
20026    } fproc[NR_PROCS];
20027
20028    /* Field values. */
20029    #define NOT_SUSPENDED      0    /* process is not suspended on pipe or task */
20030    #define SUSPENDED          1    /* process is suspended on pipe or task */
20031    #define NOT_REVIVING       0    /* process is not being revived */
20032    #define REVIVING           1    /* process is being revived from suspension */
```

++
src/fs/buf.h
++

```
20100    /* Buffer (block) cache.  To acquire a block, a routine calls get_block(),
20101     * telling which block it wants.  The block is then regarded as "in use"
20102     * and has its 'b_count' field incremented.  All the blocks that are not
20103     * in use are chained together in an LRU list, with 'front' pointing
20104     * to the least recently used block, and 'rear' to the most recently used
20105     * block.  A reverse chain, using the field b_prev is also maintained.
20106     * Usage for LRU is measured by the time the put_block() is done.  The second
20107     * parameter to put_block() can violate the LRU order and put a block on the
20108     * front of the list, if it will probably not be needed soon.  If a block
20109     * is modified, the modifying routine must set b_dirt to DIRTY, so the block
20110     * will eventually be rewritten to the disk.
20111     */
20112
20113    #include <sys/dir.h>                    /* need struct direct */
20114
```



```
20115    EXTERN struct buf {
20116      /* Data portion of the buffer. */
20117      union {
20118        char b__data[BLOCK_SIZE];               /* ordinary user data */
20119        struct direct b__dir[NR_DIR_ENTRIES];   /* directory block */
20120        zone1_t b__v1_ind[V1_INDIRECTS];        /* V1 indirect block */
20121        zone_t  b__v2_ind[V2_INDIRECTS];        /* V2 indirect block */
20122        d1_inode b__v1_ino[V1_INODES_PER_BLOCK]; /* V1 inode block */
20123        d2_inode b__v2_ino[V2_INODES_PER_BLOCK]; /* V2 inode block */
20124        bitchunk_t b__bitmap[BITMAP_CHUNKS];    /* bit map block */
20125      } b;
20126
20127      /* Header portion of the buffer. */
20128      struct buf *b_next;           /* used to link all free bufs in a chain */
20129      struct buf *b_prev;           /* used to link all free bufs the other way */
20130      struct buf *b_hash;           /* used to link bufs on hash chains */
20131      block_t b_blocknr;            /* block number of its (minor) device */
20132      dev_t b_dev;                  /* major | minor device where block resides */
20133      char b_dirt;                  /* CLEAN or DIRTY */
20134      char b_count;                 /* number of users of this buffer */
20135    } buf[NR_BUFS];
20136
20137    /* A block is free if b_dev == NO_DEV. */
20138
20139    #define NIL_BUF ((struct buf *) 0)      /* indicates absence of a buffer */
20140
20141    /* These defs make it possible to use to bp->b_data instead of bp->b.b__data */
20142    #define b_data   b.b__data
20143    #define b_dir    b.b__dir
20144    #define b_v1_ind b.b__v1_ind
20145    #define b_v2_ind b.b__v2_ind
20146    #define b_v1_ino b.b__v1_ino
20147    #define b_v2_ino b.b__v2_ino
20148    #define b_bitmap b.b__bitmap
20149
20150    EXTERN struct buf *buf_hash[NR_BUF_HASH];       /* the buffer hash table */
20151
20152    EXTERN struct buf *front;       /* points to least recently used free block */
20153    EXTERN struct buf *rear;        /* points to most recently used free block */
20154    EXTERN int bufs_in_use;         /* # bufs currently in use (not on free list)*/
20155
20156    /* When a block is released, the type of usage is passed to put_block(). */
20157    #define WRITE_IMMED        0100 /* block should be written to disk now */
20158    #define ONE_SHOT           0200 /* set if block not likely to be needed soon */
20159
20160    #define INODE_BLOCK       (0 + MAYBE_WRITE_IMMED)        /* inode block */
20161    #define DIRECTORY_BLOCK   (1 + MAYBE_WRITE_IMMED)        /* directory block */
20162    #define INDIRECT_BLOCK    (2 + MAYBE_WRITE_IMMED)        /* pointer block */
20163    #define MAP_BLOCK         (3 + MAYBE_WRITE_IMMED)        /* bit map */
20164    #define ZUPER_BLOCK       (4 + WRITE_IMMED + ONE_SHOT)   /* super block */
20165    #define FULL_DATA_BLOCK    5                             /* data, fully used */
20166    #define PARTIAL_DATA_BLOCK 6                             /* data, partly used*/
20167
20168    #define HASH_MASK (NR_BUF_HASH - 1)     /* mask for hashing block numbers */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/fs/dev.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

20200    /* Device table.  This table is indexed by major device number.  It provides
20201     * the link between major device numbers and the routines that process them.
20202     */
20203
20204    typedef _PROTOTYPE (void (*dmap_t), (int task, message *m_ptr) );
20205
20206    extern struct dmap {
20207      dmap_t dmap_open;
20208      dmap_t dmap_rw;
20209      dmap_t dmap_close;
20210      int dmap_task;
20211    } dmap[];
20212
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/fs/file.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

20300    /* This is the filp table.  It is an intermediary between file descriptors and
20301     * inodes.  A slot is free if filp_count == 0.
20302     */
20303
20304    EXTERN struct filp {
20305      mode_t filp_mode;             /* RW bits, telling how file is opened */
20306      int filp_flags;               /* flags from open and fcntl */
20307      int filp_count;               /* how many file descriptors share this slot?*/
20308      struct inode *filp_ino;       /* pointer to the inode */
20309      off_t filp_pos;               /* file position */
20310    } filp[NR_FILPS];
20311
20312    #define FILP_CLOSED     0       /* filp_mode: associated device closed */
20313
20314    #define NIL_FILP (struct filp *) 0      /* indicates absence of a filp slot */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/fs/lock.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

20400    /* This is the file locking table.  Like the filp table, it points to the
20401     * inode table, however, in this case to achieve advisory locking.
20402     */
20403    EXTERN struct file_lock {
20404      short lock_type;              /* F_RDLOCK or F_WRLOCK; 0 means unused slot */
20405      pid_t lock_pid;               /* pid of the process holding the lock */
20406      struct inode *lock_inode;     /* pointer to the inode locked */
20407      off_t lock_first;             /* offset of first byte locked */
20408      off_t lock_last;              /* offset of last byte locked */
20409    } file_lock[NR_LOCKS];
```



```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/fs/inode.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

20500    /* Inode table.  This table holds inodes that are currently in use.  In some
20501     * cases they have been opened by an open() or creat() system call, in other
20502     * cases the file system itself needs the inode for one reason or another,
20503     * such as to search a directory for a path name.
20504     * The first part of the struct holds fields that are present on the
20505     * disk; the second part holds fields not present on the disk.
20506     * The disk inode part is also declared in "type.h" as 'd1_inode' for V1
20507     * file systems and 'd2_inode' for V2 file systems.
20508     */
20509
20510    EXTERN struct inode {
20511      mode_t i_mode;                /* file type, protection, etc. */
20512      nlink_t i_nlinks;             /* how many links to this file */
20513      uid_t i_uid;                  /* user id of the file's owner */
20514      gid_t i_gid;                  /* group number */
20515      off_t i_size;                 /* current file size in bytes */
20516      time_t i_atime;               /* time of last access (V2 only) */
20517      time_t i_mtime;               /* when was file data last changed */
20518      time_t i_ctime;               /* when was inode itself changed (V2 only)*/
20519      zone_t i_zone[V2_NR_TZONES];  /* zone numbers for direct, ind, and dbl ind */
20520
20521      /* The following items are not present on the disk. */
20522      dev_t i_dev;                  /* which device is the inode on */
20523      ino_t i_num;                  /* inode number on its (minor) device */
20524      int i_count;                  /* # times inode used; 0 means slot is free */
20525      int i_ndzones;                /* # direct zones (Vx_NR_DZONES) */
20526      int i_nindirs;                /* # indirect zones per indirect block */
20527      struct super_block *i_sp;     /* pointer to super block for inode's device */
20528      char i_dirt;                  /* CLEAN or DIRTY */
20529      char i_pipe;                  /* set to I_PIPE if pipe */
20530      char i_mount;                 /* this bit is set if file mounted on */
20531      char i_seek;                  /* set on LSEEK, cleared on READ/WRITE */
20532      char i_update;                /* the ATIME, CTIME, and MTIME bits are here */
20533    } inode[NR_INODES];
20534
20535
20536    #define NIL_INODE (struct inode *) 0    /* indicates absence of inode slot */
20537
20538    /* Field values.  Note that CLEAN and DIRTY are defined in "const.h" */
20539    #define NO_PIPE             0    /* i_pipe is NO_PIPE if inode is not a pipe */
20540    #define I_PIPE              1    /* i_pipe is I_PIPE if inode is a pipe */
20541    #define NO_MOUNT            0    /* i_mount is NO_MOUNT if file not mounted on*/
20542    #define I_MOUNT             1    /* i_mount is I_MOUNT if file mounted on */
20543    #define NO_SEEK             0    /* i_seek = NO_SEEK if last op was not SEEK */
20544    #define ISEEK               1    /* i_seek = ISEEK if last op was SEEK */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/fs/param.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

20600    /* The following names are synonyms for the variables in the input message. */
20601    #define acc_time      m.m2_l1
20602    #define addr          m.m1_i3
20603    #define buffer        m.m1_p1
20604    #define child         m.m1_i2
20605    #define co_mode       m.m1_i1
20606    #define eff_grp_id    m.m1_i3
20607    #define eff_user_id   m.m1_i3
20608    #define erki          m.m1_p1
20609    #define fd            m.m1_i1
20610    #define fd2           m.m1_i2
20611    #define ioflags       m.m1_i3
20612    #define group         m.m1_i3
20613    #define real_grp_id   m.m1_i2
20614    #define ls_fd         m.m2_i1
20615    #define mk_mode       m.m1_i2
20616    #define mode          m.m3_i2
20617    #define c_mode        m.m1_i3
20618    #define c_name        m.m1_p1
20619    #define name          m.m3_p1
20620    #define name1         m.m1_p1
20621    #define name2         m.m1_p2
20622    #define name_length   m.m3_i1
20623    #define name1_length  m.m1_i1
20624    #define name2_length  m.m1_i2
20625    #define nbytes        m.m1_i2
20626    #define offset        m.m2_l1
20627    #define owner         m.m1_i2
20628    #define parent        m.m1_i1
20629    #define pathname      m.m3_ca1
20630    #define pid           m.m1_i3
20631    #define pro           m.m1_i1
20632    #define rd_only       m.m1_i3
20633    #define real_user_id  m.m1_i2
20634    #define request       m.m1_i2
20635    #define sig           m.m1_i2
20636    #define slot1         m.m1_i1
20637    #define tp            m.m2_l1
20638    #define utime_actime  m.m2_l1
20639    #define utime_modtime m.m2_l2
20640    #define utime_file    m.m2_p1
20641    #define utime_length  m.m2_i1
20642    #define whence        m.m2_i2
20643
20644    /* The following names are synonyms for the variables in the output message. */
20645    #define reply_type    m1.m_type
20646    #define reply_l1      m1.m2_l1
20647    #define reply_i1      m1.m1_i1
20648    #define reply_i2      m1.m1_i2
20649    #define reply_t1      m1.m4_l1
20650    #define reply_t2      m1.m4_l2
20651    #define reply_t3      m1.m4_l3
20652    #define reply_t4      m1.m4_l4
20653    #define reply_t5      m1.m4_l5
```



```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/fs/super.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

20700    /* Super block table.  The root file system and every mounted file system
20701     * has an entry here.  The entry holds information about the sizes of the bit
20702     * maps and inodes.  The s_ninodes field gives the number of inodes available
20703     * for files and directories, including the root directory.  Inode 0 is
20704     * on the disk, but not used.  Thus s_ninodes = 4 means that 5 bits will be
20705     * used in the bit map, bit 0, which is always 1 and not used, and bits 1-4
20706     * for files and directories.  The disk layout is:
20707     *
20708     *      Item        # blocks
20709     *    boot block      1
20710     *    super block     1
20711     *    inode map     s_imap_blocks
20712     *    zone map      s_zmap_blocks
20713     *    inodes        (s_ninodes + 'inodes per block' - 1)/'inodes per block'
20714     *    unused        whatever is needed to fill out the current zone
20715     *    data zones    (s_zones - s_firstdatazone) << s_log_zone_size
20716     *
20717     * A super_block slot is free if s_dev == NO_DEV.
20718     */
20719
20720
20721    EXTERN struct super_block {
20722      ino_t s_ninodes;              /* # usable inodes on the minor device */
20723      zone1_t  s_nzones;            /* total device size, including bit maps etc */
20724      short s_imap_blocks;          /* # of blocks used by inode bit map */
20725      short s_zmap_blocks;          /* # of blocks used by zone bit map */
20726      zone1_t s_firstdatazone;      /* number of first data zone */
20727      short s_log_zone_size;        /* log2 of blocks/zone */
20728      off_t s_max_size;             /* maximum file size on this device */
20729      short s_magic;                /* magic number to recognize super-blocks */
20730      short s_pad;                  /* try to avoid compiler-dependent padding */
20731      zone_t s_zones;               /* number of zones (replaces s_nzones in V2) */
20732
20733      /* The following items are only used when the super_block is in memory. */
20734      struct inode *s_isup;         /* inode for root dir of mounted file sys */
20735      struct inode *s_imount;       /* inode mounted on */
20736      unsigned s_inodes_per_block;  /* precalculated from magic number */
20737      dev_t s_dev;                  /* whose super block is this? */
20738      int s_rd_only;                /* set to 1 iff file sys mounted read only */
20739      int s_native;                 /* set to 1 iff not byte swapped file system */
20740      int s_version;                /* file system version, zero means bad magic */
20741      int s_ndzones;                /* # direct zones in an inode */
20742      int s_nindirs;                /* # indirect zones per indirect block */
20743      bit_t s_isearch;              /* inodes below this bit number are in use */
20744      bit_t s_zsearch;              /* all zones below this bit number are in use*/
20745    } super_block[NR_SUPERS];
20746
20747    #define NIL_SUPER (struct super_block *) 0
20748    #define IMAP            0       /* operating on the inode bit map */
20749    #define ZMAP            1       /* operating on the zone bit map */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/fs/table.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

20800    /* This file contains the table used to map system call numbers onto the
20801     * routines that perform them.
20802     */
20803
20804    #define _TABLE
20805
20806    #include "fs.h"
20807    #include <minix/callnr.h>
20808    #include <minix/com.h>
20809    #include "buf.h"
20810    #include "dev.h"
20811    #include "file.h"
20812    #include "fproc.h"
20813    #include "inode.h"
20814    #include "lock.h"
20815    #include "super.h"
20816
20817    PUBLIC _PROTOTYPE (int (*call_vector[NCALLS]), (void) ) = {
20818            no_sys,         /*  0 = unused  */
20819            do_exit,        /*  1 = exit    */
20820            do_fork,        /*  2 = fork    */
20821            do_read,        /*  3 = read    */
20822            do_write,       /*  4 = write   */
20823            do_open,        /*  5 = open    */
20824            do_close,       /*  6 = close   */
20825            no_sys,         /*  7 = wait    */
20826            do_creat,       /*  8 = creat   */
20827            do_link,        /*  9 = link    */
20828            do_unlink,      /* 10 = unlink  */
20829            no_sys,         /* 11 = waitpid */
20830            do_chdir,       /* 12 = chdir   */
20831            do_time,        /* 13 = time    */
20832            do_mknod,       /* 14 = mknod   */
20833            do_chmod,       /* 15 = chmod   */
20834            do_chown,       /* 16 = chown   */
20835            no_sys,         /* 17 = break   */
20836            do_stat,        /* 18 = stat    */
20837            do_lseek,       /* 19 = lseek   */
20838            no_sys,         /* 20 = getpid  */
20839            do_mount,       /* 21 = mount   */
20840            do_umount,      /* 22 = umount  */
20841            do_set,         /* 23 = setuid  */
20842            no_sys,         /* 24 = getuid  */
20843            do_stime,       /* 25 = stime   */
20844            no_sys,         /* 26 = ptrace  */
20845            no_sys,         /* 27 = alarm   */
20846            do_fstat,       /* 28 = fstat   */
20847            no_sys,         /* 29 = pause   */
20848            do_utime,       /* 30 = utime   */
20849            no_sys,         /* 31 = (stty)  */
20850            no_sys,         /* 32 = (gtty)  */
20851            do_access,      /* 33 = access  */
20852            no_sys,         /* 34 = (nice)  */
20853            no_sys,         /* 35 = (ftime) */
20854            do_sync,        /* 36 = sync    */
```



```
20855            no_sys,         /* 37 = kill    */
20856            do_rename,      /* 38 = rename  */
20857            do_mkdir,       /* 39 = mkdir   */
20858            do_unlink,      /* 40 = rmdir   */
20859            do_dup,         /* 41 = dup     */
20860            do_pipe,        /* 42 = pipe    */
20861            do_tims,        /* 43 = times   */
20862            no_sys,         /* 44 = (prof)  */
20863            no_sys,         /* 45 = unused  */
20864            do_set,         /* 46 = setgid  */
20865            no_sys,         /* 47 = getgid  */
20866            no_sys,         /* 48 = (signal)*/
20867            no_sys,         /* 49 = unused  */
20868            no_sys,         /* 50 = unused  */
20869            no_sys,         /* 51 = (acct)  */
20870            no_sys,         /* 52 = (phys)  */
20871            no_sys,         /* 53 = (lock)  */
20872            do_ioctl,       /* 54 = ioctl   */
20873            do_fcntl,       /* 55 = fcntl   */
20874            no_sys,         /* 56 = (mpx)   */
20875            no_sys,         /* 57 = unused  */
20876            no_sys,         /* 58 = unused  */
20877            do_exec,        /* 59 = execve  */
20878            do_umask,       /* 60 = umask   */
20879            do_chroot,      /* 61 = chroot  */
20880            do_setsid,      /* 62 = setsid  */
20881            no_sys,         /* 63 = getpgrp */
20882
20883            no_sys,         /* 64 = KSIG: signals originating in the kernel */
20884            do_unpause,     /* 65 = UNPAUSE */
20885            no_sys,         /* 66 = unused  */
20886            do_revive,      /* 67 = REVIVE  */
20887            no_sys,         /* 68 = TASK_REPLY      */
20888            no_sys,         /* 69 = unused */
20889            no_sys,         /* 70 = unused */
20890            no_sys,         /* 71 = SIGACTION */
20891            no_sys,         /* 72 = SIGSUSPEND */
20892            no_sys,         /* 73 = SIGPENDING */
20893            no_sys,         /* 74 = SIGPROCMASK */
20894            no_sys,         /* 75 = SIGRETURN */
20895            no_sys,         /* 76 = REBOOT */
20896    };
20897
20898
20899    /* Some devices may or may not be there in the next table. */
20900    #define DT(enable, open, rw, close, task) \
20901            { (enable ? (open) : no_dev), (enable ? (rw) : no_dev), \
20902              (enable ? (close) : no_dev), (enable ? (task) : 0) },
20903
20904    /* The order of the entries here determines the mapping between major device
20905     * numbers and tasks.  The first entry (major device 0) is not used.  The
20906     * next entry is major device 1, etc.  Character and block devices can be
20907     * intermixed at random.  If this ordering is changed, the devices in
20908     * <include/minix/boot.h> must be changed to correspond to the new values.
20909     * Note that the major device numbers used in /dev are NOT the same as the
20910     * task numbers used inside the kernel (as defined in <include/minix/com.h>).
20911     * Also note that if /dev/mem is changed from 1, NULL_MAJOR must be changed
20912     * in <include/minix/com.h>.
20913     */
20914    PUBLIC struct dmap dmap[] = {
```

```
20915   /*  ?   Open       Read/Write   Close       Task #      Device  File
20916       -   ----       ----------   -----       -------     ------  ----       */
20917     DT(1, no_dev,    no_dev,      no_dev,     0)          /* 0 = not used   */
20918     DT(1, dev_opcl,  call_task,   dev_opcl,   MEM)        /* 1 = /dev/mem   */
20919     DT(1, dev_opcl,  call_task,   dev_opcl,   FLOPPY)     /* 2 = /dev/fd0   */
20920     DT(ENABLE_WINI,
20921           dev_opcl,  call_task,   dev_opcl,   WINCHESTER)  /* 3 = /dev/hd0   */
20922     DT(1, tty_open,  call_task,   dev_opcl,   TTY)        /* 4 = /dev/tty00 */
20923     DT(1, ctty_open, call_ctty,   ctty_close, TTY)        /* 5 = /dev/tty   */
20924     DT(1, dev_opcl,  call_task,   dev_opcl,    PRINTER)     /* 6 = /dev/lp    */
20925
20926   #if (MACHINE == IBM_PC)
20927     DT(ENABLE_NETWORKING,
20928           net_open,  call_task,   dev_opcl,   INET_PROC_NR)/* 7 = /dev/ip    */
20929     DT(ENABLE_CDROM,
20930           dev_opcl,  call_task,   dev_opcl,   CDROM)       /* 8 = /dev/cd0   */
20931     DT(0, 0,         0,          0,          0)           /* 9 = not used   */
20932     DT(ENABLE_SCSI,
20933           dev_opcl,  call_task,   dev_opcl,   SCSI)        /*10 = /dev/sd0   */
20934     DT(0, 0,         0,          0,          0)           /*11 = not used   */
20935     DT(0, 0,         0,          0,          0)           /*12 = not used   */
20936     DT(ENABLE_AUDIO,
20937           dev_opcl,  call_task,   dev_opcl,   AUDIO)       /*13 = /dev/audio */
20938     DT(ENABLE_AUDIO,
20939           dev_opcl,  call_task,   dev_opcl,   MIXER)       /*14 = /dev/mixer */
20940   #endif /* IBM_PC */
20941
20942   #if (MACHINE == ATARI)
20943     DT(ENABLE_SCSI,
20944           dev_opcl,  call_task,   dev_opcl,   SCSI)        /* 7 = /dev/hdscsi0 */
20945   #endif
20946   };
20947
20948   PUBLIC int max_major = sizeof(dmap)/sizeof(struct dmap);
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                              src/fs/cache.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

21000   /* The file system maintains a buffer cache to reduce the number of disk
21001    * accesses needed.  Whenever a read or write to the disk is done, a check is
21002    * first made to see if the block is in the cache.  This file manages the
21003    * cache.
21004    *
21005    * The entry points into this file are:
21006    *   get_block:   request to fetch a block for reading or writing from cache
21007    *   put_block:   return a block previously requested with get_block
21008    *   alloc_zone:  allocate a new zone (to increase the length of a file)
21009    *   free_zone:   release a zone (when a file is removed)
21010    *   rw_block:    read or write a block from the disk itself
21011    *   invalidate:  remove all the cache blocks on some device
21012    */
21013
21014   #include "fs.h"
21015   #include <minix/com.h>
21016   #include <minix/boot.h>
21017   #include "buf.h"
21018   #include "file.h"
21019   #include "fproc.h"
```



```
21020   #include "super.h"
21021
21022   FORWARD _PROTOTYPE( void rm_lru, (struct buf *bp) );
21023
21024   /*===========================================================================*
21025    *                              get_block                                    *
21026    *===========================================================================*/
21027   PUBLIC struct buf *get_block(dev, block, only_search)
21028   register dev_t dev;             /* on which device is the block? */
21029   register block_t block;         /* which block is wanted? */
21030   int only_search;                /* if NO_READ, don't read, else act normal */
21031   {
21032   /* Check to see if the requested block is in the block cache.  If so, return
21033    * a pointer to it.  If not, evict some other block and fetch it (unless
21034    * 'only_search' is 1).  All the blocks in the cache that are not in use
21035    * are linked together in a chain, with 'front' pointing to the least recently
21036    * used block and 'rear' to the most recently used block.  If 'only_search' is
21037    * 1, the block being requested will be overwritten in its entirety, so it is
21038    * only necessary to see if it is in the cache; if it is not, any free buffer
21039    * will do.  It is not necessary to actually read the block in from disk.
21040    * If 'only_search' is PREFETCH, the block need not be read from the disk,
21041    * and the device is not to be marked on the block, so callers can tell if
21042    * the block returned is valid.
21043    * In addition to the LRU chain, there is also a hash chain to link together
21044    * blocks whose block numbers end with the same bit strings, for fast lookup.
21045    */
21046
21047     int b;
21048     register struct buf *bp, *prev_ptr;
21049
21050     /* Search the hash chain for (dev, block). Do_read() can use
21051      * get_block(NO_DEV ...) to get an unnamed block to fill with zeros when
21052      * someone wants to read from a hole in a file, in which case this search
21053      * is skipped
21054      */
21055     if (dev != NO_DEV) {
21056           b = (int) block & HASH_MASK;
21057           bp = buf_hash[b];
21058           while (bp != NIL_BUF) {
21059                   if (bp->b_blocknr == block && bp->b_dev == dev) {
21060                           /* Block needed has been found. */
21061                           if (bp->b_count == 0) rm_lru(bp);
21062                           bp->b_count++;  /* record that block is in use */
21063                           return(bp);
21064                   } else {
21065                           /* This block is not the one sought. */
21066                           bp = bp->b_hash; /* move to next block on hash chain */
21067                   }
21068           }
21069     }
21070
21071     /* Desired block is not on available chain.  Take oldest block ('front'). */
21072     if ((bp = front) == NIL_BUF) panic("all buffers in use", NR_BUFS);
21073     rm_lru(bp);
21074
21075     /* Remove the block that was just taken from its hash chain. */
21076     b = (int) bp->b_blocknr & HASH_MASK;
21077     prev_ptr = buf_hash[b];
21078     if (prev_ptr == bp) {
21079           buf_hash[b] = bp->b_hash;
```

```
21080     } else {
21081           /* The block just taken is not on the front of its hash chain. */
21082           while (prev_ptr->b_hash != NIL_BUF)
21083                   if (prev_ptr->b_hash == bp) {
21084                           prev_ptr->b_hash = bp->b_hash;  /* found it */
21085                           break;
21086                   } else {
21087                           prev_ptr = prev_ptr->b_hash;    /* keep looking */
21088                   }
21089     }
21090
21091     /* If the block taken is dirty, make it clean by writing it to the disk.
21092      * Avoid hysteresis by flushing all other dirty blocks for the same device.
21093      */
21094     if (bp->b_dev != NO_DEV) {
21095           if (bp->b_dirt == DIRTY) flushall(bp->b_dev);
21096     }
21097
21098     /* Fill in block's parameters and add it to the hash chain where it goes. */
21099     bp->b_dev = dev;              /* fill in device number */
21100     bp->b_blocknr = block;        /* fill in block number */
21101     bp->b_count++;                /* record that block is being used */
21102     b = (int) bp->b_blocknr & HASH_MASK;
21103     bp->b_hash = buf_hash[b];
21104     buf_hash[b] = bp;             /* add to hash list */
21105
21106     /* Go get the requested block unless searching or prefetching. */
21107     if (dev != NO_DEV) {
21108           if (only_search == PREFETCH) bp->b_dev = NO_DEV;
21109           else
21110           if (only_search == NORMAL) rw_block(bp, READING);
21111     }
21112     return(bp);                   /* return the newly acquired block */
21113   }


21116   /*===========================================================================*
21117    *                              put_block                                    *
21118    *===========================================================================*/
21119   PUBLIC void put_block(bp, block_type)
21120   register struct buf *bp;        /* pointer to the buffer to be released */
21121   int block_type;                 /* INODE_BLOCK, DIRECTORY_BLOCK, or whatever */
21122   {
21123   /* Return a block to the list of available blocks.   Depending on 'block_type'
21124    * it may be put on the front or rear of the LRU chain.  Blocks that are
21125    * expected to be needed again shortly (e.g., partially full data blocks)
21126    * go on the rear; blocks that are unlikely to be needed again shortly
21127    * (e.g., full data blocks) go on the front.  Blocks whose loss can hurt
21128    * the integrity of the file system (e.g., inode blocks) are written to
21129    * disk immediately if they are dirty.
21130    */
21131
21132     if (bp == NIL_BUF) return;    /* it is easier to check here than in caller */
21133
21134     bp->b_count--;                /* there is one use fewer now */
21135     if (bp->b_count != 0) return; /* block is still in use */
21136
21137     bufs_in_use--;                /* one fewer block buffers in use */
21138
21139     /* Put this block back on the LRU chain.  If the ONE_SHOT bit is set in
```



```
21140      * 'block_type', the block is not likely to be needed again shortly, so put
21141      * it on the front of the LRU chain where it will be the first one to be
21142      * taken when a free buffer is needed later.
21143      */
21144     if (block_type & ONE_SHOT) {
21145           /* Block probably won't be needed quickly. Put it on front of chain.
21146            * It will be the next block to be evicted from the cache.
21147            */
21148           bp->b_prev = NIL_BUF;
21149           bp->b_next = front;
21150           if (front == NIL_BUF)
21151                   rear = bp;      /* LRU chain was empty */
21152           else
21153                   front->b_prev = bp;
21154           front = bp;
21155     } else {
21156           /* Block probably will be needed quickly.  Put it on rear of chain.
21157            * It will not be evicted from the cache for a long time.
21158            */
21159           bp->b_prev = rear;
21160           bp->b_next = NIL_BUF;
21161           if (rear == NIL_BUF)
21162                   front = bp;
21163           else
21164                   rear->b_next = bp;
21165           rear = bp;
21166     }
21167
21168     /* Some blocks are so important (e.g., inodes, indirect blocks) that they
21169      * should be written to the disk immediately to avoid messing up the file
21170      * system in the event of a crash.
21171      */
21172     if ((block_type & WRITE_IMMED) && bp->b_dirt==DIRTY && bp->b_dev != NO_DEV)
21173           rw_block(bp, WRITING);
21174   }


21177   /*===========================================================================*
21178    *                              alloc_zone                                   *
21179    *===========================================================================*/
21180   PUBLIC zone_t alloc_zone(dev, z)
21181   dev_t dev;                      /* device where zone wanted */
21182   zone_t z;                       /* try to allocate new zone near this one */
21183   {
21184   /* Allocate a new zone on the indicated device and return its number. */
21185
21186     int major, minor;
21187     bit_t b, bit;
21188     struct super_block *sp;
21189
21190     /* Note that the routine alloc_bit() returns 1 for the lowest possible
21191      * zone, which corresponds to sp->s_firstdatazone.  To convert a value
21192      * between the bit number, 'b', used by alloc_bit() and the zone number, 'z',
21193      * stored in the inode, use the formula:
21194      *     z = b + sp->s_firstdatazone - 1
21195      * Alloc_bit() never returns 0, since this is used for NO_BIT (failure).
21196      */
21197     sp = get_super(dev);          /* find the super_block for this device */
21198
21199     /* If z is 0, skip initial part of the map known to be fully in use. */
```

```
21200     if (z == sp->s_firstdatazone) {
21201           bit = sp->s_zsearch;
21202     } else {
21203           bit = (bit_t) z - (sp->s_firstdatazone - 1);
21204     }
21205     b = alloc_bit(sp, ZMAP, bit);
21206     if (b == NO_BIT) {
21207           err_code = ENOSPC;
21208           major = (int) (sp->s_dev >> MAJOR) & BYTE;
21209           minor = (int) (sp->s_dev >> MINOR) & BYTE;
21210           printf("No space on %sdevice %d/%d\n",
21211                   sp->s_dev == ROOT_DEV ? "root " : "", major, minor);
21212           return(NO_ZONE);
21213     }
21214     if (z == sp->s_firstdatazone) sp->s_zsearch = b;      /* for next time */
21215     return(sp->s_firstdatazone - 1 + (zone_t) b);
21216   }


21219   /*===========================================================================*
21220    *                                free_zone                                  *
21221    *===========================================================================*/
21222   PUBLIC void free_zone(dev, numb)
21223   dev_t dev;                            /* device where zone located */
21224   zone_t numb;                          /* zone to be returned */
21225   {
21226   /* Return a zone. */
21227
21228     register struct super_block *sp;
21229     bit_t bit;
21230
21231     /* Locate the appropriate super_block and return bit. */
21232     sp = get_super(dev);
21233     if (numb < sp->s_firstdatazone || numb >= sp->s_zones) return;
21234     bit = (bit_t) (numb - (sp->s_firstdatazone - 1));
21235     free_bit(sp, ZMAP, bit);
21236     if (bit < sp->s_zsearch) sp->s_zsearch = bit;
21237   }


21240   /*===========================================================================*
21241    *                                rw_block                                   *
21242    *===========================================================================*/
21243   PUBLIC void rw_block(bp, rw_flag)
21244   register struct buf *bp;        /* buffer pointer */
21245   int rw_flag;                    /* READING or WRITING */
21246   {
21247   /* Read or write a disk block. This is the only routine in which actual disk
21248    * I/O is invoked. If an error occurs, a message is printed here, but the error
21249    * is not reported to the caller.  If the error occurred while purging a block
21250    * from the cache, it is not clear what the caller could do about it anyway.
21251    */
21252
21253     int r, op;
21254     off_t pos;
21255     dev_t dev;
21256
21257     if ( (dev = bp->b_dev) != NO_DEV) {
21258           pos = (off_t) bp->b_blocknr * BLOCK_SIZE;
21259           op = (rw_flag == READING ? DEV_READ : DEV_WRITE);
```



```
21260           r = dev_io(op, FALSE, dev, pos, BLOCK_SIZE, FS_PROC_NR, bp->b_data);
21261           if (r != BLOCK_SIZE) {
21262               if (r >= 0) r = END_OF_FILE;
21263               if (r != END_OF_FILE)
21264                 printf("Unrecoverable disk error on device %d/%d, block %ld\n",
21265                           (dev>>MAJOR)&BYTE, (dev>>MINOR)&BYTE, bp->b_blocknr);
21266                   bp->b_dev = NO_DEV;     /* invalidate block */
21267
21268                   /* Report read errors to interested parties. */
21269                   if (rw_flag == READING) rdwt_err = r;
21270           }
21271     }
21272
21273     bp->b_dirt = CLEAN;
21274   }


21277   /*===========================================================================*
21278    *                                invalidate                                 *
21279    *===========================================================================*/
21280   PUBLIC void invalidate(device)
21281   dev_t device;                   /* device whose blocks are to be purged */
21282   {
21283   /* Remove all the blocks belonging to some device from the cache. */
21284
21285     register struct buf *bp;
21286
21287     for (bp = &buf[0]; bp < &buf[NR_BUFS]; bp++)
21288           if (bp->b_dev == device) bp->b_dev = NO_DEV;
21289   }


21292   /*===========================================================================*
21293    *                                flushall                                   *
21294    *===========================================================================*/
21295   PUBLIC void flushall(dev)
21296   dev_t dev;                      /* device to flush */
21297   {
21298   /* Flush all dirty blocks for one device. */
21299
21300     register struct buf *bp;
21301     static struct buf *dirty[NR_BUFS];    /* static so it isn't on stack */
21302     int ndirty;
21303
21304     for (bp = &buf[0], ndirty = 0; bp < &buf[NR_BUFS]; bp++)
21305           if (bp->b_dirt == DIRTY && bp->b_dev == dev) dirty[ndirty++] = bp;
21306     rw_scattered(dev, dirty, ndirty, WRITING);
21307   }


21310   /*===========================================================================*
21311    *                                rw_scattered                               *
21312    *===========================================================================*/
21313   PUBLIC void rw_scattered(dev, bufq, bufqsize, rw_flag)
21314   dev_t dev;                      /* major-minor device number */
21315   struct buf **bufq;              /* pointer to array of buffers */
21316   int bufqsize;                   /* number of buffers */
21317   int rw_flag;                    /* READING or WRITING */
21318   {
21319   /* Read or write scattered data from a device. */
```

```
21320
21321     register struct buf *bp;
21322     int gap;
21323     register int i;
21324     register struct iorequest_s *iop;
21325     static struct iorequest_s iovec[NR_IOREQS];  /* static so it isn't on stack */
21326     int j;
21327
21328     /* (Shell) sort buffers on b_blocknr. */
21329     gap = 1;
21330     do
21331           gap = 3 * gap + 1;
21332     while (gap <= bufqsize);
21333     while (gap != 1) {
21334           gap /= 3;
21335           for (j = gap; j < bufqsize; j++) {
21336                   for (i = j - gap;
21337                        i >= 0 && bufq[i]->b_blocknr > bufq[i + gap]->b_blocknr;
21338                        i -= gap) {
21339                           bp = bufq[i];
21340                           bufq[i] = bufq[i + gap];
21341                           bufq[i + gap] = bp;
21342                   }
21343           }
21344     }
21345
21346     /* Set up i/o vector and do i/o.  The result of dev_io is discarded because
21347      * all results are returned in the vector.  If dev_io fails completely, the
21348      * vector is unchanged and all results are taken as errors.
21349      */
21350     while (bufqsize > 0) {
21351           for (j = 0, iop = iovec; j < NR_IOREQS && j < bufqsize; j++, iop++) {
21352                   bp = bufq[j];
21353                   iop->io_position = (off_t) bp->b_blocknr * BLOCK_SIZE;
21354                   iop->io_buf = bp->b_data;
21355                   iop->io_nbytes = BLOCK_SIZE;
21356                   iop->io_request = rw_flag == WRITING ?
21357                                     DEV_WRITE : DEV_READ | OPTIONAL_IO;
21358           }
21359           (void) dev_io(SCATTERED_IO, 0, dev, (off_t) 0, j, FS_PROC_NR,
21360                                                          (char *) iovec);
21361
21362           /* Harvest the results.  Leave read errors for rw_block() to complain. */
21363           for (i = 0, iop = iovec; i < j; i++, iop++) {
21364                   bp = bufq[i];
21365                   if (rw_flag == READING) {
21366                       if (iop->io_nbytes == 0)
21367                           bp->b_dev = dev;        /* validate block */
21368                       put_block(bp, PARTIAL_DATA_BLOCK);
21369                   } else {
21370                       if (iop->io_nbytes != 0) {
21371                        printf("Unrecoverable write error on device %d/%d, block %ld\n",
21372                                   (dev>>MAJOR)&BYTE, (dev>>MINOR)&BYTE, bp->b_blocknr);
21373                           bp->b_dev = NO_DEV;     /* invalidate block */
21374                       }
21375                       bp->b_dirt = CLEAN;
21376                   }
21377           }
21378           bufq += j;
21379           bufqsize -= j;
```



```
21380       }
21381     }


21384     /*===========================================================================*
21385      *                              rm_lru                                       *
21386      *===========================================================================*/
21387     PRIVATE void rm_lru(bp)
21388     struct buf *bp;
21389     {
21390     /* Remove a block from its LRU chain. */
21391
21392       struct buf *next_ptr, *prev_ptr;
21393
21394       bufs_in_use++;
21395       next_ptr = bp->b_next;        /* successor on LRU chain */
21396       prev_ptr = bp->b_prev;        /* predecessor on LRU chain */
21397       if (prev_ptr != NIL_BUF)
21398             prev_ptr->b_next = next_ptr;
21399       else
21400             front = next_ptr;       /* this block was at front of chain */
21401
21402       if (next_ptr != NIL_BUF)
21403             next_ptr->b_prev = prev_ptr;
21404       else
21405             rear = prev_ptr;        /* this block was at rear of chain */
21406     }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                    src/fs/inode.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

21500     /* This file manages the inode table.  There are procedures to allocate and
21501      * deallocate inodes, acquire, erase, and release them, and read and write
21502      * them from the disk.
21503      *
21504      * The entry points into this file are
21505      *   get_inode:    search inode table for a given inode; if not there, read it
21506      *   put_inode:    indicate that an inode is no longer needed in memory
21507      *   alloc_inode:  allocate a new, unused inode
21508      *   wipe_inode:   erase some fields of a newly allocated inode
21509      *   free_inode:   mark an inode as available for a new file
21510      *   update_times: update atime, ctime, and mtime
21511      *   rw_inode:     read a disk block and extract an inode, or corresp. write
21512      *   old_icopy:    copy to/from in-core inode struct and disk inode (V1.x)
21513      *   new_icopy:    copy to/from in-core inode struct and disk inode (V2.x)
21514      *   dup_inode:    indicate that someone else is using an inode table entry
21515      */
21516
21517     #include "fs.h"
21518     #include <minix/boot.h>
21519     #include "buf.h"
21520     #include "file.h"
21521     #include "fproc.h"
21522     #include "inode.h"
21523     #include "super.h"
21524
```

```
21525   FORWARD _PROTOTYPE( void old_icopy, (struct inode *rip, d1_inode *dip,
21526                                         int direction, int norm));
21527   FORWARD _PROTOTYPE( void new_icopy, (struct inode *rip, d2_inode *dip,
21528                                         int direction, int norm));
21529
21530
21531   /*===========================================================================*
21532    *                              get_inode                                    *
21533    *===========================================================================*/
21534   PUBLIC struct inode *get_inode(dev, numb)
21535   dev_t dev;                    /* device on which inode resides */
21536   int numb;                     /* inode number (ANSI: may not be unshort) */
21537   {
21538   /* Find a slot in the inode table, load the specified inode into it, and
21539    * return a pointer to the slot.  If 'dev' == NO_DEV, just return a free slot.
21540    */
21541
21542     register struct inode *rip, *xp;
21543
21544     /* Search the inode table both for (dev, numb) and a free slot. */
21545     xp = NIL_INODE;
21546     for (rip = &inode[0]; rip < &inode[NR_INODES]; rip++) {
21547           if (rip->i_count > 0) { /* only check used slots for (dev, numb) */
21548                 if (rip->i_dev == dev && rip->i_num == numb) {
21549                       /* This is the inode that we are looking for. */
21550                       rip->i_count++;
21551                       return(rip);    /* (dev, numb) found */
21552                 }
21553           } else {
21554                 xp = rip;       /* remember this free slot for later */
21555           }
21556     }
21557
21558     /* Inode we want is not currently in use.  Did we find a free slot? */
21559     if (xp == NIL_INODE) {        /* inode table completely full */
21560           err_code = ENFILE;
21561           return(NIL_INODE);
21562     }
21563
21564     /* A free inode slot has been located.  Load the inode into it. */
21565     xp->i_dev = dev;
21566     xp->i_num = numb;
21567     xp->i_count = 1;
21568     if (dev != NO_DEV) rw_inode(xp, READING);     /* get inode from disk */
21569     xp->i_update = 0;             /* all the times are initially up-to-date */
21570
21571     return(xp);
21572   }


21575   /*===========================================================================*
21576    *                              put_inode                                    *
21577    *===========================================================================*/
21578   PUBLIC void put_inode(rip)
21579   register struct inode *rip;     /* pointer to inode to be released */
21580   {
21581   /* The caller is no longer using this inode.  If no one else is using it either
21582    * write it back to the disk immediately.  If it has no links, truncate it and
21583    * return it to the pool of available inodes.
21584    */
```



```
21585
21586     if (rip == NIL_INODE) return; /* checking here is easier than in caller */
21587     if (--rip->i_count == 0) {    /* i_count == 0 means no one is using it now */
21588           if ((rip->i_nlinks & BYTE) == 0) {
21589                   /* i_nlinks == 0 means free the inode. */
21590                   truncate(rip);  /* return all the disk blocks */
21591                   rip->i_mode = I_NOT_ALLOC;      /* clear I_TYPE field */
21592                   rip->i_dirt = DIRTY;
21593                   free_inode(rip->i_dev, rip->i_num);
21594           } else {
21595                   if (rip->i_pipe == I_PIPE) truncate(rip);
21596           }
21597           rip->i_pipe = NO_PIPE;  /* should always be cleared */
21598           if (rip->i_dirt == DIRTY) rw_inode(rip, WRITING);
21599     }
21600   }

21602   /*===========================================================================*
21603    *                              alloc_inode                                  *
21604    *===========================================================================*/
21605   PUBLIC struct inode *alloc_inode(dev, bits)
21606   dev_t dev;                    /* device on which to allocate the inode */
21607   mode_t bits;                  /* mode of the inode */
21608   {
21609   /* Allocate a free inode on 'dev', and return a pointer to it. */
21610
21611     register struct inode *rip;
21612     register struct super_block *sp;
21613     int major, minor, inumb;
21614     bit_t b;
21615
21616     sp = get_super(dev);  /* get pointer to super_block */
21617     if (sp->s_rd_only) {  /* can't allocate an inode on a read only device. */
21618           err_code = EROFS;
21619           return(NIL_INODE);
21620     }
21621
21622     /* Acquire an inode from the bit map. */
21623     b = alloc_bit(sp, IMAP, sp->s_isearch);
21624     if (b == NO_BIT) {
21625           err_code = ENFILE;
21626           major = (int) (sp->s_dev >> MAJOR) & BYTE;
21627           minor = (int) (sp->s_dev >> MINOR) & BYTE;
21628           printf("Out of i-nodes on %sdevice %d/%d\n",
21629                   sp->s_dev == ROOT_DEV ? "root " : "", major, minor);
21630           return(NIL_INODE);
21631     }
21632     sp->s_isearch = b;            /* next time start here */
21633     inumb = (int) b;              /* be careful not to pass unshort as param */
21634
21635     /* Try to acquire a slot in the inode table. */
21636     if ((rip = get_inode(NO_DEV, inumb)) == NIL_INODE) {
21637           /* No inode table slots available.  Free the inode just allocated. */
21638           free_bit(sp, IMAP, b);
21639     } else {
21640           /* An inode slot is available. Put the inode just allocated into it. */
21641           rip->i_mode = bits;             /* set up RWX bits */
21642           rip->i_nlinks = (nlink_t) 0;    /* initial no links */
21643           rip->i_uid = fp->fp_effuid;     /* file's uid is owner's */
21644           rip->i_gid = fp->fp_effgid;     /* ditto group id */
```

```
21645           rip->i_dev = dev;               /* mark which device it is on */
21646           rip->i_ndzones = sp->s_ndzones; /* number of direct zones */
21647           rip->i_nindirs = sp->s_nindirs; /* number of indirect zones per blk*/
21648           rip->i_sp = sp;                 /* pointer to super block */
21649
21650           /* Fields not cleared already are cleared in wipe_inode().  They have
21651            * been put there because truncate() needs to clear the same fields if
21652            * the file happens to be open while being truncated.  It saves space
21653            * not to repeat the code twice.
21654            */
21655           wipe_inode(rip);
21656     }
21657
21658     return(rip);
21659   }

21661   /*===========================================================================*
21662    *                              wipe_inode                                   *
21663    *===========================================================================*/
21664   PUBLIC void wipe_inode(rip)
21665   register struct inode *rip;     /* the inode to be erased */
21666   {
21667   /* Erase some fields in the inode.  This function is called from alloc_inode()
21668    * when a new inode is to be allocated, and from truncate(), when an existing
21669    * inode is to be truncated.
21670    */
21671
21672     register int i;
21673
21674     rip->i_size = 0;
21675     rip->i_update = ATIME | CTIME | MTIME;        /* update all times later */
21676     rip->i_dirt = DIRTY;
21677     for (i = 0; i < V2_NR_TZONES; i++) rip->i_zone[i] = NO_ZONE;
21678   }


21681   /*===========================================================================*
21682    *                              free_inode                                   *
21683    *===========================================================================*/
21684   PUBLIC void free_inode(dev, inumb)
21685   dev_t dev;                      /* on which device is the inode */
21686   ino_t inumb;                    /* number of inode to be freed */
21687   {
21688   /* Return an inode to the pool of unallocated inodes. */
21689
21690     register struct super_block *sp;
21691     bit_t b;
21692
21693     /* Locate the appropriate super_block. */
21694     sp = get_super(dev);
21695     if (inumb <= 0 || inumb > sp->s_ninodes) return;
21696     b = inumb;
21697     free_bit(sp, IMAP, b);
21698     if (b < sp->s_isearch) sp->s_isearch = b;
21699   }

21701   /*===========================================================================*
21702    *                              update_times                                 *
21703    *===========================================================================*/
21704   PUBLIC void update_times(rip)
```



```
21705   register struct inode *rip;     /* pointer to inode to be read/written */
21706   {
21707   /* Various system calls are required by the standard to update atime, ctime,
21708    * or mtime.  Since updating a time requires sending a message to the clock
21709    * task--an expensive business--the times are marked for update by setting
21710    * bits in i_update.  When a stat, fstat, or sync is done, or an inode is
21711    * released, update_times() may be called to actually fill in the times.
21712    */
21713
21714     time_t cur_time;
21715     struct super_block *sp;
21716
21717     sp = rip->i_sp;               /* get pointer to super block. */
21718     if (sp->s_rd_only) return;    /* no updates for read-only file systems */
21719
21720     cur_time = clock_time();
21721     if (rip->i_update & ATIME) rip->i_atime = cur_time;
21722     if (rip->i_update & CTIME) rip->i_ctime = cur_time;
21723     if (rip->i_update & MTIME) rip->i_mtime = cur_time;
21724     rip->i_update = 0;            /* they are all up-to-date now */
21725   }


21728   /*===========================================================================*
21729    *                              rw_inode                                     *
21730    *===========================================================================*/
21731   PUBLIC void rw_inode(rip, rw_flag)
21732   register struct inode *rip;     /* pointer to inode to be read/written */
21733   int rw_flag;                    /* READING or WRITING */
21734   {
21735   /* An entry in the inode table is to be copied to or from the disk. */
21736
21737     register struct buf *bp;
21738     register struct super_block *sp;
21739     d1_inode *dip;
21740     d2_inode *dip2;
21741     block_t b, offset;
21742
21743     /* Get the block where the inode resides. */
21744     sp = get_super(rip->i_dev);   /* get pointer to super block */
21745     rip->i_sp = sp;               /* inode must contain super block pointer */
21746     offset = sp->s_imap_blocks + sp->s_zmap_blocks + 2;
21747     b = (block_t) (rip->i_num - 1)/sp->s_inodes_per_block + offset;
21748     bp = get_block(rip->i_dev, b, NORMAL);
21749     dip  = bp->b_v1_ino + (rip->i_num - 1) % V1_INODES_PER_BLOCK;
21750     dip2 = bp->b_v2_ino + (rip->i_num - 1) % V2_INODES_PER_BLOCK;
21751
21752     /* Do the read or write. */
21753     if (rw_flag == WRITING) {
21754           if (rip->i_update) update_times(rip);   /* times need updating */
21755           if (sp->s_rd_only == FALSE) bp->b_dirt = DIRTY;
21756     }
21757
21758     /* Copy the inode from the disk block to the in-core table or vice versa.
21759      * If the fourth parameter below is FALSE, the bytes are swapped.
21760      */
21761     if (sp->s_version == V1)
21762           old_icopy(rip, dip,  rw_flag, sp->s_native);
21763     else
21764           new_icopy(rip, dip2, rw_flag, sp->s_native);
```

```
21765
21766     put_block(bp, INODE_BLOCK);
21767     rip->i_dirt = CLEAN;
21768   }


21771   /*===========================================================================*
21772    *                              old_icopy                                    *
21773    *===========================================================================*/
21774   PRIVATE void old_icopy(rip, dip, direction, norm)
21775   register struct inode *rip;     /* pointer to the in-core inode struct */
21776   register d1_inode *dip;         /* pointer to the d1_inode inode struct */
21777   int direction;                  /* READING (from disk) or WRITING (to disk) */
21778   int norm;                       /* TRUE = do not swap bytes; FALSE = swap */
21779
21780   {
21781   /* The V1.x IBM disk, the V1.x 68000 disk, and the V2 disk (same for IBM and
21782    * 68000) all have different inode layouts.  When an inode is read or written
21783    * this routine handles the conversions so that the information in the inode
21784    * table is independent of the disk structure from which the inode came.
21785    * The old_icopy routine copies to and from V1 disks.
21786    */
21787
21788     int i;
21789
21790     if (direction == READING) {
21791           /* Copy V1.x inode to the in-core table, swapping bytes if need be. */
21792           rip->i_mode    = conv2(norm, (int) dip->d1_mode);
21793           rip->i_uid     = conv2(norm, (int) dip->d1_uid );
21794           rip->i_size    = conv4(norm,       dip->d1_size);
21795           rip->i_mtime   = conv4(norm,       dip->d1_mtime);
21796           rip->i_atime   = rip->i_mtime;
21797           rip->i_ctime   = rip->i_mtime;
21798           rip->i_nlinks  = (nlink_t) dip->d1_nlinks;      /* 1 char */
21799           rip->i_gid     = (gid_t) dip->d1_gid;           /* 1 char */
21800           rip->i_ndzones = V1_NR_DZONES;
21801           rip->i_nindirs = V1_INDIRECTS;
21802           for (i = 0; i < V1_NR_TZONES; i++)
21803                   rip->i_zone[i] = conv2(norm, (int) dip->d1_zone[i]);
21804     } else {
21805           /* Copying V1.x inode to disk from the in-core table. */
21806           dip->d1_mode   = conv2(norm, (int) rip->i_mode);
21807           dip->d1_uid    = conv2(norm, (int) rip->i_uid );
21808           dip->d1_size   = conv4(norm,       rip->i_size);
21809           dip->d1_mtime  = conv4(norm,       rip->i_mtime);
21810           dip->d1_nlinks = (nlink_t) rip->i_nlinks;       /* 1 char */
21811           dip->d1_gid    = (gid_t) rip->i_gid;            /* 1 char */
21812           for (i = 0; i < V1_NR_TZONES; i++)
21813                   dip->d1_zone[i] = conv2(norm, (int) rip->i_zone[i]);
21814     }
21815   }


21818   /*===========================================================================*
21819    *                              new_icopy                                    *
21820    *===========================================================================*/
21821   PRIVATE void new_icopy(rip, dip, direction, norm)
21822   register struct inode *rip;     /* pointer to the in-core inode struct */
21823   register d2_inode *dip; /* pointer to the d2_inode struct */
21824   int direction;                  /* READING (from disk) or WRITING (to disk) */
```



```
21825   int norm;                       /* TRUE = do not swap bytes; FALSE = swap */
21826
21827   {
21828   /* Same as old_icopy, but to/from V2 disk layout. */
21829
21830     int i;
21831
21832     if (direction == READING) {
21833           /* Copy V2.x inode to the in-core table, swapping bytes if need be. */
21834           rip->i_mode    = conv2(norm,dip->d2_mode);
21835           rip->i_uid     = conv2(norm,dip->d2_uid );
21836           rip->i_nlinks  = conv2(norm,(int) dip->d2_nlinks);
21837           rip->i_gid     = conv2(norm,(int) dip->d2_gid );
21838           rip->i_size    = conv4(norm,dip->d2_size);
21839           rip->i_atime   = conv4(norm,dip->d2_atime);
21840           rip->i_ctime   = conv4(norm,dip->d2_ctime);
21841           rip->i_mtime   = conv4(norm,dip->d2_mtime);
21842           rip->i_ndzones = V2_NR_DZONES;
21843           rip->i_nindirs = V2_INDIRECTS;
21844           for (i = 0; i < V2_NR_TZONES; i++)
21845                   rip->i_zone[i] = conv4(norm, (long) dip->d2_zone[i]);
21846     } else {
21847           /* Copying V2.x inode to disk from the in-core table. */
21848           dip->d2_mode   = conv2(norm,rip->i_mode);
21849           dip->d2_uid    = conv2(norm,rip->i_uid );
21850           dip->d2_nlinks = conv2(norm,rip->i_nlinks);
21851           dip->d2_gid    = conv2(norm,rip->i_gid );
21852           dip->d2_size   = conv4(norm,rip->i_size);
21853           dip->d2_atime  = conv4(norm,rip->i_atime);
21854           dip->d2_ctime  = conv4(norm,rip->i_ctime);
21855           dip->d2_mtime  = conv4(norm,rip->i_mtime);
21856           for (i = 0; i < V2_NR_TZONES; i++)
21857                   dip->d2_zone[i] = conv4(norm, (long) rip->i_zone[i]);
21858     }
21859   }


21862   /*===========================================================================*
21863    *                              dup_inode                                    *
21864    *===========================================================================*/
21865   PUBLIC void dup_inode(ip)
21866   struct inode *ip;               /* The inode to be duplicated. */
21867   {
21868   /* This routine is a simplified form of get_inode() for the case where
21869    * the inode pointer is already known.
21870    */
21871
21872     ip->i_count++;
21873   }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                              src/fs/super.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

21900   /* This file manages the super block table and the related data structures,
21901    * namely, the bit maps that keep track of which zones and which inodes are
21902    * allocated and which are free.  When a new inode or zone is needed, the
21903    * appropriate bit map is searched for a free entry.
21904    *
21905    * The entry points into this file are
21906    *   alloc_bit:       somebody wants to allocate a zone or inode; find one
21907    *   free_bit:        indicate that a zone or inode is available for allocation
21908    *   get_super:       search the 'superblock' table for a device
21909    *   mounted:         tells if file inode is on mounted (or ROOT) file system
21910    *   read_super:      read a superblock
21911    */
21912
21913   #include "fs.h"
21914   #include <string.h>
21915   #include <minix/boot.h>
21916   #include "buf.h"
21917   #include "inode.h"
21918   #include "super.h"
21919
21920   #define BITCHUNK_BITS   (usizeof(bitchunk_t) * CHAR_BIT)
21921   #define BITS_PER_BLOCK  (BITMAP_CHUNKS * BITCHUNK_BITS)
21922
21923   /*===========================================================================*
21924    *                              alloc_bit                                    *
21925    *===========================================================================*/
21926   PUBLIC bit_t alloc_bit(sp, map, origin)
21927   struct super_block *sp;         /* the filesystem to allocate from */
21928   int map;                        /* IMAP (inode map) or ZMAP (zone map) */
21929   bit_t origin;                   /* number of bit to start searching at */
21930   {
21931   /* Allocate a bit from a bit map and return its bit number. */
21932
21933     block_t start_block;          /* first bit block */
21934     bit_t map_bits;               /* how many bits are there in the bit map? */
21935     unsigned bit_blocks;          /* how many blocks are there in the bit map? */
21936     unsigned block, word, bcount, wcount;
21937     struct buf *bp;
21938     bitchunk_t *wptr, *wlim, k;
21939     bit_t i, b;
21940
21941     if (sp->s_rd_only)
21942           panic("can't allocate bit on read-only filesys.", NO_NUM);
21943
21944     if (map == IMAP) {
21945           start_block = SUPER_BLOCK + 1;
21946           map_bits = sp->s_ninodes + 1;
21947           bit_blocks = sp->s_imap_blocks;
21948     } else {
21949           start_block = SUPER_BLOCK + 1 + sp->s_imap_blocks;
21950           map_bits = sp->s_zones - (sp->s_firstdatazone - 1);
21951           bit_blocks = sp->s_zmap_blocks;
21952     }
21953
21954     /* Figure out where to start the bit search (depends on 'origin'). */
```



```
21955     if (origin >= map_bits) origin = 0;   /* for robustness */
21956
21957     /* Locate the starting place. */
21958     block = origin / BITS_PER_BLOCK;
21959     word = (origin % BITS_PER_BLOCK) / BITCHUNK_BITS;
21960
21961     /* Iterate over all blocks plus one, because we start in the middle. */
21962     bcount = bit_blocks + 1;
21963     do {
21964           bp = get_block(sp->s_dev, start_block + block, NORMAL);
21965           wlim = &bp->b_bitmap[BITMAP_CHUNKS];
21966
21967           /* Iterate over the words in block. */
21968           for (wptr = &bp->b_bitmap[word]; wptr < wlim; wptr++) {
21969
21970                 /* Does this word contain a free bit? */
21971                 if (*wptr == (bitchunk_t) ~0) continue;
21972
21973                 /* Find and allocate the free bit. */
21974                 k = conv2(sp->s_native, (int) *wptr);
21975                 for (i = 0; (k & (1 << i)) != 0; ++i) {}
21976
21977                 /* Bit number from the start of the bit map. */
21978                 b = ((bit_t) block * BITS_PER_BLOCK)
21979                     + (wptr - &bp->b_bitmap[0]) * BITCHUNK_BITS
21980                     + i;
21981
21982                 /* Don't allocate bits beyond the end of the map. */
21983                 if (b >= map_bits) break;
21984
21985                 /* Allocate and return bit number. */
21986                 k |= 1 << i;
21987                 *wptr = conv2(sp->s_native, (int) k);
21988                 bp->b_dirt = DIRTY;
21989                 put_block(bp, MAP_BLOCK);
21990                 return(b);
21991           }
21992           put_block(bp, MAP_BLOCK);
21993           if (++block >= bit_blocks) block = 0;   /* last block, wrap around */
21994           word = 0;
21995     } while (--bcount > 0);
21996     return(NO_BIT);               /* no bit could be allocated */
21997   }


22000   /*===========================================================================*
22001    *                              free_bit                                     *
22002    *===========================================================================*/
22003   PUBLIC void free_bit(sp, map, bit_returned)
22004   struct super_block *sp;         /* the filesystem to operate on */
22005   int map;                        /* IMAP (inode map) or ZMAP (zone map) */
22006   bit_t bit_returned;             /* number of bit to insert into the map */
22007   {
22008   /* Return a zone or inode by turning off its bitmap bit. */
22009
22010     unsigned block, word, bit;
22011     struct buf *bp;
22012     bitchunk_t k, mask;
22013     block_t start_block;
22014
```

```
22015     if (sp->s_rd_only)
22016           panic("can't free bit on read-only filesys.", NO_NUM);
22017
22018     if (map == IMAP) {
22019           start_block = SUPER_BLOCK + 1;
22020     } else {
22021           start_block = SUPER_BLOCK + 1 + sp->s_imap_blocks;
22022     }
22023     block = bit_returned / BITS_PER_BLOCK;
22024     word = (bit_returned % BITS_PER_BLOCK) / BITCHUNK_BITS;
22025     bit = bit_returned % BITCHUNK_BITS;
22026     mask = 1 << bit;
22027
22028     bp = get_block(sp->s_dev, start_block + block, NORMAL);
22029
22030     k = conv2(sp->s_native, (int) bp->b_bitmap[word]);
22031     if (!(k & mask)) {
22032           panic(map == IMAP ? "tried to free unused inode" :
22033                 "tried to free unused block", NO_NUM);
22034     }
22035
22036     k &= ~mask;
22037     bp->b_bitmap[word] = conv2(sp->s_native, (int) k);
22038     bp->b_dirt = DIRTY;
22039
22040     put_block(bp, MAP_BLOCK);
22041   }


22044   /*===========================================================================*
22045    *                              get_super                                    *
22046    *===========================================================================*/
22047   PUBLIC struct super_block *get_super(dev)
22048   dev_t dev;                    /* device number whose super_block is sought */
22049   {
22050   /* Search the superblock table for this device.  It is supposed to be there. */
22051
22052     register struct super_block *sp;
22053
22054     for (sp = &super_block[0]; sp < &super_block[NR_SUPERS]; sp++)
22055           if (sp->s_dev == dev) return(sp);
22056
22057     /* Search failed.  Something wrong. */
22058     panic("can't find superblock for device (in decimal)", (int) dev);
22059
22060     return(NIL_SUPER);          /* to keep the compiler and lint quiet */
22061   }


22064   /*===========================================================================*
22065    *                              mounted                                      *
22066    *===========================================================================*/
22067   PUBLIC int mounted(rip)
22068   register struct inode *rip;     /* pointer to inode */
22069   {
22070   /* Report on whether the given inode is on a mounted (or ROOT) file system. */
22071
22072     register struct super_block *sp;
22073     register dev_t dev;
22074
```



```
22075     dev = (dev_t) rip->i_zone[0];
22076     if (dev == ROOT_DEV) return(TRUE);    /* inode is on root file system */
22077
22078     for (sp = &super_block[0]; sp < &super_block[NR_SUPERS]; sp++)
22079           if (sp->s_dev == dev) return(TRUE);
22080
22081     return(FALSE);
22082   }


22085   /*===========================================================================*
22086    *                              read_super                                   *
22087    *===========================================================================*/
22088   PUBLIC int read_super(sp)
22089   register struct super_block *sp; /* pointer to a superblock */
22090   {
22091   /* Read a superblock. */
22092
22093     register struct buf *bp;
22094     dev_t dev;
22095     int magic;
22096     int version, native;
22097
22098     dev = sp->s_dev;              /* save device (will be overwritten by copy) */
22099     bp = get_block(sp->s_dev, SUPER_BLOCK, NORMAL);
22100     memcpy( (char *) sp, bp->b_data, (size_t) SUPER_SIZE);
22101     put_block(bp, ZUPER_BLOCK);
22102     sp->s_dev = NO_DEV;           /* restore later */
22103     magic = sp->s_magic;          /* determines file system type */
22104
22105     /* Get file system version and type. */
22106     if (magic == SUPER_MAGIC || magic == conv2(BYTE_SWAP, SUPER_MAGIC)) {
22107           version = V1;
22108           native  = (magic == SUPER_MAGIC);
22109     } else if (magic == SUPER_V2 || magic == conv2(BYTE_SWAP, SUPER_V2)) {
22110           version = V2;
22111           native  = (magic == SUPER_V2);
22112     } else {
22113           return(EINVAL);
22114     }
22115
22116     /* If the super block has the wrong byte order, swap the fields; the magic
22117      * number doesn't need conversion. */
22118     sp->s_ninodes =       conv2(native, (int) sp->s_ninodes);
22119     sp->s_nzones =        conv2(native, (int) sp->s_nzones);
22120     sp->s_imap_blocks =   conv2(native, (int) sp->s_imap_blocks);
22121     sp->s_zmap_blocks =   conv2(native, (int) sp->s_zmap_blocks);
22122     sp->s_firstdatazone = conv2(native, (int) sp->s_firstdatazone);
22123     sp->s_log_zone_size = conv2(native, (int) sp->s_log_zone_size);
22124     sp->s_max_size =      conv4(native, sp->s_max_size);
22125     sp->s_zones =         conv4(native, sp->s_zones);
22126
22127     /* In V1, the device size was kept in a short, s_nzones, which limited
22128      * devices to 32K zones.  For V2, it was decided to keep the size as a
22129      * long.  However, just changing s_nzones to a long would not work, since
22130      * then the position of s_magic in the super block would not be the same
22131      * in V1 and V2 file systems, and there would be no way to tell whether
22132      * a newly mounted file system was V1 or V2.  The solution was to introduce
22133      * a new variable, s_zones, and copy the size there.
22134      *
```

```
22135       * Calculate some other numbers that depend on the version here too, to
22136       * hide some of the differences.
22137       */
22138      if (version == V1) {
22139            sp->s_zones = sp->s_nzones;     /* only V1 needs this copy */
22140            sp->s_inodes_per_block = V1_INODES_PER_BLOCK;
22141            sp->s_ndzones = V1_NR_DZONES;
22142            sp->s_nindirs = V1_INDIRECTS;
22143      } else {
22144            sp->s_inodes_per_block = V2_INODES_PER_BLOCK;
22145            sp->s_ndzones = V2_NR_DZONES;
22146            sp->s_nindirs = V2_INDIRECTS;
22147      }
22148
22149      sp->s_isearch = 0;            /* inode searches initially start at 0 */
22150      sp->s_zsearch = 0;            /* zone searches initially start at 0 */
22151      sp->s_version = version;
22152      sp->s_native  = native;
22153
22154      /* Make a few basic checks to see if super block looks reasonable. */
22155      if (sp->s_imap_blocks < 1 || sp->s_zmap_blocks < 1
22156                                    || sp->s_ninodes < 1 || sp->s_zones < 1
22157                                    || (unsigned) sp->s_log_zone_size > 4) {
22158            return(EINVAL);
22159      }
22160      sp->s_dev = dev;              /* restore device number */
22161      return(OK);
22162    }
```

++

src/fs/filedes.c

++

```
22200    /* This file contains the procedures that manipulate file descriptors.
22201     *
22202     * The entry points into this file are
22203     *   get_fd:    look for free file descriptor and free filp slots
22204     *   get_filp:  look up the filp entry for a given file descriptor
22205     *   find_filp: find a filp slot that points to a given inode
22206     */
22207
22208    #include "fs.h"
22209    #include "file.h"
22210    #include "fproc.h"
22211    #include "inode.h"
22212
22213    /*===========================================================================*
22214     *                              get_fd                                       *
22215     *===========================================================================*/
22216    PUBLIC int get_fd(start, bits, k, fpt)
22217    int start;                    /* start of search (used for F_DUPFD) */
22218    mode_t bits;                  /* mode of the file to be created (RWX bits) */
22219    int *k;                       /* place to return file descriptor */
22220    struct filp **fpt;            /* place to return filp slot */
22221    {
22222    /* Look for a free file descriptor and a free filp slot.  Fill in the mode word
22223     * in the latter, but don't claim either one yet, since the open() or creat()
22224     * may yet fail.
```



```
22225     */
22226
22227      register struct filp *f;
22228      register int i;
22229
22230      *k = -1;                      /* we need a way to tell if file desc found */
22231
22232      /* Search the fproc fp_filp table for a free file descriptor. */
22233      for (i = start; i < OPEN_MAX; i++) {
22234            if (fp->fp_filp[i] == NIL_FILP) {
22235                  /* A file descriptor has been located. */
22236                  *k = i;
22237                  break;
22238            }
22239      }
22240
22241      /* Check to see if a file descriptor has been found. */
22242      if (*k < 0) return(EMFILE);   /* this is why we initialized k to -1 */
22243
22244      /* Now that a file descriptor has been found, look for a free filp slot. */
22245      for (f = &filp[0]; f < &filp[NR_FILPS]; f++) {
22246            if (f->filp_count == 0) {
22247                  f->filp_mode = bits;
22248                  f->filp_pos = 0L;
22249                  f->filp_flags = 0;
22250                  *fpt = f;
22251                  return(OK);
22252            }
22253      }
22254
22255      /* If control passes here, the filp table must be full.  Report that back. */
22256      return(ENFILE);
22257    }


22260    /*===========================================================================*
22261     *                              get_filp                                     *
22262     *===========================================================================*/
22263    PUBLIC struct filp *get_filp(fild)
22264    int fild;                     /* file descriptor */
22265    {
22266    /* See if 'fild' refers to a valid file descr.  If so, return its filp ptr. */
22267
22268      err_code = EBADF;
22269      if (fild < 0 || fild >= OPEN_MAX ) return(NIL_FILP);
22270      return(fp->fp_filp[fild]);    /* may also be NIL_FILP */
22271    }


22274    /*===========================================================================*
22275     *                              find_filp                                    *
22276     *===========================================================================*/
22277    PUBLIC struct filp *find_filp(rip, bits)
22278    register struct inode *rip;   /* inode referred to by the filp to be found */
22279    Mode_t bits;                  /* mode of the filp to be found (RWX bits) */
22280    {
22281    /* Find a filp slot that refers to the inode 'rip' in a way as described
22282     * by the mode bit 'bits'. Used for determining whether somebody is still
22283     * interested in either end of a pipe.  Also used when opening a FIFO to
22284     * find partners to share a filp field with (to shared the file position).
```

```
22285     * Like 'get_fd' it performs its job by linear search through the filp table.
22286     */
22287
22288      register struct filp *f;
22289
22290      for (f = &filp[0]; f < &filp[NR_FILPS]; f++) {
22291            if (f->filp_count != 0 && f->filp_ino == rip && (f->filp_mode & bits)){
22292                  return(f);
22293            }
22294      }
22295
22296      /* If control passes here, the filp wasn't there.  Report that back. */
22297      return(NIL_FILP);
22298    }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                              src/fs/lock.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
22300    /* This file handles advisory file locking as required by POSIX.
22301     *
22302     * The entry points into this file are
22303     *   lock_op:   perform locking operations for FCNTL system call
22304     *   lock_revive: revive processes when a lock is released
22305     */
22306
22307    #include "fs.h"
22308    #include <fcntl.h>
22309    #include <unistd.h>     /* cc runs out of memory with unistd.h :-( */
22310    #include "file.h"
22311    #include "fproc.h"
22312    #include "inode.h"
22313    #include "lock.h"
22314    #include "param.h"
22315
22316    /*===========================================================================*
22317     *                              lock_op                                      *
22318     *===========================================================================*/
22319    PUBLIC int lock_op(f, req)
22320    struct filp *f;
22321    int req;                        /* either F_SETLK or F_SETLKW */
22322    {
22323    /* Perform the advisory locking required by POSIX. */
22324
22325      int r, ltype, i, conflict = 0, unlocking = 0;
22326      mode_t mo;
22327      off_t first, last;
22328      struct flock flock;
22329      vir_bytes user_flock;
22330      struct file_lock *flp, *flp2, *empty;
22331
22332      /* Fetch the flock structure from user space. */
22333      user_flock = (vir_bytes) name1;
22334      r = sys_copy(who, D, (phys_bytes) user_flock,
22335            FS_PROC_NR, D, (phys_bytes) &flock, (phys_bytes) sizeof(flock));
22336      if (r != OK) return(EINVAL);
22337
22338      /* Make some error checks. */
22339      ltype = flock.l_type;
```



```
22340      mo = f->filp_mode;
22341      if (ltype != F_UNLCK && ltype != F_RDLCK && ltype != F_WRLCK) return(EINVAL);
22342      if (req == F_GETLK && ltype == F_UNLCK) return(EINVAL);
22343      if ( (f->filp_ino->i_mode & I_TYPE) != I_REGULAR) return(EINVAL);
22344      if (req != F_GETLK && ltype == F_RDLCK && (mo & R_BIT) == 0) return(EBADF);
22345      if (req != F_GETLK && ltype == F_WRLCK && (mo & W_BIT) == 0) return(EBADF);
22346
22347      /* Compute the first and last bytes in the lock region. */
22348      switch (flock.l_whence) {
22349            case SEEK_SET:  first = 0; break;
22350            case SEEK_CUR:  first = f->filp_pos; break;
22351            case SEEK_END:  first = f->filp_ino->i_size; break;
22352            default:        return(EINVAL);
22353      }
22354      /* Check for overflow. */
22355      if (((long)flock.l_start > 0) && ((first + flock.l_start) < first))
22356            return(EINVAL);
22357      if (((long)flock.l_start < 0) && ((first + flock.l_start) > first))
22358            return(EINVAL);
22359      first = first + flock.l_start;
22360      last = first + flock.l_len - 1;
22361      if (flock.l_len == 0) last = MAX_FILE_POS;
22362      if (last < first) return(EINVAL);
22363
22364      /* Check if this region conflicts with any existing lock. */
22365      empty = (struct file_lock *) 0;
22366      for (flp = &file_lock[0]; flp < & file_lock[NR_LOCKS]; flp++) {
22367            if (flp->lock_type == 0) {
22368                  if (empty == (struct file_lock *) 0) empty = flp;
22369                  continue;       /* 0 means unused slot */
22370            }
22371            if (flp->lock_inode != f->filp_ino) continue;   /* different file */
22372            if (last < flp->lock_first) continue;   /* new one is in front */
22373            if (first > flp->lock_last) continue;   /* new one is afterwards */
22374            if (ltype == F_RDLCK && flp->lock_type == F_RDLCK) continue;
22375            if (ltype != F_UNLCK && flp->lock_pid == fp->fp_pid) continue;
22376
22377            /* There might be a conflict.  Process it. */
22378            conflict = 1;
22379            if (req == F_GETLK) break;
22380
22381            /* If we are trying to set a lock, it just failed. */
22382            if (ltype == F_RDLCK || ltype == F_WRLCK) {
22383                  if (req == F_SETLK) {
22384                        /* For F_SETLK, just report back failure. */
22385                        return(EAGAIN);
22386                  } else {
22387                        /* For F_SETLKW, suspend the process. */
22388                        suspend(XLOCK);
22389                        return(0);
22390                  }
22391            }
22392
22393            /* We are clearing a lock and we found something that overlaps. */
22394            unlocking = 1;
22395            if (first <= flp->lock_first && last >= flp->lock_last) {
22396                  flp->lock_type = 0;     /* mark slot as unused */
22397                  nr_locks--;             /* number of locks is now 1 less */
22398                  continue;
22399            }
```

```
22400
22401           /* Part of a locked region has been unlocked. */
22402           if (first <= flp->lock_first) {
22403                   flp->lock_first = last + 1;
22404                   continue;
22405           }
22406
22407           if (last >= flp->lock_last) {
22408                   flp->lock_last = first - 1;
22409                   continue;
22410           }
22411
22412           /* Bad luck. A lock has been split in two by unlocking the middle. */
22413           if (nr_locks == NR_LOCKS) return(ENOLCK);
22414           for (i = 0; i < NR_LOCKS; i++)
22415                   if (file_lock[i].lock_type == 0) break;
22416           flp2 = &file_lock[i];
22417           flp2->lock_type = flp->lock_type;
22418           flp2->lock_pid = flp->lock_pid;
22419           flp2->lock_inode = flp->lock_inode;
22420           flp2->lock_first = last + 1;
22421           flp2->lock_last = flp->lock_last;
22422           flp->lock_last = first - 1;
22423           nr_locks++;
22424       }
22425       if (unlocking) lock_revive();
22426
22427       if (req == F_GETLK) {
22428           if (conflict) {
22429                   /* GETLK and conflict. Report on the conflicting lock. */
22430                   flock.l_type = flp->lock_type;
22431                   flock.l_whence = SEEK_SET;
22432                   flock.l_start = flp->lock_first;
22433                   flock.l_len = flp->lock_last - flp->lock_first + 1;
22434                   flock.l_pid = flp->lock_pid;
22435
22436           } else {
22437                   /* It is GETLK and there is no conflict. */
22438                   flock.l_type = F_UNLCK;
22439           }
22440
22441           /* Copy the flock structure back to the caller. */
22442           r = sys_copy(FS_PROC_NR, D, (phys_bytes) &flock,
22443                   who, D, (phys_bytes) user_flock, (phys_bytes) sizeof(flock));
22444           return(r);
22445       }
22446
22447       if (ltype == F_UNLCK) return(OK);     /* unlocked a region with no locks */
22448
22449       /* There is no conflict.  If space exists, store new lock in the table. */
22450       if (empty == (struct file_lock *) 0) return(ENOLCK);  /* table full */
22451       empty->lock_type = ltype;
22452       empty->lock_pid = fp->fp_pid;
22453       empty->lock_inode = f->filp_ino;
22454       empty->lock_first = first;
22455       empty->lock_last = last;
22456       nr_locks++;
22457       return(OK);
22458   }
```



```
22460   /*===========================================================================*
22461    *                              lock_revive                                  *
22462    *===========================================================================*/
22463   PUBLIC void lock_revive()
22464   {
22465   /* Go find all the processes that are waiting for any kind of lock and
22466    * revive them all.  The ones that are still blocked will block again when
22467    * they run.  The others will complete.  This strategy is a space-time
22468    * tradeoff.  Figuring out exactly which ones to unblock now would take
22469    * extra code, and the only thing it would win would be some performance in
22470    * extremely rare circumstances (namely, that somebody actually used
22471    * locking).
22472    */
22473
22474     int task;
22475     struct fproc *fptr;
22476
22477     for (fptr = &fproc[INIT_PROC_NR + 1]; fptr < &fproc[NR_PROCS]; fptr++){
22478           task = -fptr->fp_task;
22479           if (fptr->fp_suspended == SUSPENDED && task == XLOCK) {
22480                   revive( (int) (fptr - fproc), 0);
22481           }
22482     }
22483   }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/fs/main.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

22500   /* This file contains the main program of the File System.  It consists of
22501    * a loop that gets messages requesting work, carries out the work, and sends
22502    * replies.
22503    *
22504    * The entry points into this file are
22505    *   main:      main program of the File System
22506    *   reply:     send a reply to a process after the requested work is done
22507    */
22508
22509   struct super_block;             /* proto.h needs to know this */
22510
22511   #include "fs.h"
22512   #include <fcntl.h>
22513   #include <string.h>
22514   #include <sys/ioctl.h>
22515   #include <minix/callnr.h>
22516   #include <minix/com.h>
22517   #include <minix/boot.h>
22518   #include "buf.h"
22519   #include "dev.h"
22520   #include "file.h"
22521   #include "fproc.h"
22522   #include "inode.h"
22523   #include "param.h"
22524   #include "super.h"
22525
22526   FORWARD _PROTOTYPE( void buf_pool, (void)                               );
22527   FORWARD _PROTOTYPE( void fs_init, (void)                                );
22528   FORWARD _PROTOTYPE( void get_boot_parameters, (void)                    );
22529   FORWARD _PROTOTYPE( void get_work, (void)                               );
```

```
22530   FORWARD _PROTOTYPE( void load_ram, (void)                               );
22531   FORWARD _PROTOTYPE( void load_super, (Dev_t super_dev)                  );
22532
22533
22534   /*===========================================================================*
22535    *                              main                                         *
22536    *===========================================================================*/
22537   PUBLIC void main()
22538   {
22539   /* This is the main program of the file system.  The main loop consists of
22540    * three major activities: getting new work, processing the work, and sending
22541    * the reply.  This loop never terminates as long as the file system runs.
22542    */
22543     int error;
22544
22545     fs_init();
22546
22547     /* This is the main loop that gets work, processes it, and sends replies. */
22548     while (TRUE) {
22549           get_work();             /* sets who and fs_call */
22550
22551           fp = &fproc[who];       /* pointer to proc table struct */
22552           super_user = (fp->fp_effuid == SU_UID ? TRUE : FALSE);   /* su? */
22553           dont_reply = FALSE;     /* in other words, do reply is default */
22554
22555           /* Call the internal function that does the work. */
22556           if (fs_call < 0 || fs_call >= NCALLS)
22557                   error = EBADCALL;
22558           else
22559                   error = (*call_vector[fs_call])();
22560
22561           /* Copy the results back to the user and send reply. */
22562           if (dont_reply) continue;
22563           reply(who, error);
22564           if (rdahed_inode != NIL_INODE) read_ahead(); /* do block read ahead */
22565     }
22566   }


22569   /*===========================================================================*
22570    *                              get_work                                     *
22571    *===========================================================================*/
22572   PRIVATE void get_work()
22573   {
22574     /* Normally wait for new input.  However, if 'reviving' is
22575      * nonzero, a suspended process must be awakened.
22576      */
22577
22578     register struct fproc *rp;
22579
22580     if (reviving != 0) {
22581           /* Revive a suspended process. */
22582           for (rp = &fproc[0]; rp < &fproc[NR_PROCS]; rp++)
22583                   if (rp->fp_revived == REVIVING) {
22584                           who = (int)(rp - fproc);
22585                           fs_call = rp->fp_fd & BYTE;
22586                           fd = (rp->fp_fd >>8) & BYTE;
22587                           buffer = rp->fp_buffer;
22588                           nbytes = rp->fp_nbytes;
22589                           rp->fp_suspended = NOT_SUSPENDED; /*no longer hanging*/
```



```
22590                           rp->fp_revived = NOT_REVIVING;
22591                           reviving--;
22592                           return;
22593                   }
22594           panic("get_work couldn't revive anyone", NO_NUM);
22595     }
22596
22597     /* Normal case.  No one to revive. */
22598     if (receive(ANY, &m) != OK) panic("fs receive error", NO_NUM);
22599
22600     who = m.m_source;
22601     fs_call = m.m_type;
22602   }


22605   /*===========================================================================*
22606    *                              reply                                        *
22607    *===========================================================================*/
22608   PUBLIC void reply(whom, result)
22609   int whom;                       /* process to reply to */
22610   int result;                     /* result of the call (usually OK or error #) */
22611   {
22612   /* Send a reply to a user process. It may fail (if the process has just
22613    * been killed by a signal), so don't check the return code.  If the send
22614    * fails, just ignore it.
22615    */
22616
22617     reply_type = result;
22618     send(whom, &m1);
22619   }


22622   /*===========================================================================*
22623    *                              fs_init                                      *
22624    *===========================================================================*/
22625   PRIVATE void fs_init()
22626   {
22627   /* Initialize global variables, tables, etc. */
22628
22629     register struct inode *rip;
22630     int i;
22631     message mess;
22632
22633     /* The following initializations are needed to let dev_opcl succeed .*/
22634     fp = (struct fproc *) NULL;
22635     who = FS_PROC_NR;
22636
22637     buf_pool();                   /* initialize buffer pool */
22638     get_boot_parameters();        /* get the parameters from the menu */
22639     load_ram();                   /* init RAM disk, load if it is root */
22640     load_super(ROOT_DEV);         /* load super block for root device */
22641
22642     /* Initialize the 'fproc' fields for process 0 .. INIT. */
22643     for (i = 0; i <= LOW_USER; i+= 1) {
22644           if (i == FS_PROC_NR) continue;  /* do not initialize FS */
22645           fp = &fproc[i];
22646           rip = get_inode(ROOT_DEV, ROOT_INODE);
22647           fp->fp_rootdir = rip;
22648           dup_inode(rip);
22649           fp->fp_workdir = rip;
```

```
22650           fp->fp_realuid = (uid_t) SYS_UID;
22651           fp->fp_effuid = (uid_t) SYS_UID;
22652           fp->fp_realgid = (gid_t) SYS_GID;
22653           fp->fp_effgid = (gid_t) SYS_GID;
22654           fp->fp_umask = ~0;
22655       }
22656
22657       /* Certain relations must hold for the file system to work at all. */
22658       if (SUPER_SIZE > BLOCK_SIZE) panic("SUPER_SIZE > BLOCK_SIZE", NO_NUM);
22659       if (BLOCK_SIZE % V2_INODE_SIZE != 0)  /* this checks V1_INODE_SIZE too */
22660           panic("BLOCK_SIZE % V2_INODE_SIZE != 0", NO_NUM);
22661       if (OPEN_MAX > 127) panic("OPEN_MAX > 127", NO_NUM);
22662       if (NR_BUFS < 6) panic("NR_BUFS < 6", NO_NUM);
22663       if (V1_INODE_SIZE != 32) panic("V1 inode size != 32", NO_NUM);
22664       if (V2_INODE_SIZE != 64) panic("V2 inode size != 64", NO_NUM);
22665       if (OPEN_MAX > 8 * sizeof(long)) panic("Too few bits in fp_cloexec", NO_NUM);
22666
22667       /* Tell the memory task where my process table is for the sake of ps(1). */
22668       mess.m_type = DEV_IOCTL;
22669       mess.PROC_NR = FS_PROC_NR;
22670       mess.REQUEST = MIOCSPSINFO;
22671       mess.ADDRESS = (void *) fproc;
22672       (void) sendrec(MEM, &mess);
22673   }


22676   /*===========================================================================*
22677    *                              buf_pool                                     *
22678    *===========================================================================*/
22679   PRIVATE void buf_pool()
22680   {
22681   /* Initialize the buffer pool. */
22682
22683     register struct buf *bp;
22684
22685     bufs_in_use = 0;
22686     front = &buf[0];
22687     rear = &buf[NR_BUFS - 1];
22688
22689     for (bp = &buf[0]; bp < &buf[NR_BUFS]; bp++) {
22690           bp->b_blocknr = NO_BLOCK;
22691           bp->b_dev = NO_DEV;
22692           bp->b_next = bp + 1;
22693           bp->b_prev = bp - 1;
22694     }
22695     buf[0].b_prev = NIL_BUF;
22696     buf[NR_BUFS - 1].b_next = NIL_BUF;
22697
22698     for (bp = &buf[0]; bp < &buf[NR_BUFS]; bp++) bp->b_hash = bp->b_next;
22699     buf_hash[0] = front;
22700   }


22703   /*===========================================================================*
22704    *                              get_boot_parameters                          *
22705    *===========================================================================*/
22706   PUBLIC struct bparam_s boot_parameters;
22707
22708   PRIVATE void get_boot_parameters()
22709   {
```



```
22710   /* Ask kernel for boot parameters. */
22711
22712     m1.m_type = SYS_GBOOT;
22713     m1.PROC1 = FS_PROC_NR;
22714     m1.MEM_PTR = (char *) &boot_parameters;
22715     (void) sendrec(SYSTASK, &m1);
22716   }


22719   /*===========================================================================*
22720    *                              load_ram                                     *
22721    *===========================================================================*/
22722   PRIVATE void load_ram()
22723   {
22724   /* If the root device is the RAM disk, copy the entire root image device
22725    * block-by-block to a RAM disk with the same size as the image.
22726    * Otherwise, just allocate a RAM disk with size given in the boot parameters.
22727    */
22728
22729     register struct buf *bp, *bp1;
22730     long k_loaded, lcount;
22731     u32_t ram_size, fsmax;
22732     zone_t zones;
22733     struct super_block *sp, *dsp;
22734     block_t b;
22735     int major, task;
22736     message dev_mess;
22737
22738     ram_size = boot_parameters.bp_ramsize;
22739
22740     /* Open the root device. */
22741     major = (ROOT_DEV >> MAJOR) & BYTE;   /* major device nr */
22742     task = dmap[major].dmap_task;         /* device task nr */
22743     dev_mess.m_type = DEV_OPEN;           /* distinguish from close */
22744     dev_mess.DEVICE = ROOT_DEV;
22745     dev_mess.COUNT = R_BIT|W_BIT;
22746     (*dmap[major].dmap_open)(task, &dev_mess);
22747     if (dev_mess.REP_STATUS != OK) panic("Cannot open root device",NO_NUM);
22748
22749     /* If the root device is the ram disk then fill it from the image device. */
22750     if (ROOT_DEV == DEV_RAM) {
22751           major = (IMAGE_DEV >> MAJOR) & BYTE;    /* major device nr */
22752           task = dmap[major].dmap_task;           /* device task nr */
22753           dev_mess.m_type = DEV_OPEN;             /* distinguish from close */
22754           dev_mess.DEVICE = IMAGE_DEV;
22755           dev_mess.COUNT = R_BIT;
22756           (*dmap[major].dmap_open)(task, &dev_mess);
22757           if (dev_mess.REP_STATUS != OK) panic("Cannot open root device", NO_NUM);
22758
22759           /* Get size of RAM disk by reading root file system's super block. */
22760           sp = &super_block[0];
22761           sp->s_dev = IMAGE_DEV;
22762           if (read_super(sp) != OK) panic("Bad root file system", NO_NUM);
22763
22764           lcount = sp->s_zones << sp->s_log_zone_size;    /* # blks on root dev*/
22765
22766           /* Stretch the RAM disk file system to the boot parameters size, but
22767            * no further than the last zone bit map block allows.
22768            */
22769           if (ram_size < lcount) ram_size = lcount;
```

```
22770           fsmax = (u32_t) sp->s_zmap_blocks * CHAR_BIT * BLOCK_SIZE;
22771           fsmax = (fsmax + (sp->s_firstdatazone-1)) << sp->s_log_zone_size;
22772           if (ram_size > fsmax) ram_size = fsmax;
22773       }
22774
22775       /* Tell RAM driver how big the RAM disk must be. */
22776       m1.m_type = DEV_IOCTL;
22777       m1.PROC_NR = FS_PROC_NR;
22778       m1.REQUEST = MIOCRAMSIZE;
22779       m1.POSITION = ram_size;
22780       if (sendrec(MEM, &m1) != OK || m1.REP_STATUS != OK)
22781             panic("Can't set RAM disk size", NO_NUM);
22782
22783       /* Tell MM the RAM disk size, and wait for it to come "on-line". */
22784       m1.m1_i1 = ((long) ram_size * BLOCK_SIZE) >> CLICK_SHIFT;
22785       if (sendrec(MM_PROC_NR, &m1) != OK)
22786             panic("FS can't sync up with MM", NO_NUM);
22787
22788       /* If the root device is not the RAM disk, it doesn't need loading. */
22789       if (ROOT_DEV != DEV_RAM) return;
22790
22791       /* Copy the blocks one at a time from the image to the RAM disk. */
22792       printf("Loading RAM disk.\33[23CLoaded:    0K ");
22793
22794       inode[0].i_mode = I_BLOCK_SPECIAL;    /* temp inode for rahead() */
22795       inode[0].i_size = LONG_MAX;
22796       inode[0].i_dev = IMAGE_DEV;
22797       inode[0].i_zone[0] = IMAGE_DEV;
22798
22799       for (b = 0; b < (block_t) lcount; b++) {
22800             bp = rahead(&inode[0], b, (off_t)BLOCK_SIZE * b, BLOCK_SIZE);
22801             bp1 = get_block(ROOT_DEV, b, NO_READ);
22802             memcpy(bp1->b_data, bp->b_data, (size_t) BLOCK_SIZE);
22803             bp1->b_dirt = DIRTY;
22804             put_block(bp, FULL_DATA_BLOCK);
22805             put_block(bp1, FULL_DATA_BLOCK);
22806             k_loaded = ( (long) b * BLOCK_SIZE)/1024L;      /* K loaded so far */
22807             if (k_loaded % 5 == 0) printf("\b\b\b\b\b\b\b%5ldK ", k_loaded);
22808       }
22809
22810       printf("\rRAM disk loaded.\33[K\n\n");
22811
22812       /* Close and invalidate image device. */
22813       dev_mess.m_type = DEV_CLOSE;
22814       dev_mess.DEVICE = IMAGE_DEV;
22815       (*dmap[major].dmap_close)(task, &dev_mess);
22816       invalidate(IMAGE_DEV);
22817
22818       /* Resize the RAM disk root file system. */
22819       bp = get_block(ROOT_DEV, SUPER_BLOCK, NORMAL);
22820       dsp = (struct super_block *) bp->b_data;
22821       zones = ram_size >> sp->s_log_zone_size;
22822       dsp->s_nzones = conv2(sp->s_native, (u16_t) zones);
22823       dsp->s_zones = conv4(sp->s_native, zones);
22824       bp->b_dirt = DIRTY;
22825       put_block(bp, ZUPER_BLOCK);
22826   }


22829   /*===========================================================================*
```



```
22830    *                               load_super                                  *
22831    *===========================================================================*/
22832   PRIVATE void load_super(super_dev)
22833   dev_t super_dev;                        /* place to get superblock from */
22834   {
22835     int bad;
22836     register struct super_block *sp;
22837     register struct inode *rip;
22838
22839     /* Initialize the super_block table. */
22840     for (sp = &super_block[0]; sp < &super_block[NR_SUPERS]; sp++)
22841           sp->s_dev = NO_DEV;
22842
22843     /* Read in super_block for the root file system. */
22844     sp = &super_block[0];
22845     sp->s_dev = super_dev;
22846
22847     /* Check super_block for consistency (is it the right diskette?). */
22848     bad = (read_super(sp) != OK);
22849     if (!bad) {
22850           rip = get_inode(super_dev, ROOT_INODE); /* inode for root dir */
22851           if ( (rip->i_mode & I_TYPE) != I_DIRECTORY || rip->i_nlinks < 3) bad++;
22852     }
22853     if (bad)panic("Invalid root file system.  Possibly wrong diskette.",NO_NUM);
22854
22855     sp->s_imount = rip;
22856     dup_inode(rip);
22857     sp->s_isup = rip;
22858     sp->s_rd_only = 0;
22859     return;
22860   }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                  src/fs/open.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

22900   /* This file contains the procedures for creating, opening, closing, and
22901    * seeking on files.
22902    *
22903    * The entry points into this file are
22904    *   do_creat:  perform the CREAT system call
22905    *   do_open:   perform the OPEN system call
22906    *   do_mknod:  perform the MKNOD system call
22907    *   do_mkdir:  perform the MKDIR system call
22908    *   do_close:  perform the CLOSE system call
22909    *   do_lseek:  perform the LSEEK system call
22910    */
22911
22912   #include "fs.h"
22913   #include <sys/stat.h>
22914   #include <fcntl.h>
22915   #include <minix/callnr.h>
22916   #include <minix/com.h>
22917   #include "buf.h"
22918   #include "dev.h"
22919   #include "file.h"
```

```
22920  #include "fproc.h"
22921  #include "inode.h"
22922  #include "lock.h"
22923  #include "param.h"
22924
22925  PRIVATE message dev_mess;
22926  PRIVATE char mode_map[] = {R_BIT, W_BIT, R_BIT|W_BIT, 0};
22927
22928  FORWARD _PROTOTYPE( int common_open, (int oflags, Mode_t omode)         );
22929  FORWARD _PROTOTYPE( int pipe_open, (struct inode *rip,Mode_t bits,int oflags));
22930  FORWARD _PROTOTYPE( struct inode *new_node, (char *path, Mode_t bits,
22931                                                  zone_t z0)      );
22932
22933
22934  /*===========================================================================*
22935   *                              do_creat                                     *
22936   *===========================================================================*/
22937  PUBLIC int do_creat()
22938  {
22939  /* Perform the creat(name, mode) system call. */
22940    int r;
22941
22942    if (fetch_name(name, name_length, M3) != OK) return(err_code);
22943    r = common_open(O_WRONLY | O_CREAT | O_TRUNC, (mode_t) mode);
22944    return(r);
22945  }


22948  /*===========================================================================*
22949   *                              do_open                                      *
22950   *===========================================================================*/
22951  PUBLIC int do_open()
22952  {
22953  /* Perform the open(name, flags,...) system call. */
22954
22955    int create_mode = 0;          /* is really mode_t but this gives problems */
22956    int r;
22957
22958    /* If O_CREAT is set, open has three parameters, otherwise two. */
22959    if (mode & O_CREAT) {
22960          create_mode = c_mode;
22961          r = fetch_name(c_name, name1_length, M1);
22962    } else {
22963          r = fetch_name(name, name_length, M3);
22964    }
22965
22966    if (r != OK) return(err_code); /* name was bad */
22967    r = common_open(mode, create_mode);
22968    return(r);
22969  }


22972  /*===========================================================================*
22973   *                              common_open                                  *
22974   *===========================================================================*/
22975  PRIVATE int common_open(oflags, omode)
22976  register int oflags;
22977  mode_t omode;
22978  {
22979  /* Common code from do_creat and do_open. */
```



```
22980
22981    register struct inode *rip;
22982    int r, b, major, task, exist = TRUE;
22983    dev_t dev;
22984    mode_t bits;
22985    off_t pos;
22986    struct filp *fil_ptr, *filp2;
22987
22988    /* Remap the bottom two bits of oflags. */
22989    bits = (mode_t) mode_map[oflags & O_ACCMODE];
22990
22991    /* See if file descriptor and filp slots are available. */
22992    if ( (r = get_fd(0, bits, &fd, &fil_ptr)) != OK) return(r);
22993
22994    /* If O_CREATE is set, try to make the file. */
22995    if (oflags & O_CREAT) {
22996          /* Create a new inode by calling new_node(). */
22997          omode = I_REGULAR | (omode & ALL_MODES & fp->fp_umask);
22998          rip = new_node(user_path, omode, NO_ZONE);
22999          r = err_code;
23000          if (r == OK) exist = FALSE;      /* we just created the file */
23001          else if (r != EEXIST) return(r); /* other error */
23002          else exist = !(oflags & O_EXCL); /* file exists, if the O_EXCL
23003                                              flag is set this is an error */
23004    } else {
23005           /* Scan path name. */
23006          if ( (rip = eat_path(user_path)) == NIL_INODE) return(err_code);
23007    }
23008
23009    /* Claim the file descriptor and filp slot and fill them in. */
23010    fp->fp_filp[fd] = fil_ptr;
23011    fil_ptr->filp_count = 1;
23012    fil_ptr->filp_ino = rip;
23013    fil_ptr->filp_flags = oflags;
23014
23015    /* Only do the normal open code if we didn't just create the file. */
23016    if (exist) {
23017          /* Check protections. */
23018          if ((r = forbidden(rip, bits)) == OK) {
23019                  /* Opening reg. files directories and special files differ. */
23020                  switch (rip->i_mode & I_TYPE) {
23021                     case I_REGULAR:
23022                          /* Truncate regular file if O_TRUNC. */
23023                          if (oflags & O_TRUNC) {
23024                                  if ((r = forbidden(rip, W_BIT)) !=OK) break;
23025                                  truncate(rip);
23026                                  wipe_inode(rip);
23027                                  /* Send the inode from the inode cache to the
23028                                   * block cache, so it gets written on the next
23029                                   * cache flush.
23030                                   */
23031                                  rw_inode(rip, WRITING);
23032                          }
23033                          break;
23034
23035                     case I_DIRECTORY:
23036                          /* Directories may be read but not written. */
23037                          r = (bits & W_BIT ? EISDIR : OK);
23038                          break;
23039
```

```
23040                   case I_CHAR_SPECIAL:
23041                   case I_BLOCK_SPECIAL:
23042                        /* Invoke the driver for special processing. */
23043                        dev_mess.m_type = DEV_OPEN;
23044                        dev = (dev_t) rip->i_zone[0];
23045                        dev_mess.DEVICE = dev;
23046                        dev_mess.COUNT = bits | (oflags & ~O_ACCMODE);
23047                        major = (dev >> MAJOR) & BYTE;  /* major device nr */
23048                        if (major <= 0 || major >= max_major) {
23049                                r = ENODEV;
23050                                break;
23051                        }
23052                        task = dmap[major].dmap_task;   /* device task nr */
23053                        (*dmap[major].dmap_open)(task, &dev_mess);
23054                        r = dev_mess.REP_STATUS;
23055                        break;
23056
23057                   case I_NAMED_PIPE:
23058                        oflags |= O_APPEND;     /* force append mode */
23059                        fil_ptr->filp_flags = oflags;
23060                        r = pipe_open(rip, bits, oflags);
23061                        if (r == OK) {
23062                                /* See if someone else is doing a rd or wt on
23063                                 * the FIFO.  If so, use its filp entry so the
23064                                 * file position will be automatically shared.
23065                                 */
23066                                b = (bits & R_BIT ? R_BIT : W_BIT);
23067                                fil_ptr->filp_count = 0; /* don't find self */
23068                                if ((filp2 = find_filp(rip, b)) != NIL_FILP) {
23069                                        /* Co-reader or writer found. Use it.*/
23070                                        fp->fp_filp[fd] = filp2;
23071                                        filp2->filp_count++;
23072                                        filp2->filp_ino = rip;
23073                                        filp2->filp_flags = oflags;
23074
23075                                        /* i_count was incremented incorrectly
23076                                         * by eatpath above, not knowing that
23077                                         * we were going to use an existing
23078                                         * filp entry.  Correct this error.
23079                                         */
23080                                        rip->i_count--;
23081                                } else {
23082                                        /* Nobody else found.  Restore filp. */
23083                                        fil_ptr->filp_count = 1;
23084                                        if (b == R_BIT)
23085                                             pos = rip->i_zone[V2_NR_DZONES+1];
23086                                        else
23087                                             pos = rip->i_zone[V2_NR_DZONES+2];
23088                                        fil_ptr->filp_pos = pos;
23089                                }
23090                        }
23091                        break;
23092                }
23093           }
23094      }
23095
23096      /* If error, release inode. */
23097      if (r != OK) {
23098           fp->fp_filp[fd] = NIL_FILP;
23099           fil_ptr->filp_count= 0;
```



```
23100           put_inode(rip);
23101           return(r);
23102      }
23103
23104      return(fd);
23105   }


23108   /*===========================================================================*
23109    *                              new_node                                     *
23110    *===========================================================================*/
23111   PRIVATE struct inode *new_node(path, bits, z0)
23112   char *path;                     /* pointer to path name */
23113   mode_t bits;                    /* mode of the new inode */
23114   zone_t z0;                      /* zone number 0 for new inode */
23115   {
23116   /* New_node() is called by common_open(), do_mknod(), and do_mkdir().
23117    * In all cases it allocates a new inode, makes a directory entry for it on
23118    * the path 'path', and initializes it.  It returns a pointer to the inode if
23119    * it can do this; otherwise it returns NIL_INODE.  It always sets 'err_code'
23120    * to an appropriate value (OK or an error code).
23121    */
23122
23123     register struct inode *rlast_dir_ptr, *rip;
23124     register int r;
23125     char string[NAME_MAX];
23126
23127     /* See if the path can be opened down to the last directory. */
23128     if ((rlast_dir_ptr = last_dir(path, string)) == NIL_INODE) return(NIL_INODE);
23129
23130     /* The final directory is accessible. Get final component of the path. */
23131     rip = advance(rlast_dir_ptr, string);
23132     if ( rip == NIL_INODE && err_code == ENOENT) {
23133           /* Last path component does not exist.  Make new directory entry. */
23134           if ( (rip = alloc_inode(rlast_dir_ptr->i_dev, bits)) == NIL_INODE) {
23135                   /* Can't creat new inode: out of inodes. */
23136                   put_inode(rlast_dir_ptr);
23137                   return(NIL_INODE);
23138           }
23139
23140           /* Force inode to the disk before making directory entry to make
23141            * the system more robust in the face of a crash: an inode with
23142            * no directory entry is much better than the opposite.
23143            */
23144           rip->i_nlinks++;
23145           rip->i_zone[0] = z0;            /* major/minor device numbers */
23146           rw_inode(rip, WRITING);         /* force inode to disk now */
23147
23148           /* New inode acquired.  Try to make directory entry. */
23149           if ((r = search_dir(rlast_dir_ptr, string, &rip->i_num,ENTER)) != OK) {
23150                   put_inode(rlast_dir_ptr);
23151                   rip->i_nlinks--;        /* pity, have to free disk inode */
23152                   rip->i_dirt = DIRTY;    /* dirty inodes are written out */
23153                   put_inode(rip); /* this call frees the inode */
23154                   err_code = r;
23155                   return(NIL_INODE);
23156           }
23157
23158     } else {
23159           /* Either last component exists, or there is some problem. */
```

```
23160           if (rip != NIL_INODE)
23161                   r = EEXIST;
23162           else
23163                   r = err_code;
23164   }
23165
23166   /* Return the directory inode and exit. */
23167   put_inode(rlast_dir_ptr);
23168   err_code = r;
23169   return(rip);
23170 }


23173 /*===========================================================================*
23174  *                              pipe_open                                    *
23175  *===========================================================================*/
23176 PRIVATE int pipe_open(rip, bits, oflags)
23177 register struct inode *rip;
23178 register mode_t bits;
23179 register int oflags;
23180 {
23181 /*  This function is called from common_open. It checks if
23182  *  there is at least one reader/writer pair for the pipe, if not
23183  *  it suspends the caller, otherwise it revives all other blocked
23184  *  processes hanging on the pipe.
23185  */
23186
23187   if (find_filp(rip, bits & W_BIT ? R_BIT : W_BIT) == NIL_FILP) {
23188         if (oflags & O_NONBLOCK) {
23189                 if (bits & W_BIT) return(ENXIO);
23190         } else
23191                 suspend(XPOPEN);        /* suspend caller */
23192   } else if (susp_count > 0) {/* revive blocked processes */
23193         release(rip, OPEN, susp_count);
23194         release(rip, CREAT, susp_count);
23195   }
23196   rip->i_pipe = I_PIPE;
23197
23198   return(OK);
23199 }


23202 /*===========================================================================*
23203  *                              do_mknod                                     *
23204  *===========================================================================*/
23205 PUBLIC int do_mknod()
23206 {
23207 /* Perform the mknod(name, mode, addr) system call. */
23208
23209   register mode_t bits, mode_bits;
23210   struct inode *ip;
23211
23212   /* Only the super_user may make nodes other than fifos. */
23213   mode_bits = (mode_t) m.m1_i2; /* mode of the inode */
23214   if (!super_user && ((mode_bits & I_TYPE) != I_NAMED_PIPE)) return(EPERM);
23215   if (fetch_name(m.m1_p1, m.m1_i1, M1) != OK) return(err_code);
23216   bits = (mode_bits & I_TYPE) | (mode_bits & ALL_MODES & fp->fp_umask);
23217   ip = new_node(user_path, bits, (zone_t) m.m1_i3);
23218   put_inode(ip);
23219   return(err_code);
```



```
23220 }


23223 /*===========================================================================*
23224  *                              do_mkdir                                     *
23225  *===========================================================================*/
23226 PUBLIC int do_mkdir()
23227 {
23228 /* Perform the mkdir(name, mode) system call. */
23229
23230   int r1, r2;                   /* status codes */
23231   ino_t dot, dotdot;            /* inode numbers for . and .. */
23232   mode_t bits;                  /* mode bits for the new inode */
23233   char string[NAME_MAX];        /* last component of the new dir's path name */
23234   register struct inode *rip, *ldirp;
23235
23236   /* Check to see if it is possible to make another link in the parent dir. */
23237   if (fetch_name(name1, name1_length, M1) != OK) return(err_code);
23238   ldirp = last_dir(user_path, string);  /* pointer to new dir's parent */
23239   if (ldirp == NIL_INODE) return(err_code);
23240   if ( (ldirp->i_nlinks & BYTE) >= LINK_MAX) {
23241         put_inode(ldirp);       /* return parent */
23242         return(EMLINK);
23243   }
23244
23245   /* Next make the inode. If that fails, return error code. */
23246   bits = I_DIRECTORY | (mode & RWX_MODES & fp->fp_umask);
23247   rip = new_node(user_path, bits, (zone_t) 0);
23248   if (rip == NIL_INODE || err_code == EEXIST) {
23249         put_inode(rip);         /* can't make dir: it already exists */
23250         put_inode(ldirp);       /* return parent too */
23251         return(err_code);
23252   }
23253
23254   /* Get the inode numbers for . and .. to enter in the directory. */
23255   dotdot = ldirp->i_num;        /* parent's inode number */
23256   dot = rip->i_num;             /* inode number of the new dir itself */
23257
23258   /* Now make dir entries for . and .. unless the disk is completely full. */
23259   /* Use dot1 and dot2, so the mode of the directory isn't important. */
23260   rip->i_mode = bits;   /* set mode */
23261   r1 = search_dir(rip, dot1, &dot, ENTER);      /* enter . in the new dir */
23262   r2 = search_dir(rip, dot2, &dotdot, ENTER);   /* enter .. in the new dir */
23263
23264   /* If both . and .. were successfully entered, increment the link counts. */
23265   if (r1 == OK && r2 == OK) {
23266         /* Normal case.  It was possible to enter . and .. in the new dir. */
23267         rip->i_nlinks++;        /* this accounts for . */
23268         ldirp->i_nlinks++;      /* this accounts for .. */
23269         ldirp->i_dirt = DIRTY;  /* mark parent's inode as dirty */
23270   } else {
23271         /* It was not possible to enter . or .. probably disk was full. */
23272         (void) search_dir(ldirp, string, (ino_t *) 0, DELETE);
23273         rip->i_nlinks--;        /* undo the increment done in new_node() */
23274   }
23275   rip->i_dirt = DIRTY;          /* either way, i_nlinks has changed */
23276
23277   put_inode(ldirp);             /* return the inode of the parent dir */
23278   put_inode(rip);               /* return the inode of the newly made dir */
23279   return(err_code);             /* new_node() always sets 'err_code' */
```

```
23280   }


23283   /*===========================================================================*
23284    *                              do_close                                     *
23285    *===========================================================================*/
23286   PUBLIC int do_close()
23287   {
23288   /* Perform the close(fd) system call. */
23289
23290     register struct filp *rfilp;
23291     register struct inode *rip;
23292     struct file_lock *flp;
23293     int rw, mode_word, major, task, lock_count;
23294     dev_t dev;
23295
23296     /* First locate the inode that belongs to the file descriptor. */
23297     if ( (rfilp = get_filp(fd)) == NIL_FILP) return(err_code);
23298     rip = rfilp->filp_ino;        /* 'rip' points to the inode */
23299
23300     if (rfilp->filp_count - 1 == 0 && rfilp->filp_mode != FILP_CLOSED) {
23301           /* Check to see if the file is special. */
23302           mode_word = rip->i_mode & I_TYPE;
23303           if (mode_word == I_CHAR_SPECIAL || mode_word == I_BLOCK_SPECIAL) {
23304                   dev = (dev_t) rip->i_zone[0];
23305                   if (mode_word == I_BLOCK_SPECIAL)  {
23306                           /* Invalidate cache entries unless special is mounted
23307                            * or ROOT
23308                            */
23309                           if (!mounted(rip)) {
23310                                   (void) do_sync();       /* purge cache */
23311                                   invalidate(dev);
23312                           }
23313                   }
23314                   /* Use the dmap_close entry to do any special processing
23315                    * required.
23316                    */
23317                   dev_mess.m_type = DEV_CLOSE;
23318                   dev_mess.DEVICE = dev;
23319                   major = (dev >> MAJOR) & BYTE;  /* major device nr */
23320                   task = dmap[major].dmap_task;   /* device task nr */
23321                   (*dmap[major].dmap_close)(task, &dev_mess);
23322           }
23323     }
23324
23325     /* If the inode being closed is a pipe, release everyone hanging on it. */
23326     if (rip->i_pipe == I_PIPE) {
23327           rw = (rfilp->filp_mode & R_BIT ? WRITE : READ);
23328           release(rip, rw, NR_PROCS);
23329     }
23330
23331     /* If a write has been done, the inode is already marked as DIRTY. */
23332     if (--rfilp->filp_count == 0) {
23333           if (rip->i_pipe == I_PIPE && rip->i_count > 1) {
23334                   /* Save the file position in the i-node in case needed later.
23335                    * The read and write positions are saved separately.  The
23336                    * last 3 zones in the i-node are not used for (named) pipes.
23337                    */
23338                   if (rfilp->filp_mode == R_BIT)
23339                           rip->i_zone[V2_NR_DZONES+1] = (zone_t) rfilp->filp_pos;
```



```
23340                   else
23341                           rip->i_zone[V2_NR_DZONES+2] = (zone_t) rfilp->filp_pos;
23342           }
23343           put_inode(rip);
23344     }
23345
23346     fp->fp_cloexec &= ~(1L << fd);        /* turn off close-on-exec bit */
23347     fp->fp_filp[fd] = NIL_FILP;
23348
23349     /* Check to see if the file is locked.  If so, release all locks. */
23350     if (nr_locks == 0) return(OK);
23351     lock_count = nr_locks;        /* save count of locks */
23352     for (flp = &file_lock[0]; flp < &file_lock[NR_LOCKS]; flp++) {
23353           if (flp->lock_type == 0) continue;      /* slot not in use */
23354           if (flp->lock_inode == rip && flp->lock_pid == fp->fp_pid) {
23355                   flp->lock_type = 0;
23356                   nr_locks--;
23357           }
23358     }
23359     if (nr_locks < lock_count) lock_revive();     /* lock released */
23360     return(OK);
23361   }


23364   /*===========================================================================*
23365    *                              do_lseek                                     *
23366    *===========================================================================*/
23367   PUBLIC int do_lseek()
23368   {
23369   /* Perform the lseek(ls_fd, offset, whence) system call. */
23370
23371     register struct filp *rfilp;
23372     register off_t pos;
23373
23374     /* Check to see if the file descriptor is valid. */
23375     if ( (rfilp = get_filp(ls_fd)) == NIL_FILP) return(err_code);
23376
23377     /* No lseek on pipes. */
23378     if (rfilp->filp_ino->i_pipe == I_PIPE) return(ESPIPE);
23379
23380     /* The value of 'whence' determines the start position to use. */
23381     switch(whence) {
23382           case 0: pos = 0;        break;
23383           case 1: pos = rfilp->filp_pos;  break;
23384           case 2: pos = rfilp->filp_ino->i_size;  break;
23385           default: return(EINVAL);
23386     }
23387
23388     /* Check for overflow. */
23389     if (((long)offset > 0) && ((long)(pos + offset) < (long)pos)) return(EINVAL);
23390     if (((long)offset < 0) && ((long)(pos + offset) > (long)pos)) return(EINVAL);
23391     pos = pos + offset;
23392
23393     if (pos != rfilp->filp_pos)
23394           rfilp->filp_ino->i_seek = ISEEK;        /* inhibit read ahead */
23395     rfilp->filp_pos = pos;
23396     reply_l1 = pos;               /* insert the long into the output message */
23397     return(OK);
23398   }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                              src/fs/read.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

23400   /* This file contains the heart of the mechanism used to read (and write)
23401    * files.  Read and write requests are split up into chunks that do not cross
23402    * block boundaries.  Each chunk is then processed in turn.  Reads on special
23403    * files are also detected and handled.
23404    *
23405    * The entry points into this file are
23406    *   do_read:    perform the READ system call by calling read_write
23407    *   read_write: actually do the work of READ and WRITE
23408    *   read_map:   given an inode and file position, look up its zone number
23409    *   rd_indir:   read an entry in an indirect block
23410    *   read_ahead: manage the block read ahead business
23411    */
23412
23413   #include "fs.h"
23414   #include <fcntl.h>
23415   #include <minix/com.h>
23416   #include "buf.h"
23417   #include "file.h"
23418   #include "fproc.h"
23419   #include "inode.h"
23420   #include "param.h"
23421   #include "super.h"
23422
23423   #define FD_MASK          077    /* max file descriptor is 63 */
23424
23425   PRIVATE message umess;          /* message for asking SYSTASK for user copy */
23426
23427   FORWARD _PROTOTYPE( int rw_chunk, (struct inode *rip, off_t position,
23428                           unsigned off, int chunk, unsigned left, int rw_flag,
23429                           char *buff, int seg, int usr)                  );
23430
23431   /*===========================================================================*
23432    *                               do_read                                     *
23433    *===========================================================================*/
23434   PUBLIC int do_read()
23435   {
23436     return(read_write(READING));
23437   }


23440   /*===========================================================================*
23441    *                               read_write                                  *
23442    *===========================================================================*/
23443   PUBLIC int read_write(rw_flag)
23444   int rw_flag;                    /* READING or WRITING */
23445   {
23446   /* Perform read(fd, buffer, nbytes) or write(fd, buffer, nbytes) call. */
23447
23448     register struct inode *rip;
23449     register struct filp *f;
23450     off_t bytes_left, f_size, position;
23451     unsigned int off, cum_io;
23452     int op, oflags, r, chunk, usr, seg, block_spec, char_spec;
23453     int regular, partial_pipe = 0, partial_cnt = 0;
23454     dev_t dev;
```



```
23455     mode_t mode_word;
23456     struct filp *wf;
23457
23458     /* MM loads segments by putting funny things in upper 10 bits of 'fd'. */
23459     if (who == MM_PROC_NR && (fd & (~BYTE)) ) {
23460           usr = (fd >> 8) & BYTE;
23461           seg = (fd >> 6) & 03;
23462           fd &= FD_MASK;          /* get rid of user and segment bits */
23463     } else {
23464           usr = who;              /* normal case */
23465           seg = D;
23466     }
23467
23468     /* If the file descriptor is valid, get the inode, size and mode. */
23469     if (nbytes < 0) return(EINVAL);
23470     if ((f = get_filp(fd)) == NIL_FILP) return(err_code);
23471     if (((f->filp_mode) & (rw_flag == READING ? R_BIT : W_BIT)) == 0) {
23472           return(f->filp_mode == FILP_CLOSED ? EIO : EBADF);
23473     }
23474     if (nbytes == 0) return(0);   /* so char special files need not check for 0*/
23475     position = f->filp_pos;
23476     if (position > MAX_FILE_POS) return(EINVAL);
23477     if (position + nbytes < position) return(EINVAL); /* unsigned overflow */
23478     oflags = f->filp_flags;
23479     rip = f->filp_ino;
23480     f_size = rip->i_size;
23481     r = OK;
23482     if (rip->i_pipe == I_PIPE) {
23483           /* fp->fp_cum_io_partial is only nonzero when doing partial writes */
23484           cum_io = fp->fp_cum_io_partial;
23485     } else {
23486           cum_io = 0;
23487     }
23488     op = (rw_flag == READING ? DEV_READ : DEV_WRITE);
23489     mode_word = rip->i_mode & I_TYPE;
23490     regular = mode_word == I_REGULAR || mode_word == I_NAMED_PIPE;
23491
23492     char_spec = (mode_word == I_CHAR_SPECIAL ? 1 : 0);
23493     block_spec = (mode_word == I_BLOCK_SPECIAL ? 1 : 0);
23494     if (block_spec) f_size = LONG_MAX;
23495     rdwt_err = OK;                /* set to EIO if disk error occurs */
23496
23497     /* Check for character special files. */
23498     if (char_spec) {
23499           dev = (dev_t) rip->i_zone[0];
23500           r = dev_io(op, oflags & O_NONBLOCK, dev, position, nbytes, who,buffer);
23501           if (r >= 0) {
23502                   cum_io = r;
23503                   position += r;
23504                   r = OK;
23505           }
23506     } else {
23507           if (rw_flag == WRITING && block_spec == 0) {
23508                   /* Check in advance to see if file will grow too big. */
23509                   if (position > rip->i_sp->s_max_size - nbytes) return(EFBIG);
23510
23511                   /* Check for O_APPEND flag. */
23512                   if (oflags & O_APPEND) position = f_size;
23513
23514                   /* Clear the zone containing present EOF if hole about
```

```
23515                    * to be created.  This is necessary because all unwritten
23516                    * blocks prior to the EOF must read as zeros.
23517                    */
23518                   if (position > f_size) clear_zone(rip, f_size, 0);
23519           }
23520
23521           /* Pipes are a little different.  Check. */
23522           if (rip->i_pipe == I_PIPE) {
23523                  r = pipe_check(rip,rw_flag,oflags,nbytes,position,&partial_cnt);
23524                  if (r <= 0) return(r);
23525           }
23526
23527           if (partial_cnt > 0) partial_pipe = 1;
23528
23529           /* Split the transfer into chunks that don't span two blocks. */
23530           while (nbytes != 0) {
23531                   off = (unsigned int) (position % BLOCK_SIZE);/* offset in blk*/
23532                   if (partial_pipe) {  /* pipes only */
23533                           chunk = MIN(partial_cnt, BLOCK_SIZE - off);
23534                   } else
23535                           chunk = MIN(nbytes, BLOCK_SIZE - off);
23536                   if (chunk < 0) chunk = BLOCK_SIZE - off;
23537
23538                   if (rw_flag == READING) {
23539                           bytes_left = f_size - position;
23540                           if (position >= f_size) break;  /* we are beyond EOF */
23541                           if (chunk > bytes_left) chunk = (int) bytes_left;
23542                   }
23543
23544                   /* Read or write 'chunk' bytes. */
23545                   r = rw_chunk(rip, position, off, chunk, (unsigned) nbytes,
23546                                   rw_flag, buffer, seg, usr);
23547                   if (r != OK) break;     /* EOF reached */
23548                   if (rdwt_err < 0) break;
23549
23550                   /* Update counters and pointers. */
23551                   buffer += chunk;        /* user buffer address */
23552                   nbytes -= chunk;        /* bytes yet to be read */
23553                   cum_io += chunk;        /* bytes read so far */
23554                   position += chunk;      /* position within the file */
23555
23556                   if (partial_pipe) {
23557                           partial_cnt -= chunk;
23558                           if (partial_cnt <= 0)  break;
23559                   }
23560           }
23561   }
23562
23563   /* On write, update file size and access time. */
23564   if (rw_flag == WRITING) {
23565         if (regular || mode_word == I_DIRECTORY) {
23566                 if (position > f_size) rip->i_size = position;
23567         }
23568   } else {
23569         if (rip->i_pipe == I_PIPE && position >= rip->i_size) {
23570                 /* Reset pipe pointers. */
23571                 rip->i_size = 0;        /* no data left */
23572                 position = 0;           /* reset reader(s) */
23573                 if ( (wf = find_filp(rip, W_BIT)) != NIL_FILP) wf->filp_pos =0;
23574         }
```



```
23575   }
23576   f->filp_pos = position;
23577
23578   /* Check to see if read-ahead is called for, and if so, set it up. */
23579   if (rw_flag == READING && rip->i_seek == NO_SEEK && position % BLOCK_SIZE== 0
23580                   && (regular || mode_word == I_DIRECTORY)) {
23581         rdahed_inode = rip;
23582         rdahedpos = position;
23583   }
23584   rip->i_seek = NO_SEEK;
23585
23586   if (rdwt_err != OK) r = rdwt_err;     /* check for disk error */
23587   if (rdwt_err == END_OF_FILE) r = OK;
23588   if (r == OK) {
23589         if (rw_flag == READING) rip->i_update |= ATIME;
23590         if (rw_flag == WRITING) rip->i_update |= CTIME | MTIME;
23591         rip->i_dirt = DIRTY;            /* inode is thus now dirty */
23592         if (partial_pipe) {
23593                 partial_pipe = 0;
23594                         /* partial write on pipe with */
23595                         /* O_NONBLOCK, return write count */
23596                 if (!(oflags & O_NONBLOCK)) {
23597                         fp->fp_cum_io_partial = cum_io;
23598                         suspend(XPIPE); /* partial write on pipe with */
23599                         return(0);      /* nbyte > PIPE_SIZE - non-atomic */
23600                 }
23601         }
23602         fp->fp_cum_io_partial = 0;
23603         return(cum_io);
23604   } else {
23605         return(r);
23606   }
23607 }


23610 /*===========================================================================*
23611  *                                rw_chunk                                   *
23612  *===========================================================================*/
23613 PRIVATE int rw_chunk(rip, position, off, chunk, left, rw_flag, buff, seg, usr)
23614 register struct inode *rip;     /* pointer to inode for file to be rd/wr */
23615 off_t position;                 /* position within file to read or write */
23616 unsigned off;                   /* off within the current block */
23617 int chunk;                      /* number of bytes to read or write */
23618 unsigned left;                  /* max number of bytes wanted after position */
23619 int rw_flag;                    /* READING or WRITING */
23620 char *buff;                     /* virtual address of the user buffer */
23621 int seg;                        /* T or D segment in user space */
23622 int usr;                        /* which user process */
23623 {
23624 /* Read or write (part of) a block. */
23625
23626   register struct buf *bp;
23627   register int r;
23628   int n, block_spec;
23629   block_t b;
23630   dev_t dev;
23631
23632   block_spec = (rip->i_mode & I_TYPE) == I_BLOCK_SPECIAL;
23633   if (block_spec) {
23634         b = position/BLOCK_SIZE;
```

```
23635           dev = (dev_t) rip->i_zone[0];
23636       } else {
23637           b = read_map(rip, position);
23638           dev = rip->i_dev;
23639       }
23640
23641       if (!block_spec && b == NO_BLOCK) {
23642           if (rw_flag == READING) {
23643               /* Reading from a nonexistent block.  Must read as all zeros.*/
23644               bp = get_block(NO_DEV, NO_BLOCK, NORMAL);    /* get a buffer */
23645               zero_block(bp);
23646           } else {
23647               /* Writing to a nonexistent block. Create and enter in inode.*/
23648               if ((bp= new_block(rip, position)) == NIL_BUF)return(err_code);
23649           }
23650       } else if (rw_flag == READING) {
23651           /* Read and read ahead if convenient. */
23652           bp = rahead(rip, b, position, left);
23653       } else {
23654           /* Normally an existing block to be partially overwritten is first read
23655            * in.  However, a full block need not be read in.  If it is already in
23656            * the cache, acquire it, otherwise just acquire a free buffer.
23657            */
23658           n = (chunk == BLOCK_SIZE ? NO_READ : NORMAL);
23659           if (!block_spec && off == 0 && position >= rip->i_size) n = NO_READ;
23660           bp = get_block(dev, b, n);
23661       }
23662
23663       /* In all cases, bp now points to a valid buffer. */
23664       if (rw_flag == WRITING && chunk != BLOCK_SIZE && !block_spec &&
23665                                       position >= rip->i_size && off == 0) {
23666           zero_block(bp);
23667       }
23668       if (rw_flag == READING) {
23669           /* Copy a chunk from the block buffer to user space. */
23670           r = sys_copy(FS_PROC_NR, D, (phys_bytes) (bp->b_data+off),
23671                        usr, seg, (phys_bytes) buff,
23672                        (phys_bytes) chunk);
23673       } else {
23674           /* Copy a chunk from user space to the block buffer. */
23675           r = sys_copy(usr, seg, (phys_bytes) buff,
23676                        FS_PROC_NR, D, (phys_bytes) (bp->b_data+off),
23677                        (phys_bytes) chunk);
23678           bp->b_dirt = DIRTY;
23679       }
23680       n = (off + chunk == BLOCK_SIZE ? FULL_DATA_BLOCK : PARTIAL_DATA_BLOCK);
23681       put_block(bp, n);
23682       return(r);
23683   }


23686   /*===========================================================================*
23687    *                              read_map                                     *
23688    *===========================================================================*/
23689   PUBLIC block_t read_map(rip, position)
23690   register struct inode *rip;     /* ptr to inode to map from */
23691   off_t position;                 /* position in file whose blk wanted */
23692   {
23693   /* Given an inode and a position within the corresponding file, locate the
23694    * block (not zone) number in which that position is to be found and return it.
```



```
23695    */
23696
23697       register struct buf *bp;
23698       register zone_t z;
23699       int scale, boff, dzones, nr_indirects, index, zind, ex;
23700       block_t b;
23701       long excess, zone, block_pos;
23702
23703       scale = rip->i_sp->s_log_zone_size;   /* for block-zone conversion */
23704       block_pos = position/BLOCK_SIZE;      /* relative blk # in file */
23705       zone = block_pos >> scale;    /* position's zone */
23706       boff = (int) (block_pos - (zone << scale) ); /* relative blk # within zone */
23707       dzones = rip->i_ndzones;
23708       nr_indirects = rip->i_nindirs;
23709
23710       /* Is 'position' to be found in the inode itself? */
23711       if (zone < dzones) {
23712           zind = (int) zone;      /* index should be an int */
23713           z = rip->i_zone[zind];
23714           if (z == NO_ZONE) return(NO_BLOCK);
23715           b = ((block_t) z << scale) + boff;
23716           return(b);
23717       }
23718
23719       /* It is not in the inode, so it must be single or double indirect. */
23720       excess = zone - dzones;       /* first Vx_NR_DZONES don't count */
23721
23722       if (excess < nr_indirects) {
23723           /* 'position' can be located via the single indirect block. */
23724           z = rip->i_zone[dzones];
23725       } else {
23726           /* 'position' can be located via the double indirect block. */
23727           if ( (z = rip->i_zone[dzones+1]) == NO_ZONE) return(NO_BLOCK);
23728           excess -= nr_indirects;                 /* single indir doesn't count*/
23729           b = (block_t) z << scale;
23730           bp = get_block(rip->i_dev, b, NORMAL);  /* get double indirect block */
23731           index = (int) (excess/nr_indirects);
23732           z = rd_indir(bp, index);                /* z= zone for single*/
23733           put_block(bp, INDIRECT_BLOCK);          /* release double ind block */
23734           excess = excess % nr_indirects;         /* index into single ind blk */
23735       }
23736
23737       /* 'z' is zone num for single indirect block; 'excess' is index into it. */
23738       if (z == NO_ZONE) return(NO_BLOCK);
23739       b = (block_t) z << scale;                   /* b is blk # for single ind */
23740       bp = get_block(rip->i_dev, b, NORMAL);      /* get single indirect block */
23741       ex = (int) excess;                          /* need an integer */
23742       z = rd_indir(bp, ex);                       /* get block pointed to */
23743       put_block(bp, INDIRECT_BLOCK);              /* release single indir blk */
23744       if (z == NO_ZONE) return(NO_BLOCK);
23745       b = ((block_t) z << scale) + boff;
23746       return(b);
23747   }


23750   /*===========================================================================*
23751    *                              rd_indir                                     *
23752    *===========================================================================*/
23753   PUBLIC zone_t rd_indir(bp, index)
23754   struct buf *bp;                 /* pointer to indirect block */
```

```
23755   int index;                      /* index into *bp */
23756   {
23757   /* Given a pointer to an indirect block, read one entry.  The reason for
23758    * making a separate routine out of this is that there are four cases:
23759    * V1 (IBM and 68000), and V2 (IBM and 68000).
23760    */
23761
23762     struct super_block *sp;
23763     zone_t zone;                  /* V2 zones are longs (shorts in V1) */
23764
23765     sp = get_super(bp->b_dev);    /* need super block to find file sys type */
23766
23767     /* read a zone from an indirect block */
23768     if (sp->s_version == V1)
23769           zone = (zone_t) conv2(sp->s_native, (int)  bp->b_v1_ind[index]);
23770     else
23771           zone = (zone_t) conv4(sp->s_native, (long) bp->b_v2_ind[index]);
23772
23773     if (zone != NO_ZONE &&
23774                   (zone < (zone_t) sp->s_firstdatazone || zone >= sp->s_zones)) {
23775           printf("Illegal zone number %ld in indirect block, index %d\n",
23776                  (long) zone, index);
23777           panic("check file system", NO_NUM);
23778     }
23779     return(zone);
23780   }


23783   /*===========================================================================*
23784    *                               read_ahead                                  *
23785    *===========================================================================*/
23786   PUBLIC void read_ahead()
23787   {
23788   /* Read a block into the cache before it is needed. */
23789
23790     register struct inode *rip;
23791     struct buf *bp;
23792     block_t b;
23793
23794     rip = rdahed_inode;           /* pointer to inode to read ahead from */
23795     rdahed_inode = NIL_INODE;     /* turn off read ahead */
23796     if ( (b = read_map(rip, rdahedpos)) == NO_BLOCK) return;      /* at EOF */
23797     bp = rahead(rip, b, rdahedpos, BLOCK_SIZE);
23798     put_block(bp, PARTIAL_DATA_BLOCK);
23799   }


23802   /*===========================================================================*
23803    *                               rahead                                      *
23804    *===========================================================================*/
23805   PUBLIC struct buf *rahead(rip, baseblock, position, bytes_ahead)
23806   register struct inode *rip;     /* pointer to inode for file to be read */
23807   block_t baseblock;              /* block at current position */
23808   off_t position;                 /* position within file */
23809   unsigned bytes_ahead;           /* bytes beyond position for immediate use */
23810   {
23811   /* Fetch a block from the cache or the device.  If a physical read is
23812    * required, prefetch as many more blocks as convenient into the cache.
23813    * This usually covers bytes_ahead and is at least BLOCKS_MINIMUM.
23814    * The device driver may decide it knows better and stop reading at a
```



```
23815    * cylinder boundary (or after an error).  Rw_scattered() puts an optional
23816    * flag on all reads to allow this.
23817    */
23818
23819   /* Minimum number of blocks to prefetch. */
23820   # define BLOCKS_MINIMUM         (NR_BUFS < 50 ? 18 : 32)
23821
23822     int block_spec, scale, read_q_size;
23823     unsigned int blocks_ahead, fragment;
23824     block_t block, blocks_left;
23825     off_t ind1_pos;
23826     dev_t dev;
23827     struct buf *bp;
23828     static struct buf *read_q[NR_BUFS];
23829
23830     block_spec = (rip->i_mode & I_TYPE) == I_BLOCK_SPECIAL;
23831     if (block_spec) {
23832           dev = (dev_t) rip->i_zone[0];
23833     } else {
23834           dev = rip->i_dev;
23835     }
23836
23837     block = baseblock;
23838     bp = get_block(dev, block, PREFETCH);
23839     if (bp->b_dev != NO_DEV) return(bp);
23840
23841     /* The best guess for the number of blocks to prefetch:  A lot.
23842      * It is impossible to tell what the device looks like, so we don't even
23843      * try to guess the geometry, but leave it to the driver.
23844      *
23845      * The floppy driver can read a full track with no rotational delay, and it
23846      * avoids reading partial tracks if it can, so handing it enough buffers to
23847      * read two tracks is perfect.  (Two, because some diskette types have
23848      * an odd number of sectors per track, so a block may span tracks.)
23849      *
23850      * The disk drivers don't try to be smart.  With todays disks it is
23851      * impossible to tell what the real geometry looks like, so it is best to
23852      * read as much as you can.  With luck the caching on the drive allows
23853      * for a little time to start the next read.
23854      *
23855      * The current solution below is a bit of a hack, it just reads blocks from
23856      * the current file position hoping that more of the file can be found.  A
23857      * better solution must look at the already available zone pointers and
23858      * indirect blocks (but don't call read_map!).
23859      */
23860
23861     fragment = position % BLOCK_SIZE;
23862     position -= fragment;
23863     bytes_ahead += fragment;
23864
23865     blocks_ahead = (bytes_ahead + BLOCK_SIZE - 1) / BLOCK_SIZE;
23866
23867     if (block_spec && rip->i_size == 0) {
23868           blocks_left = NR_IOREQS;
23869     } else {
23870           blocks_left = (rip->i_size - position + BLOCK_SIZE - 1) / BLOCK_SIZE;
23871
23872           /* Go for the first indirect block if we are in its neighborhood. */
23873           if (!block_spec) {
23874                   scale = rip->i_sp->s_log_zone_size;
```

```
23875                   indl_pos = (off_t) rip->i_ndzones * (BLOCK_SIZE << scale);
23876                   if (position <= indl_pos && rip->i_size > indl_pos) {
23877                           blocks_ahead++;
23878                           blocks_left++;
23879                   }
23880           }
23881      }
23882
23883      /* No more than the maximum request. */
23884      if (blocks_ahead > NR_IOREQS) blocks_ahead = NR_IOREQS;
23885
23886      /* Read at least the minimum number of blocks, but not after a seek. */
23887      if (blocks_ahead < BLOCKS_MINIMUM && rip->i_seek == NO_SEEK)
23888           blocks_ahead = BLOCKS_MINIMUM;
23889
23890      /* Can't go past end of file. */
23891      if (blocks_ahead > blocks_left) blocks_ahead = blocks_left;
23892
23893      read_q_size = 0;
23894
23895      /* Acquire block buffers. */
23896      for (;;) {
23897           read_q[read_q_size++] = bp;
23898
23899           if (--blocks_ahead == 0) break;
23900
23901           /* Don't trash the cache, leave 4 free. */
23902           if (bufs_in_use >= NR_BUFS - 4) break;
23903
23904           block++;
23905
23906           bp = get_block(dev, block, PREFETCH);
23907           if (bp->b_dev != NO_DEV) {
23908                   /* Oops, block already in the cache, get out. */
23909                   put_block(bp, FULL_DATA_BLOCK);
23910                   break;
23911           }
23912      }
23913      rw_scattered(dev, read_q, read_q_size, READING);
23914      return(get_block(dev, baseblock, NORMAL));
23915 }
```

++

src/fs/write.c

++

```
24000 /* This file is the counterpart of "read.c".  It contains the code for writing
24001  * insofar as this is not contained in read_write().
24002  *
24003  * The entry points into this file are
24004  *   do_write:     call read_write to perform the WRITE system call
24005  *   clear_zone:   erase a zone in the middle of a file
24006  *   new_block:    acquire a new block
24007  */
24008
24009 #include "fs.h"
```



```
24010 #include <string.h>
24011 #include "buf.h"
24012 #include "file.h"
24013 #include "fproc.h"
24014 #include "inode.h"
24015 #include "super.h"
24016
24017 FORWARD _PROTOTYPE( int write_map, (struct inode *rip, off_t position,
24018                         zone_t new_zone)                                );
24019
24020 FORWARD _PROTOTYPE( void wr_indir, (struct buf *bp, int index, zone_t zone) );
24021
24022 /*===========================================================================*
24023  *                              do_write                                     *
24024  *===========================================================================*/
24025 PUBLIC int do_write()
24026 {
24027 /* Perform the write(fd, buffer, nbytes) system call. */
24028
24029   return(read_write(WRITING));
24030 }


24033 /*===========================================================================*
24034  *                              write_map                                    *
24035  *===========================================================================*/
24036 PRIVATE int write_map(rip, position, new_zone)
24037 register struct inode *rip;     /* pointer to inode to be changed */
24038 off_t position;                 /* file address to be mapped */
24039 zone_t new_zone;                /* zone # to be inserted */
24040 {
24041 /* Write a new zone into an inode. */
24042   int scale, ind_ex, new_ind, new_dbl, zones, nr_indirects, single, zindex, ex;
24043   zone_t z, z1;
24044   register block_t b;
24045   long excess, zone;
24046   struct buf *bp;
24047
24048   rip->i_dirt = DIRTY;          /* inode will be changed */
24049   bp = NIL_BUF;
24050   scale = rip->i_sp->s_log_zone_size;           /* for zone-block conversion */
24051   zone = (position/BLOCK_SIZE) >> scale;        /* relative zone # to insert */
24052   zones = rip->i_ndzones;       /* # direct zones in the inode */
24053   nr_indirects = rip->i_nindirs;/* # indirect zones per indirect block */
24054
24055   /* Is 'position' to be found in the inode itself? */
24056   if (zone < zones) {
24057         zindex = (int) zone;    /* we need an integer here */
24058         rip->i_zone[zindex] = new_zone;
24059         return(OK);
24060   }
24061
24062   /* It is not in the inode, so it must be single or double indirect. */
24063   excess = zone - zones;        /* first Vx_NR_DZONES don't count */
24064   new_ind = FALSE;
24065   new_dbl = FALSE;
24066
24067   if (excess < nr_indirects) {
24068         /* 'position' can be located via the single indirect block. */
24069         z1 = rip->i_zone[zones];        /* single indirect zone */
```

```
24070           single = TRUE;
24071     } else {
24072           /* 'position' can be located via the double indirect block. */
24073           if ( (z = rip->i_zone[zones+1]) == NO_ZONE) {
24074                   /* Create the double indirect block. */
24075                   if ( (z = alloc_zone(rip->i_dev, rip->i_zone[0])) == NO_ZONE)
24076                           return(err_code);
24077                   rip->i_zone[zones+1] = z;
24078                   new_dbl = TRUE; /* set flag for later */
24079           }
24080
24081           /* Either way, 'z' is zone number for double indirect block. */
24082           excess -= nr_indirects; /* single indirect doesn't count */
24083           ind_ex = (int) (excess / nr_indirects);
24084           excess = excess % nr_indirects;
24085           if (ind_ex >= nr_indirects) return(EFBIG);
24086           b = (block_t) z << scale;
24087           bp = get_block(rip->i_dev, b, (new_dbl ? NO_READ : NORMAL));
24088           if (new_dbl) zero_block(bp);
24089           z1 = rd_indir(bp, ind_ex);
24090           single = FALSE;
24091     }
24092
24093     /* z1 is now single indirect zone; 'excess' is index. */
24094     if (z1 == NO_ZONE) {
24095           /* Create indirect block and store zone # in inode or dbl indir blk. */
24096           z1 = alloc_zone(rip->i_dev, rip->i_zone[0]);
24097           if (single)
24098                   rip->i_zone[zones] = z1;        /* update inode */
24099           else
24100                   wr_indir(bp, ind_ex, z1);       /* update dbl indir */
24101
24102           new_ind = TRUE;
24103           if (bp != NIL_BUF) bp->b_dirt = DIRTY;  /* if double ind, it is dirty*/
24104           if (z1 == NO_ZONE) {
24105                   put_block(bp, INDIRECT_BLOCK);  /* release dbl indirect blk */
24106                   return(err_code);       /* couldn't create single ind */
24107           }
24108     }
24109     put_block(bp, INDIRECT_BLOCK);        /* release double indirect blk */
24110
24111     /* z1 is indirect block's zone number. */
24112     b = (block_t) z1 << scale;
24113     bp = get_block(rip->i_dev, b, (new_ind ? NO_READ : NORMAL) );
24114     if (new_ind) zero_block(bp);
24115     ex = (int) excess;                    /* we need an int here */
24116     wr_indir(bp, ex, new_zone);
24117     bp->b_dirt = DIRTY;
24118     put_block(bp, INDIRECT_BLOCK);
24119
24120     return(OK);
24121   }


24124   /*===========================================================================*
24125    *                              wr_indir                                     *
24126    *===========================================================================*/
24127   PRIVATE void wr_indir(bp, index, zone)
24128   struct buf *bp;                 /* pointer to indirect block */
24129   int index;                      /* index into *bp */
```



```
24130   zone_t zone;                    /* zone to write */
24131   {
24132   /* Given a pointer to an indirect block, write one entry. */
24133
24134     struct super_block *sp;
24135
24136     sp = get_super(bp->b_dev);    /* need super block to find file sys type */
24137
24138     /* write a zone into an indirect block */
24139     if (sp->s_version == V1)
24140           bp->b_v1_ind[index] = (zonel_t) conv2(sp->s_native, (int)  zone);
24141     else
24142           bp->b_v2_ind[index] = (zone_t)  conv4(sp->s_native, (long) zone);
24143   }


24146   /*===========================================================================*
24147    *                              clear_zone                                   *
24148    *===========================================================================*/
24149   PUBLIC void clear_zone(rip, pos, flag)
24150   register struct inode *rip;     /* inode to clear */
24151   off_t pos;                      /* points to block to clear */
24152   int flag;                       /* 0 if called by read_write, 1 by new_block */
24153   {
24154   /* Zero a zone, possibly starting in the middle.  The parameter 'pos' gives
24155    * a byte in the first block to be zeroed.  Clearzone() is called from
24156    * read_write and new_block().
24157    */
24158
24159     register struct buf *bp;
24160     register block_t b, blo, bhi;
24161     register off_t next;
24162     register int scale;
24163     register zone_t zone_size;
24164
24165     /* If the block size and zone size are the same, clear_zone() not needed. */
24166     scale = rip->i_sp->s_log_zone_size;
24167     if (scale == 0) return;
24168
24169     zone_size = (zone_t) BLOCK_SIZE << scale;
24170     if (flag == 1) pos = (pos/zone_size) * zone_size;
24171     next = pos + BLOCK_SIZE - 1;
24172
24173     /* If 'pos' is in the last block of a zone, do not clear the zone. */
24174     if (next/zone_size != pos/zone_size) return;
24175     if ( (blo = read_map(rip, next)) == NO_BLOCK) return;
24176     bhi = (  ((blo>>scale)+1) << scale)   - 1;
24177
24178     /* Clear all the blocks between 'blo' and 'bhi'. */
24179     for (b = blo; b <= bhi; b++) {
24180           bp = get_block(rip->i_dev, b, NO_READ);
24181           zero_block(bp);
24182           put_block(bp, FULL_DATA_BLOCK);
24183     }
24184   }


24187   /*===========================================================================*
24188    *                              new_block                                    *
24189    *===========================================================================*/
```

```
24190   PUBLIC struct buf *new_block(rip, position)
24191   register struct inode *rip;     /* pointer to inode */
24192   off_t position;                 /* file pointer */
24193   {
24194   /* Acquire a new block and return a pointer to it.  Doing so may require
24195    * allocating a complete zone, and then returning the initial block.
24196    * On the other hand, the current zone may still have some unused blocks.
24197    */
24198
24199     register struct buf *bp;
24200     block_t b, base_block;
24201     zone_t z;
24202     zone_t zone_size;
24203     int scale, r;
24204     struct super_block *sp;
24205
24206     /* Is another block available in the current zone? */
24207     if ( (b = read_map(rip, position)) == NO_BLOCK) {
24208           /* Choose first zone if possible. */
24209           /* Lose if the file is nonempty but the first zone number is NO_ZONE
24210            * corresponding to a zone full of zeros.  It would be better to
24211            * search near the last real zone.
24212            */
24213           if (rip->i_zone[0] == NO_ZONE) {
24214                   sp = rip->i_sp;
24215                   z = sp->s_firstdatazone;
24216           } else {
24217                   z = rip->i_zone[0];     /* hunt near first zone */
24218           }
24219           if ( (z = alloc_zone(rip->i_dev, z)) == NO_ZONE) return(NIL_BUF);
24220           if ( (r = write_map(rip, position, z)) != OK) {
24221                   free_zone(rip->i_dev, z);
24222                   err_code = r;
24223                   return(NIL_BUF);
24224           }
24225
24226           /* If we are not writing at EOF, clear the zone, just to be safe. */
24227           if ( position != rip->i_size) clear_zone(rip, position, 1);
24228           scale = rip->i_sp->s_log_zone_size;
24229           base_block = (block_t) z << scale;
24230           zone_size = (zone_t) BLOCK_SIZE << scale;
24231           b = base_block + (block_t)((position % zone_size)/BLOCK_SIZE);
24232     }
24233
24234     bp = get_block(rip->i_dev, b, NO_READ);
24235     zero_block(bp);
24236     return(bp);
24237   }


24240   /*===========================================================================*
24241    *                              zero_block                                   *
24242    *===========================================================================*/
24243   PUBLIC void zero_block(bp)
24244   register struct buf *bp;        /* pointer to buffer to zero */
24245   {
24246   /* Zero a block. */
24247
24248     memset(bp->b_data, 0, BLOCK_SIZE);
24249     bp->b_dirt = DIRTY;
```



```
24250   }
```

++
src/fs/pipe.c
++

```
24300   /* This file deals with the suspension and revival of processes.  A process can
24301    * be suspended because it wants to read or write from a pipe and can't, or
24302    * because it wants to read or write from a special file and can't.  When a
24303    * process can't continue it is suspended, and revived later when it is able
24304    * to continue.
24305    *
24306    * The entry points into this file are
24307    *   do_pipe:     perform the PIPE system call
24308    *   pipe_check:  check to see that a read or write on a pipe is feasible now
24309    *   suspend:     suspend a process that cannot do a requested read or write
24310    *   release:     check to see if a suspended process can be released and do it
24311    *   revive:      mark a suspended process as able to run again
24312    *   do_unpause:  a signal has been sent to a process; see if it suspended
24313    */
24314
24315   #include "fs.h"
24316   #include <fcntl.h>
24317   #include <signal.h>
24318   #include <minix/boot.h>
24319   #include <minix/callnr.h>
24320   #include <minix/com.h>
24321   #include "dev.h"
24322   #include "file.h"
24323   #include "fproc.h"
24324   #include "inode.h"
24325   #include "param.h"
24326
24327   PRIVATE message mess;
24328
24329   /*===========================================================================*
24330    *                                do_pipe                                    *
24331    *===========================================================================*/
24332   PUBLIC int do_pipe()
24333   {
24334   /* Perform the pipe(fil_des) system call. */
24335
24336     register struct fproc *rfp;
24337     register struct inode *rip;
24338     int r;
24339     struct filp *fil_ptr0, *fil_ptr1;
24340     int fil_des[2];               /* reply goes here */
24341
24342     /* Acquire two file descriptors. */
24343     rfp = fp;
24344     if ( (r = get_fd(0, R_BIT, &fil_des[0], &fil_ptr0)) != OK) return(r);
24345     rfp->fp_filp[fil_des[0]] = fil_ptr0;
24346     fil_ptr0->filp_count = 1;
24347     if ( (r = get_fd(0, W_BIT, &fil_des[1], &fil_ptr1)) != OK) {
24348           rfp->fp_filp[fil_des[0]] = NIL_FILP;
24349           fil_ptr0->filp_count = 0;
```

```
24350           return(r);
24351       }
24352       rfp->fp_filp[fil_des[1]] = fil_ptr1;
24353       fil_ptr1->filp_count = 1;
24354
24355       /* Make the inode on the pipe device. */
24356       if ( (rip = alloc_inode(PIPE_DEV, I_REGULAR) ) == NIL_INODE) {
24357             rfp->fp_filp[fil_des[0]] = NIL_FILP;
24358             fil_ptr0->filp_count = 0;
24359             rfp->fp_filp[fil_des[1]] = NIL_FILP;
24360             fil_ptr1->filp_count = 0;
24361             return(err_code);
24362       }
24363
24364       if (read_only(rip) != OK) panic("pipe device is read only", NO_NUM);
24365
24366       rip->i_pipe = I_PIPE;
24367       rip->i_mode &= ~I_REGULAR;
24368       rip->i_mode |= I_NAMED_PIPE;  /* pipes and FIFOs have this bit set */
24369       fil_ptr0->filp_ino = rip;
24370       fil_ptr0->filp_flags = O_RDONLY;
24371       dup_inode(rip);               /* for double usage */
24372       fil_ptr1->filp_ino = rip;
24373       fil_ptr1->filp_flags = O_WRONLY;
24374       rw_inode(rip, WRITING);       /* mark inode as allocated */
24375       reply_i1 = fil_des[0];
24376       reply_i2 = fil_des[1];
24377       rip->i_update = ATIME | CTIME | MTIME;
24378       return(OK);
24379   }


24382   /*===========================================================================*
24383    *                              pipe_check                                   *
24384    *===========================================================================*/
24385   PUBLIC int pipe_check(rip, rw_flag, oflags, bytes, position, canwrite)
24386   register struct inode *rip;     /* the inode of the pipe */
24387   int rw_flag;                    /* READING or WRITING */
24388   int oflags;                     /* flags set by open or fcntl */
24389   register int bytes;             /* bytes to be read or written (all chunks) */
24390   register off_t position;        /* current file position */
24391   int *canwrite;                  /* return: number of bytes we can write */
24392   {
24393   /* Pipes are a little different.  If a process reads from an empty pipe for
24394    * which a writer still exists, suspend the reader.  If the pipe is empty
24395    * and there is no writer, return 0 bytes.  If a process is writing to a
24396    * pipe and no one is reading from it, give a broken pipe error.
24397    */
24398
24399     int r = 0;
24400
24401     /* If reading, check for empty pipe. */
24402     if (rw_flag == READING) {
24403           if (position >= rip->i_size) {
24404                 /* Process is reading from an empty pipe. */
24405                 if (find_filp(rip, W_BIT) != NIL_FILP) {
24406                       /* Writer exists */
24407                       if (oflags & O_NONBLOCK)
24408                             r = EAGAIN;
24409                       else
```



```
24410                             suspend(XPIPE); /* block reader */
24411
24412                       /* If need be, activate sleeping writers. */
24413                       if (susp_count > 0) release(rip, WRITE, susp_count);
24414                 }
24415                 return(r);
24416           }
24417       } else {
24418           /* Process is writing to a pipe. */
24419   /*      if (bytes > PIPE_SIZE) return(EFBIG); */
24420           if (find_filp(rip, R_BIT) == NIL_FILP) {
24421                 /* Tell kernel to generate a SIGPIPE signal. */
24422                 sys_kill((int)(fp - fproc), SIGPIPE);
24423                 return(EPIPE);
24424           }
24425
24426           if (position + bytes > PIPE_SIZE) {
24427                 if ((oflags & O_NONBLOCK) && bytes < PIPE_SIZE)
24428                       return(EAGAIN);
24429                 else if ((oflags & O_NONBLOCK) && bytes > PIPE_SIZE) {
24430                       if ( (*canwrite = (PIPE_SIZE - position)) > 0)  {
24431                             /* Do a partial write. Need to wakeup reader */
24432                             release(rip, READ, susp_count);
24433                             return(1);
24434                       } else {
24435                             return(EAGAIN);
24436                       }
24437                 }
24438                 if (bytes > PIPE_SIZE) {
24439                       if ((*canwrite = PIPE_SIZE - position) > 0) {
24440                             /* Do a partial write. Need to wakeup reader
24441                              * since we'll suspend ourself in read_write()
24442                              */
24443                             release(rip, READ, susp_count);
24444                             return(1);
24445                       }
24446                 }
24447                 suspend(XPIPE); /* stop writer -- pipe full */
24448                 return(0);
24449           }
24450
24451           /* Writing to an empty pipe.  Search for suspended reader. */
24452           if (position == 0) release(rip, READ, susp_count);
24453       }
24454
24455       *canwrite = 0;
24456       return(1);
24457   }


24460   /*===========================================================================*
24461    *                              suspend                                      *
24462    *===========================================================================*/
24463   PUBLIC void suspend(task)
24464   int task;                       /* who is proc waiting for? (PIPE = pipe) */
24465   {
24466   /* Take measures to suspend the processing of the present system call.
24467    * Store the parameters to be used upon resuming in the process table.
24468    * (Actually they are not used when a process is waiting for an I/O device,
24469    * but they are needed for pipes, and it is not worth making the distinction.)
```

```
24470     */
24471
24472     if (task == XPIPE || task == XPOPEN) susp_count++;/* #procs susp'ed on pipe*/
24473     fp->fp_suspended = SUSPENDED;
24474     fp->fp_fd = fd << 8 | fs_call;
24475     fp->fp_task = -task;
24476     if (task == XLOCK) {
24477           fp->fp_buffer = (char *) name1; /*  third arg to fcntl() */
24478           fp->fp_nbytes =request;         /* second arg to fcntl() */
24479     } else {
24480           fp->fp_buffer = buffer;         /* for reads and writes */
24481           fp->fp_nbytes = nbytes;
24482     }
24483     dont_reply = TRUE;            /* do not send caller a reply message now */
24484   }


24487   /*===========================================================================*
24488    *                               release                                     *
24489    *===========================================================================*/
24490   PUBLIC void release(ip, call_nr, count)
24491   register struct inode *ip;      /* inode of pipe */
24492   int call_nr;                    /* READ, WRITE, OPEN or CREAT */
24493   int count;                      /* max number of processes to release */
24494   {
24495   /* Check to see if any process is hanging on the pipe whose inode is in 'ip'.
24496    * If one is, and it was trying to perform the call indicated by 'call_nr',
24497    * release it.
24498    */
24499
24500     register struct fproc *rp;
24501
24502     /* Search the proc table. */
24503     for (rp = &fproc[0]; rp < &fproc[NR_PROCS]; rp++) {
24504           if (rp->fp_suspended == SUSPENDED &&
24505                           rp->fp_revived == NOT_REVIVING &&
24506                           (rp->fp_fd & BYTE) == call_nr &&
24507                           rp->fp_filp[rp->fp_fd>>8]->filp_ino == ip) {
24508                   revive((int)(rp - fproc), 0);
24509                   susp_count--;   /* keep track of who is suspended */
24510                   if (--count == 0) return;
24511           }
24512     }
24513   }


24516   /*===========================================================================*
24517    *                               revive                                      *
24518    *===========================================================================*/
24519   PUBLIC void revive(proc_nr, bytes)
24520   int proc_nr;                    /* process to revive */
24521   int bytes;                      /* if hanging on task, how many bytes read */
24522   {
24523   /* Revive a previously blocked process. When a process hangs on tty, this
24524    * is the way it is eventually released.
24525    */
24526
24527     register struct fproc *rfp;
24528     register int task;
24529
```



```
24530     if (proc_nr < 0 || proc_nr >= NR_PROCS) panic("revive err", proc_nr);
24531     rfp = &fproc[proc_nr];
24532     if (rfp->fp_suspended == NOT_SUSPENDED || rfp->fp_revived == REVIVING)return;
24533
24534     /* The 'reviving' flag only applies to pipes.  Processes waiting for TTY get
24535      * a message right away.  The revival process is different for TTY and pipes.
24536      * For TTY revival, the work is already done, for pipes it is not: the proc
24537      * must be restarted so it can try again.
24538      */
24539     task = -rfp->fp_task;
24540     if (task == XPIPE || task == XLOCK) {
24541           /* Revive a process suspended on a pipe or lock. */
24542           rfp->fp_revived = REVIVING;
24543           reviving++;             /* process was waiting on pipe or lock */
24544     } else {
24545           rfp->fp_suspended = NOT_SUSPENDED;
24546           if (task == XPOPEN) /* process blocked in open or create */
24547                   reply(proc_nr, rfp->fp_fd>>8);
24548           else {
24549                   /* Revive a process suspended on TTY or other device. */
24550                   rfp->fp_nbytes = bytes; /*pretend it wants only what there is*/
24551                   reply(proc_nr, bytes);  /* unblock the process */
24552           }
24553     }
24554   }


24557   /*===========================================================================*
24558    *                               do_unpause                                  *
24559    *===========================================================================*/
24560   PUBLIC int do_unpause()
24561   {
24562   /* A signal has been sent to a user who is paused on the file system.
24563    * Abort the system call with the EINTR error message.
24564    */
24565
24566     register struct fproc *rfp;
24567     int proc_nr, task, fild;
24568     struct filp *f;
24569     dev_t dev;
24570
24571     if (who > MM_PROC_NR) return(EPERM);
24572     proc_nr = pro;
24573     if (proc_nr < 0 || proc_nr >= NR_PROCS) panic("unpause err 1", proc_nr);
24574     rfp = &fproc[proc_nr];
24575     if (rfp->fp_suspended == NOT_SUSPENDED) return(OK);
24576     task = -rfp->fp_task;
24577
24578     switch(task) {
24579           case XPIPE:             /* process trying to read or write a pipe */
24580                   break;
24581
24582           case XOPEN:             /* process trying to open a special file */
24583                   panic ("fs/do_unpause called with XOPEN\n", NO_NUM);
24584
24585           case XLOCK:             /* process trying to set a lock with FCNTL */
24586                   break;
24587
24588           case XPOPEN:            /* process trying to open a fifo */
24589                   break;
```

```
24590
24591           default:                /* process trying to do device I/O (e.g. tty)*/
24592                   fild = (rfp->fp_fd >> 8) & BYTE;/* extract file descriptor */
24593                   if (fild < 0 || fild >= OPEN_MAX)panic("unpause err 2",NO_NUM);
24594                   f = rfp->fp_filp[fild];
24595                   dev = (dev_t) f->filp_ino->i_zone[0];   /* device hung on */
24596                   mess.TTY_LINE = (dev >> MINOR) & BYTE;
24597                   mess.PROC_NR = proc_nr;
24598
24599                   /* Tell kernel R or W. Mode is from current call, not open. */
24600                   mess.COUNT = (rfp->fp_fd & BYTE) == READ ? R_BIT : W_BIT;
24601                   mess.m_type = CANCEL;
24602                   fp = rfp;       /* hack - call_ctty uses fp */
24603                   (*dmap[(dev >> MAJOR) & BYTE].dmap_rw)(task, &mess);
24604       }
24605
24606       rfp->fp_suspended = NOT_SUSPENDED;
24607       reply(proc_nr, EINTR);        /* signal interrupted call */
24608       return(OK);
24609   }
```

++
src/fs/path.c
++

```
24700   /* This file contains the procedures that look up path names in the directory
24701    * system and determine the inode number that goes with a given path name.
24702    *
24703    *  The entry points into this file are
24704    *   eat_path:   the 'main' routine of the path-to-inode conversion mechanism
24705    *   last_dir:   find the final directory on a given path
24706    *   advance:    parse one component of a path name
24707    *   search_dir: search a directory for a string and return its inode number
24708    */
24709
24710   #include "fs.h"
24711   #include <string.h>
24712   #include <minix/callnr.h>
24713   #include "buf.h"
24714   #include "file.h"
24715   #include "fproc.h"
24716   #include "inode.h"
24717   #include "super.h"
24718
24719   PUBLIC char dot1[2] = ".";      /* used for search_dir to bypass the access */
24720   PUBLIC char dot2[3] = "..";     /* permissions for . and ..                 */
24721
24722   FORWARD _PROTOTYPE( char *get_name, (char *old_name, char string [NAME_MAX]) );
24723
24724   /*===========================================================================*
24725    *                               eat_path                                    *
24726    *===========================================================================*/
24727   PUBLIC struct inode *eat_path(path)
24728   char *path;                     /* the path name to be parsed */
24729   {
```



```
24730   /* Parse the path 'path' and put its inode in the inode table. If not possible,
24731    * return NIL_INODE as function value and an error code in 'err_code'.
24732    */
24733
24734     register struct inode *ldip, *rip;
24735     char string[NAME_MAX];        /* hold 1 path component name here */
24736
24737     /* First open the path down to the final directory. */
24738     if ( (ldip = last_dir(path, string)) == NIL_INODE)
24739           return(NIL_INODE);      /* we couldn't open final directory */
24740
24741     /* The path consisting only of "/" is a special case, check for it. */
24742     if (string[0] == '\0') return(ldip);
24743
24744     /* Get final component of the path. */
24745     rip = advance(ldip, string);
24746     put_inode(ldip);
24747     return(rip);
24748   }


24751   /*===========================================================================*
24752    *                               last_dir                                    *
24753    *===========================================================================*/
24754   PUBLIC struct inode *last_dir(path, string)
24755   char *path;                     /* the path name to be parsed */
24756   char string[NAME_MAX];          /* the final component is returned here */
24757   {
24758   /* Given a path, 'path', located in the fs address space, parse it as
24759    * far as the last directory, fetch the inode for the last directory into
24760    * the inode table, and return a pointer to the inode.  In
24761    * addition, return the final component of the path in 'string'.
24762    * If the last directory can't be opened, return NIL_INODE and
24763    * the reason for failure in 'err_code'.
24764    */
24765
24766     register struct inode *rip;
24767     register char *new_name;
24768     register struct inode *new_ip;
24769
24770     /* Is the path absolute or relative?  Initialize 'rip' accordingly. */
24771     rip = (*path == '/' ? fp->fp_rootdir : fp->fp_workdir);
24772
24773     /* If dir has been removed or path is empty, return ENOENT. */
24774     if (rip->i_nlinks == 0 || *path == '\0') {
24775           err_code = ENOENT;
24776           return(NIL_INODE);
24777     }
24778
24779     dup_inode(rip);               /* inode will be returned with put_inode */
24780
24781     /* Scan the path component by component. */
24782     while (TRUE) {
24783           /* Extract one component. */
24784           if ( (new_name = get_name(path, string)) == (char*) 0) {
24785                   put_inode(rip); /* bad path in user space */
24786                   return(NIL_INODE);
24787           }
24788           if (*new_name == '\0')
24789                   if ( (rip->i_mode & I_TYPE) == I_DIRECTORY)
```

```
24790                           return(rip);    /* normal exit */
24791                   else {
24792                           /* last file of path prefix is not a directory */
24793                           put_inode(rip);
24794                           err_code = ENOTDIR;
24795                           return(NIL_INODE);
24796                   }
24797
24798           /* There is more path.  Keep parsing. */
24799           new_ip = advance(rip, string);
24800           put_inode(rip);         /* rip either obsolete or irrelevant */
24801           if (new_ip == NIL_INODE) return(NIL_INODE);
24802
24803           /* The call to advance() succeeded.  Fetch next component. */
24804           path = new_name;
24805           rip = new_ip;
24806     }
24807   }


24810   /*===========================================================================*
24811    *                              get_name                                     *
24812    *===========================================================================*/
24813   PRIVATE char *get_name(old_name, string)
24814   char *old_name;               /* path name to parse */
24815   char string[NAME_MAX];        /* component extracted from 'old_name' */
24816   {
24817   /* Given a pointer to a path name in fs space, 'old_name', copy the next
24818    * component to 'string' and pad with zeros.  A pointer to that part of
24819    * the name as yet unparsed is returned.  Roughly speaking,
24820    * 'get_name' = 'old_name' - 'string'.
24821    *
24822    * This routine follows the standard convention that /usr/ast, /usr//ast,
24823    * //usr///ast and /usr/ast/ are all equivalent.
24824    */
24825
24826     register int c;
24827     register char *np, *rnp;
24828
24829     np = string;                /* 'np' points to current position */
24830     rnp = old_name;             /* 'rnp' points to unparsed string */
24831     while ( (c = *rnp) == '/') rnp++;     /* skip leading slashes */
24832
24833     /* Copy the unparsed path, 'old_name', to the array, 'string'. */
24834     while ( rnp < &old_name[PATH_MAX]  &&  c != '/'   &&  c != '\0') {
24835           if (np < &string[NAME_MAX]) *np++ = c;
24836           c = *++rnp;           /* advance to next character */
24837     }
24838
24839     /* To make /usr/ast/ equivalent to /usr/ast, skip trailing slashes. */
24840     while (c == '/' && rnp < &old_name[PATH_MAX]) c = *++rnp;
24841
24842     if (np < &string[NAME_MAX]) *np = '\0';       /* Terminate string */
24843
24844     if (rnp >= &old_name[PATH_MAX]) {
24845           err_code = ENAMETOOLONG;
24846           return((char *) 0);
24847     }
24848     return(rnp);
24849   }
```



```
24852   /*===========================================================================*
24853    *                              advance                                      *
24854    *===========================================================================*/
24855   PUBLIC struct inode *advance(dirp, string)
24856   struct inode *dirp;           /* inode for directory to be searched */
24857   char string[NAME_MAX];        /* component name to look for */
24858   {
24859   /* Given a directory and a component of a path, look up the component in
24860    * the directory, find the inode, open it, and return a pointer to its inode
24861    * slot.  If it can't be done, return NIL_INODE.
24862    */
24863
24864     register struct inode *rip;
24865     struct inode *rip2;
24866     register struct super_block *sp;
24867     int r, inumb;
24868     dev_t mnt_dev;
24869     ino_t numb;
24870
24871     /* If 'string' is empty, yield same inode straight away. */
24872     if (string[0] == '\0') return(get_inode(dirp->i_dev, (int) dirp->i_num));
24873
24874     /* Check for NIL_INODE. */
24875     if (dirp == NIL_INODE) return(NIL_INODE);
24876
24877     /* If 'string' is not present in the directory, signal error. */
24878     if ( (r = search_dir(dirp, string, &numb, LOOK_UP)) != OK) {
24879           err_code = r;
24880           return(NIL_INODE);
24881     }
24882
24883     /* Don't go beyond the current root directory, unless the string is dot2. */
24884     if (dirp == fp->fp_rootdir && strcmp(string, "..") == 0 && string != dot2)
24885                   return(get_inode(dirp->i_dev, (int) dirp->i_num));
24886
24887     /* The component has been found in the directory.  Get inode. */
24888     if ( (rip = get_inode(dirp->i_dev, (int) numb)) == NIL_INODE)
24889           return(NIL_INODE);
24890
24891     if (rip->i_num == ROOT_INODE)
24892           if (dirp->i_num == ROOT_INODE) {
24893               if (string[1] == '.') {
24894                   for (sp = &super_block[1]; sp < &super_block[NR_SUPERS]; sp++){
24895                           if (sp->s_dev == rip->i_dev) {
24896                                   /* Release the root inode.  Replace by the
24897                                    * inode mounted on.
24898                                    */
24899                                   put_inode(rip);
24900                                   mnt_dev = sp->s_imount->i_dev;
24901                                   inumb = (int) sp->s_imount->i_num;
24902                                   rip2 = get_inode(mnt_dev, inumb);
24903                                   rip = advance(rip2, string);
24904                                   put_inode(rip2);
24905                                   break;
24906                           }
24907                   }
24908               }
24909           }
```

```
24910     if (rip == NIL_INODE) return(NIL_INODE);
24911
24912     /* See if the inode is mounted on.  If so, switch to root directory of the
24913      * mounted file system.  The super_block provides the linkage between the
24914      * inode mounted on and the root directory of the mounted file system.
24915      */
24916     while (rip != NIL_INODE && rip->i_mount == I_MOUNT) {
24917           /* The inode is indeed mounted on. */
24918           for (sp = &super_block[0]; sp < &super_block[NR_SUPERS]; sp++) {
24919                 if (sp->s_imount == rip) {
24920                       /* Release the inode mounted on.  Replace by the
24921                        * inode of the root inode of the mounted device.
24922                        */
24923                       put_inode(rip);
24924                       rip = get_inode(sp->s_dev, ROOT_INODE);
24925                       break;
24926                 }
24927           }
24928     }
24929     return(rip);          /* return pointer to inode's component */
24930 }


24933 /*===========================================================================*
24934  *                              search_dir                                   *
24935  *===========================================================================*/
24936 PUBLIC int search_dir(ldir_ptr, string, numb, flag)
24937 register struct inode *ldir_ptr;        /* ptr to inode for dir to search */
24938 char string[NAME_MAX];          /* component to search for */
24939 ino_t *numb;                    /* pointer to inode number */
24940 int flag;                       /* LOOK_UP, ENTER, DELETE or IS_EMPTY */
24941 {
24942 /* This function searches the directory whose inode is pointed to by 'ldip':
24943  * if (flag == ENTER)  enter 'string' in the directory with inode # '*numb';
24944  * if (flag == DELETE) delete 'string' from the directory;
24945  * if (flag == LOOK_UP) search for 'string' and return inode # in 'numb';
24946  * if (flag == IS_EMPTY) return OK if only . and .. in dir else ENOTEMPTY;
24947  *
24948  *    if 'string' is dot1 or dot2, no access permissions are checked.
24949  */
24950
24951   register struct direct *dp;
24952   register struct buf *bp;
24953   int i, r, e_hit, t, match;
24954   mode_t bits;
24955   off_t pos;
24956   unsigned new_slots, old_slots;
24957   block_t b;
24958   struct super_block *sp;
24959   int extended = 0;
24960
24961   /* If 'ldir_ptr' is not a pointer to a dir inode, error. */
24962   if ( (ldir_ptr->i_mode & I_TYPE) != I_DIRECTORY) return(ENOTDIR);
24963
24964   r = OK;
24965
24966   if (flag != IS_EMPTY) {
24967         bits = (flag == LOOK_UP ? X_BIT : W_BIT | X_BIT);
24968
24969         if (string == dot1 || string == dot2) {
```



```
24970               if (flag != LOOK_UP) r = read_only(ldir_ptr);
24971                                   /* only a writable device is required. */
24972         }
24973         else r = forbidden(ldir_ptr, bits); /* check access permissions */
24974     }
24975     if (r != OK) return(r);
24976
24977     /* Step through the directory one block at a time. */
24978     old_slots = (unsigned) (ldir_ptr->i_size/DIR_ENTRY_SIZE);
24979     new_slots = 0;
24980     e_hit = FALSE;
24981     match = 0;                    /* set when a string match occurs */
24982
24983     for (pos = 0; pos < ldir_ptr->i_size; pos += BLOCK_SIZE) {
24984           b = read_map(ldir_ptr, pos);    /* get block number */
24985
24986           /* Since directories don't have holes, 'b' cannot be NO_BLOCK. */
24987           bp = get_block(ldir_ptr->i_dev, b, NORMAL);     /* get a dir block */
24988
24989           /* Search a directory block. */
24990           for (dp = &bp->b_dir[0]; dp < &bp->b_dir[NR_DIR_ENTRIES]; dp++) {
24991                 if (++new_slots > old_slots) { /* not found, but room left */
24992                       if (flag == ENTER) e_hit = TRUE;
24993                       break;
24994                 }
24995
24996                 /* Match occurs if string found. */
24997                 if (flag != ENTER && dp->d_ino != 0) {
24998                       if (flag == IS_EMPTY) {
24999                             /* If this test succeeds, dir is not empty. */
25000                             if (strcmp(dp->d_name, "." ) != 0 &&
25001                                 strcmp(dp->d_name, "..") != 0) match = 1;
25002                       } else {
25003                             if (strncmp(dp->d_name, string, NAME_MAX) == 0)
25004                                   match = 1;
25005                       }
25006                 }
25007
25008                 if (match) {
25009                       /* LOOK_UP or DELETE found what it wanted. */
25010                       r = OK;
25011                       if (flag == IS_EMPTY) r = ENOTEMPTY;
25012                       else if (flag == DELETE) {
25013                             /* Save d_ino for recovery. */
25014                             t = NAME_MAX - sizeof(ino_t);
25015                             *((ino_t *) &dp->d_name[t]) = dp->d_ino;
25016                             dp->d_ino = 0;  /* erase entry */
25017                             bp->b_dirt = DIRTY;
25018                             ldir_ptr->i_update |= CTIME | MTIME;
25019                             ldir_ptr->i_dirt = DIRTY;
25020                       } else {
25021                             sp = ldir_ptr->i_sp;    /* 'flag' is LOOK_UP */
25022                             *numb = conv2(sp->s_native, (int) dp->d_ino);
25023                       }
25024                       put_block(bp, DIRECTORY_BLOCK);
25025                       return(r);
25026                 }
25027
25028
25029                 /* Check for free slot for the benefit of ENTER. */
```

```
25030                   if (flag == ENTER && dp->d_ino == 0) {
25031                           e_hit = TRUE;   /* we found a free slot */
25032                           break;
25033                   }
25034           }
25035
25036           /* The whole block has been searched or ENTER has a free slot. */
25037           if (e_hit) break;       /* e_hit set if ENTER can be performed now */
25038           put_block(bp, DIRECTORY_BLOCK); /* otherwise, continue searching dir */
25039      }
25040
25041      /* The whole directory has now been searched. */
25042      if (flag != ENTER) return(flag == IS_EMPTY ? OK : ENOENT);
25043
25044      /* This call is for ENTER.  If no free slot has been found so far, try to
25045       * extend directory.
25046       */
25047      if (e_hit == FALSE) { /* directory is full and no room left in last block */
25048           new_slots++;            /* increase directory size by 1 entry */
25049           if (new_slots == 0) return(EFBIG); /* dir size limited by slot count */
25050           if ( (bp = new_block(ldir_ptr, ldir_ptr->i_size)) == NIL_BUF)
25051                   return(err_code);
25052           dp = &bp->b_dir[0];
25053           extended = 1;
25054      }
25055
25056      /* 'bp' now points to a directory block with space. 'dp' points to slot. */
25057      (void) memset(dp->d_name, 0, (size_t) NAME_MAX); /* clear entry */
25058      for (i = 0; string[i] && i < NAME_MAX; i++) dp->d_name[i] = string[i];
25059      sp = ldir_ptr->i_sp;
25060      dp->d_ino = conv2(sp->s_native, (int) *numb);
25061      bp->b_dirt = DIRTY;
25062      put_block(bp, DIRECTORY_BLOCK);
25063      ldir_ptr->i_update |= CTIME | MTIME;  /* mark mtime for update later */
25064      ldir_ptr->i_dirt = DIRTY;
25065      if (new_slots > old_slots) {
25066           ldir_ptr->i_size = (off_t) new_slots * DIR_ENTRY_SIZE;
25067           /* Send the change to disk if the directory is extended. */
25068           if (extended) rw_inode(ldir_ptr, WRITING);
25069      }
25070      return(OK);
25071 }
```

++

src/fs/mount.c

++

```
25100 /* This file performs the MOUNT and UMOUNT system calls.
25101  *
25102  * The entry points into this file are
25103  *   do_mount:  perform the MOUNT system call
25104  *   do_umount: perform the UMOUNT system call
25105  */
25106
25107 #include "fs.h"
25108 #include <fcntl.h>
25109 #include <minix/com.h>
```



```
25110 #include <sys/stat.h>
25111 #include "buf.h"
25112 #include "dev.h"
25113 #include "file.h"
25114 #include "fproc.h"
25115 #include "inode.h"
25116 #include "param.h"
25117 #include "super.h"
25118
25119 PRIVATE message dev_mess;
25120
25121 FORWARD _PROTOTYPE( dev_t name_to_dev, (char *path)                    );
25122
25123 /*===========================================================================*
25124  *                              do_mount                                     *
25125  *===========================================================================*/
25126 PUBLIC int do_mount()
25127 {
25128 /* Perform the mount(name, mfile, rd_only) system call. */
25129
25130   register struct inode *rip, *root_ip;
25131   struct super_block *xp, *sp;
25132   dev_t dev;
25133   mode_t bits;
25134   int rdir, mdir;               /* TRUE iff {root|mount} file is dir */
25135   int r, found, major, task;
25136
25137   /* Only the super-user may do MOUNT. */
25138   if (!super_user) return(EPERM);
25139
25140   /* If 'name' is not for a block special file, return error. */
25141   if (fetch_name(name1, name1_length, M1) != OK) return(err_code);
25142   if ( (dev = name_to_dev(user_path)) == NO_DEV) return(err_code);
25143
25144   /* Scan super block table to see if dev already mounted & find a free slot.*/
25145   sp = NIL_SUPER;
25146   found = FALSE;
25147   for (xp = &super_block[0]; xp < &super_block[NR_SUPERS]; xp++) {
25148        if (xp->s_dev == dev) found = TRUE;     /* is it mounted already? */
25149        if (xp->s_dev == NO_DEV) sp = xp;       /* record free slot */
25150   }
25151   if (found) return(EBUSY);     /* already mounted */
25152   if (sp == NIL_SUPER) return(ENFILE);  /* no super block available */
25153
25154   dev_mess.m_type = DEV_OPEN;           /* distinguish from close */
25155   dev_mess.DEVICE = dev;                /* Touch the device. */
25156   if (rd_only) dev_mess.COUNT = R_BIT;
25157   else  dev_mess.COUNT = R_BIT|W_BIT;
25158
25159   major = (dev >> MAJOR) & BYTE;
25160   if (major <= 0 || major >= max_major) return(ENODEV);
25161   task = dmap[major].dmap_task;         /* device task nr */
25162   (*dmap[major].dmap_open)(task, &dev_mess);
25163   if (dev_mess.REP_STATUS != OK) return(EINVAL);
25164
25165   /* Fill in the super block. */
25166   sp->s_dev = dev;              /* read_super() needs to know which dev */
25167   r = read_super(sp);
25168
25169   /* Is it recognized as a Minix filesystem? */
```

```
25170     if (r != OK) {
25171           dev_mess.m_type = DEV_CLOSE;
25172           dev_mess.DEVICE = dev;
25173           (*dmap[major].dmap_close)(task, &dev_mess);
25174           return(r);
25175     }
25176
25177     /* Now get the inode of the file to be mounted on. */
25178     if (fetch_name(name2, name2_length, M1) != OK) {
25179           sp->s_dev = NO_DEV;
25180           dev_mess.m_type = DEV_CLOSE;
25181           dev_mess.DEVICE = dev;
25182           (*dmap[major].dmap_close)(task, &dev_mess);
25183           return(err_code);
25184     }
25185     if ( (rip = eat_path(user_path)) == NIL_INODE) {
25186           sp->s_dev = NO_DEV;
25187           dev_mess.m_type = DEV_CLOSE;
25188           dev_mess.DEVICE = dev;
25189           (*dmap[major].dmap_close)(task, &dev_mess);
25190           return(err_code);
25191     }
25192
25193     /* It may not be busy. */
25194     r = OK;
25195     if (rip->i_count > 1) r = EBUSY;
25196
25197     /* It may not be special. */
25198     bits = rip->i_mode & I_TYPE;
25199     if (bits == I_BLOCK_SPECIAL || bits == I_CHAR_SPECIAL) r = ENOTDIR;
25200
25201     /* Get the root inode of the mounted file system. */
25202     root_ip = NIL_INODE;          /* if 'r' not OK, make sure this is defined */
25203     if (r == OK) {
25204           if ( (root_ip = get_inode(dev, ROOT_INODE)) == NIL_INODE) r = err_code;
25205     }
25206     if (root_ip != NIL_INODE && root_ip->i_mode == 0) r = EINVAL;
25207
25208     /* File types of 'rip' and 'root_ip' may not conflict. */
25209     if (r == OK) {
25210           mdir = ((rip->i_mode & I_TYPE) == I_DIRECTORY);  /* TRUE iff dir */
25211           rdir = ((root_ip->i_mode & I_TYPE) == I_DIRECTORY);
25212           if (!mdir && rdir) r = EISDIR;
25213     }
25214
25215     /* If error, return the super block and both inodes; release the maps. */
25216     if (r != OK) {
25217           put_inode(rip);
25218           put_inode(root_ip);
25219           (void) do_sync();
25220           invalidate(dev);
25221
25222           sp->s_dev = NO_DEV;
25223           dev_mess.m_type = DEV_CLOSE;
25224           dev_mess.DEVICE = dev;
25225           (*dmap[major].dmap_close)(task, &dev_mess);
25226           return(r);
25227     }
25228
25229     /* Nothing else can go wrong.  Perform the mount. */
```



```
25230     rip->i_mount = I_MOUNT;       /* this bit says the inode is mounted on */
25231     sp->s_imount = rip;
25232     sp->s_isup = root_ip;
25233     sp->s_rd_only = rd_only;
25234     return(OK);
25235   }


25238   /*===========================================================================*
25239    *                              do_umount                                    *
25240    *===========================================================================*/
25241   PUBLIC int do_umount()
25242   {
25243   /* Perform the umount(name) system call. */
25244
25245     register struct inode *rip;
25246     struct super_block *sp, *sp1;
25247     dev_t dev;
25248     int count;
25249     int major, task;
25250
25251     /* Only the super-user may do UMOUNT. */
25252     if (!super_user) return(EPERM);
25253
25254     /* If 'name' is not for a block special file, return error. */
25255     if (fetch_name(name, name_length, M3) != OK) return(err_code);
25256     if ( (dev = name_to_dev(user_path)) == NO_DEV) return(err_code);
25257
25258     /* See if the mounted device is busy.  Only 1 inode using it should be
25259      * open -- the root inode -- and that inode only 1 time.
25260      */
25261     count = 0;
25262     for (rip = &inode[0]; rip< &inode[NR_INODES]; rip++)
25263           if (rip->i_count > 0 && rip->i_dev == dev) count += rip->i_count;
25264     if (count > 1) return(EBUSY); /* can't umount a busy file system */
25265
25266     /* Find the super block. */
25267     sp = NIL_SUPER;
25268     for (sp1 = &super_block[0]; sp1 < &super_block[NR_SUPERS]; sp1++) {
25269           if (sp1->s_dev == dev) {
25270                 sp = sp1;
25271                 break;
25272           }
25273     }
25274
25275     /* Sync the disk, and invalidate cache. */
25276     (void) do_sync();             /* force any cached blocks out of memory */
25277     invalidate(dev);              /* invalidate cache entries for this dev */
25278     if (sp == NIL_SUPER) return(EINVAL);
25279
25280     major = (dev >> MAJOR) & BYTE;        /* major device nr */
25281     task = dmap[major].dmap_task; /* device task nr */
25282     dev_mess.m_type = DEV_CLOSE;          /* distinguish from open */
25283     dev_mess.DEVICE = dev;
25284     (*dmap[major].dmap_close)(task, &dev_mess);
25285
25286     /* Finish off the unmount. */
25287     sp->s_imount->i_mount = NO_MOUNT;     /* inode returns to normal */
25288     put_inode(sp->s_imount);      /* release the inode mounted on */
25289     put_inode(sp->s_isup);        /* release the root inode of the mounted fs */
```

```
25290       sp->s_imount = NIL_INODE;
25291       sp->s_dev = NO_DEV;
25292       return(OK);
25293     }


25296     /*===========================================================================*
25297      *                              name_to_dev                                  *
25298      *===========================================================================*/
25299     PRIVATE dev_t name_to_dev(path)
25300     char *path;                   /* pointer to path name */
25301     {
25302     /* Convert the block special file 'path' to a device number.  If 'path'
25303      * is not a block special file, return error code in 'err_code'.
25304      */
25305
25306       register struct inode *rip;
25307       register dev_t dev;
25308
25309       /* If 'path' can't be opened, give up immediately. */
25310       if ( (rip = eat_path(path)) == NIL_INODE) return(NO_DEV);
25311
25312       /* If 'path' is not a block special file, return error. */
25313       if ( (rip->i_mode & I_TYPE) != I_BLOCK_SPECIAL) {
25314             err_code = ENOTBLK;
25315             put_inode(rip);
25316             return(NO_DEV);
25317       }
25318
25319       /* Extract the device number. */
25320       dev = (dev_t) rip->i_zone[0];
25321       put_inode(rip);
25322       return(dev);
25323     }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/fs/link.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

25400     /* This file handles the LINK and UNLINK system calls.  It also deals with
25401      * deallocating the storage used by a file when the last UNLINK is done to a
25402      * file and the blocks must be returned to the free block pool.
25403      *
25404      * The entry points into this file are
25405      *   do_link:   perform the LINK system call
25406      *   do_unlink: perform the UNLINK and RMDIR system calls
25407      *   do_rename: perform the RENAME system call
25408      *   truncate:  release all the blocks associated with an inode
25409      */
25410
25411     #include "fs.h"
25412     #include <sys/stat.h>
25413     #include <string.h>
25414     #include <minix/callnr.h>
25415     #include "buf.h"
25416     #include "file.h"
25417     #include "fproc.h"
25418     #include "inode.h"
25419     #include "param.h"
```



```
25420     #include "super.h"
25421
25422     #define SAME 1000
25423
25424     FORWARD _PROTOTYPE( int remove_dir, (struct inode *rldirp, struct inode *rip,
25425                               char dir_name[NAME_MAX])                        );
25426
25427     FORWARD _PROTOTYPE( int unlink_file, (struct inode *dirp, struct inode *rip,
25428                               char file_name[NAME_MAX])                       );
25429
25430
25431     /*===========================================================================*
25432      *                                do_link                                    *
25433      *===========================================================================*/
25434     PUBLIC int do_link()
25435     {
25436     /* Perform the link(name1, name2) system call. */
25437
25438       register struct inode *ip, *rip;
25439       register int r;
25440       char string[NAME_MAX];
25441       struct inode *new_ip;
25442
25443       /* See if 'name' (file to be linked) exists. */
25444       if (fetch_name(name1, name1_length, M1) != OK) return(err_code);
25445       if ( (rip = eat_path(user_path)) == NIL_INODE) return(err_code);
25446
25447       /* Check to see if the file has maximum number of links already. */
25448       r = OK;
25449       if ( (rip->i_nlinks & BYTE) >= LINK_MAX) r = EMLINK;
25450
25451       /* Only super_user may link to directories. */
25452       if (r == OK)
25453             if ( (rip->i_mode & I_TYPE) == I_DIRECTORY && !super_user) r = EPERM;
25454
25455       /* If error with 'name', return the inode. */
25456       if (r != OK) {
25457             put_inode(rip);
25458             return(r);
25459       }
25460
25461       /* Does the final directory of 'name2' exist? */
25462       if (fetch_name(name2, name2_length, M1) != OK) {
25463             put_inode(rip);
25464             return(err_code);
25465       }
25466       if ( (ip = last_dir(user_path, string)) == NIL_INODE) r = err_code;
25467
25468       /* If 'name2' exists in full (even if no space) set 'r' to error. */
25469       if (r == OK) {
25470             if ( (new_ip = advance(ip, string)) == NIL_INODE) {
25471                   r = err_code;
25472                   if (r == ENOENT) r = OK;
25473             } else {
25474                   put_inode(new_ip);
25475                   r = EEXIST;
25476             }
25477       }
25478
25479       /* Check for links across devices. */
```

```
25480     if (r == OK)
25481           if (rip->i_dev != ip->i_dev) r = EXDEV;
25482
25483     /* Try to link. */
25484     if (r == OK)
25485           r = search_dir(ip, string, &rip->i_num, ENTER);
25486
25487     /* If success, register the linking. */
25488     if (r == OK) {
25489           rip->i_nlinks++;
25490           rip->i_update |= CTIME;
25491           rip->i_dirt = DIRTY;
25492     }
25493
25494     /* Done.  Release both inodes. */
25495     put_inode(rip);
25496     put_inode(ip);
25497     return(r);
25498   }


25501   /*===========================================================================*
25502    *                                do_unlink                                  *
25503    *===========================================================================*/
25504   PUBLIC int do_unlink()
25505   {
25506   /* Perform the unlink(name) or rmdir(name) system call. The code for these two
25507    * is almost the same.  They differ only in some condition testing.  Unlink()
25508    * may be used by the superuser to do dangerous things; rmdir() may not.
25509    */
25510
25511     register struct inode *rip;
25512     struct inode *rldirp;
25513     int r;
25514     char string[NAME_MAX];
25515
25516     /* Get the last directory in the path. */
25517     if (fetch_name(name, name_length, M3) != OK) return(err_code);
25518     if ( (rldirp = last_dir(user_path, string)) == NIL_INODE)
25519           return(err_code);
25520
25521     /* The last directory exists.  Does the file also exist? */
25522     r = OK;
25523     if ( (rip = advance(rldirp, string)) == NIL_INODE) r = err_code;
25524
25525     /* If error, return inode. */
25526     if (r != OK) {
25527           put_inode(rldirp);
25528           return(r);
25529     }
25530
25531     /* Do not remove a mount point. */
25532     if (rip->i_num == ROOT_INODE) {
25533           put_inode(rldirp);
25534           put_inode(rip);
25535           return(EBUSY);
25536     }
25537
25538     /* Now test if the call is allowed, separately for unlink() and rmdir(). */
25539     if (fs_call == UNLINK) {
```



```
25540           /* Only the su may unlink directories, but the su can unlink any dir.*/
25541           if ( (rip->i_mode & I_TYPE) == I_DIRECTORY && !super_user) r = EPERM;
25542
25543           /* Don't unlink a file if it is the root of a mounted file system. */
25544           if (rip->i_num == ROOT_INODE) r = EBUSY;
25545
25546           /* Actually try to unlink the file; fails if parent is mode 0 etc. */
25547           if (r == OK) r = unlink_file(rldirp, rip, string);
25548
25549     } else {
25550           r = remove_dir(rldirp, rip, string); /* call is RMDIR */
25551     }
25552
25553     /* If unlink was possible, it has been done, otherwise it has not. */
25554     put_inode(rip);
25555     put_inode(rldirp);
25556     return(r);
25557   }


25560   /*===========================================================================*
25561    *                                do_rename                                  *
25562    *===========================================================================*/
25563   PUBLIC int do_rename()
25564   {
25565   /* Perform the rename(name1, name2) system call. */
25566
25567     struct inode *old_dirp, *old_ip;      /* ptrs to old dir, file inodes */
25568     struct inode *new_dirp, *new_ip;      /* ptrs to new dir, file inodes */
25569     struct inode *new_superdirp, *next_new_superdirp;
25570     int r = OK;                           /* error flag; initially no error */
25571     int odir, ndir;                       /* TRUE iff {old|new} file is dir */
25572     int same_pdir;                        /* TRUE iff parent dirs are the same */
25573     char old_name[NAME_MAX], new_name[NAME_MAX];
25574     ino_t numb;
25575     int r1;
25576
25577     /* See if 'name1' (existing file) exists.  Get dir and file inodes. */
25578     if (fetch_name(name1, name1_length, M1) != OK) return(err_code);
25579     if ( (old_dirp = last_dir(user_path, old_name))==NIL_INODE) return(err_code);
25580
25581     if ( (old_ip = advance(old_dirp, old_name)) == NIL_INODE) r = err_code;
25582
25583     /* See if 'name2' (new name) exists.  Get dir and file inodes. */
25584     if (fetch_name(name2, name2_length, M1) != OK) r = err_code;
25585     if ( (new_dirp = last_dir(user_path, new_name)) == NIL_INODE) r = err_code;
25586     new_ip = advance(new_dirp, new_name); /* not required to exist */
25587
25588     if (old_ip != NIL_INODE)
25589           odir = ((old_ip->i_mode & I_TYPE) == I_DIRECTORY);  /* TRUE iff dir */
25590
25591     /* If it is ok, check for a variety of possible errors. */
25592     if (r == OK) {
25593           same_pdir = (old_dirp == new_dirp);
25594
25595           /* The old inode must not be a superdirectory of the new last dir. */
25596           if (odir && !same_pdir) {
25597                 dup_inode(new_superdirp = new_dirp);
25598                 while (TRUE) {          /* may hang in a file system loop */
25599                       if (new_superdirp == old_ip) {
```

```
25600                                   r = EINVAL;
25601                                   break;
25602                           }
25603                           next_new_superdirp = advance(new_superdirp, dot2);
25604                           put_inode(new_superdirp);
25605                           if (next_new_superdirp == new_superdirp)
25606                                   break;  /* back at system root directory */
25607                           new_superdirp = next_new_superdirp;
25608                           if (new_superdirp == NIL_INODE) {
25609                                   /* Missing ".." entry.  Assume the worst. */
25610                                   r = EINVAL;
25611                                   break;
25612                           }
25613                   }
25614                   put_inode(new_superdirp);
25615           }
25616
25617           /* The old or new name must not be . or .. */
25618           if (strcmp(old_name, ".")==0 || strcmp(old_name, "..")==0 ||
25619               strcmp(new_name, ".")==0 || strcmp(new_name, "..")==0) r = EINVAL;
25620
25621           /* Both parent directories must be on the same device. */
25622           if (old_dirp->i_dev != new_dirp->i_dev) r = EXDEV;
25623
25624           /* Parent dirs must be writable, searchable and on a writable device */
25625           if ((rl = forbidden(old_dirp, W_BIT | X_BIT)) != OK ||
25626               (rl = forbidden(new_dirp, W_BIT | X_BIT)) != OK) r = rl;
25627
25628           /* Some tests apply only if the new path exists. */
25629           if (new_ip == NIL_INODE) {
25630                   /* don't rename a file with a file system mounted on it. */
25631                   if (old_ip->i_dev != old_dirp->i_dev) r = EXDEV;
25632                   if (odir && (new_dirp->i_nlinks & BYTE) >= LINK_MAX &&
25633                       !same_pdir && r == OK) r = EMLINK;
25634           } else {
25635                   if (old_ip == new_ip) r = SAME; /* old=new */
25636
25637                   /* has the old file or new file a file system mounted on it? */
25638                   if (old_ip->i_dev != new_ip->i_dev) r = EXDEV;
25639
25640                   ndir = ((new_ip->i_mode & I_TYPE) == I_DIRECTORY); /* dir ? */
25641                   if (odir == TRUE && ndir == FALSE) r = ENOTDIR;
25642                   if (odir == FALSE && ndir == TRUE) r = EISDIR;
25643           }
25644   }
25645
25646   /* If a process has another root directory than the system root, we might
25647    * "accidently" be moving it's working directory to a place where it's
25648    * root directory isn't a super directory of it anymore. This can make
25649    * the function chroot useless. If chroot will be used often we should
25650    * probably check for it here.
25651    */
25652
25653   /* The rename will probably work. Only two things can go wrong now:
25654    * 1. being unable to remove the new file. (when new file already exists)
25655    * 2. being unable to make the new directory entry. (new file doesn't exists)
25656    *     [directory has to grow by one block and cannot because the disk
25657    *      is completely full].
25658    */
25659   if (r == OK) {
```



```
25660           if (new_ip != NIL_INODE) {
25661                   /* There is already an entry for 'new'. Try to remove it. */
25662                   if (odir)
25663                           r = remove_dir(new_dirp, new_ip, new_name);
25664                   else
25665                           r = unlink_file(new_dirp, new_ip, new_name);
25666           }
25667           /* if r is OK, the rename will succeed, while there is now an
25668            * unused entry in the new parent directory.
25669            */
25670   }
25671
25672   if (r == OK) {
25673           /* If the new name will be in the same parent directory as the old one,
25674            * first remove the old name to free an entry for the new name,
25675            * otherwise first try to create the new name entry to make sure
25676            * the rename will succeed.
25677            */
25678           numb = old_ip->i_num;           /* inode number of old file */
25679
25680           if (same_pdir) {
25681                   r = search_dir(old_dirp, old_name, (ino_t *) 0, DELETE);
25682                                                   /* shouldn't go wrong. */
25683                   if (r==OK) (void) search_dir(old_dirp, new_name, &numb, ENTER);
25684           } else {
25685                   r = search_dir(new_dirp, new_name, &numb, ENTER);
25686                   if (r == OK)
25687                       (void) search_dir(old_dirp, old_name, (ino_t *) 0, DELETE);
25688           }
25689   }
25690   /* If r is OK, the ctime and mtime of old_dirp and new_dirp have been marked
25691    * for update in search_dir.
25692    */
25693
25694   if (r == OK && odir && !same_pdir) {
25695           /* Update the .. entry in the directory (still points to old_dirp). */
25696           numb = new_dirp->i_num;
25697           (void) unlink_file(old_ip, NIL_INODE, dot2);
25698           if (search_dir(old_ip, dot2, &numb, ENTER) == OK) {
25699                   /* New link created. */
25700                   new_dirp->i_nlinks++;
25701                   new_dirp->i_dirt = DIRTY;
25702           }
25703   }
25704
25705   /* Release the inodes. */
25706   put_inode(old_dirp);
25707   put_inode(old_ip);
25708   put_inode(new_dirp);
25709   put_inode(new_ip);
25710   return(r == SAME ? OK : r);
25711 }


25714 /*===========================================================================*
25715  *                                truncate                                   *
25716  *===========================================================================*/
25717 PUBLIC void truncate(rip)
25718 register struct inode *rip;     /* pointer to inode to be truncated */
25719 {
```

```
25720   /* Remove all the zones from the inode 'rip' and mark it dirty. */
25721
25722     register block_t b;
25723     zone_t z, zone_size, z1;
25724     off_t position;
25725     int i, scale, file_type, waspipe, single, nr_indirects;
25726     struct buf *bp;
25727     dev_t dev;
25728
25729     file_type = rip->i_mode & I_TYPE;     /* check to see if file is special */
25730     if (file_type == I_CHAR_SPECIAL || file_type == I_BLOCK_SPECIAL) return;
25731     dev = rip->i_dev;             /* device on which inode resides */
25732     scale = rip->i_sp->s_log_zone_size;
25733     zone_size = (zone_t) BLOCK_SIZE << scale;
25734     nr_indirects = rip->i_nindirs;
25735
25736     /* Pipes can shrink, so adjust size to make sure all zones are removed. */
25737     waspipe = rip->i_pipe == I_PIPE;      /* TRUE is this was a pipe */
25738     if (waspipe) rip->i_size = PIPE_SIZE;
25739
25740     /* Step through the file a zone at a time, finding and freeing the zones. */
25741     for (position = 0; position < rip->i_size; position += zone_size) {
25742           if ( (b = read_map(rip, position)) != NO_BLOCK) {
25743                 z = (zone_t) b >> scale;
25744                 free_zone(dev, z);
25745           }
25746     }
25747
25748     /* All the data zones have been freed.  Now free the indirect zones. */
25749     rip->i_dirt = DIRTY;
25750     if (waspipe) {
25751           wipe_inode(rip);        /* clear out inode for pipes */
25752           return;                 /* indirect slots contain file positions */
25753     }
25754     single = rip->i_ndzones;
25755     free_zone(dev, rip->i_zone[single]);  /* single indirect zone */
25756     if ( (z = rip->i_zone[single+1]) != NO_ZONE) {
25757           /* Free all the single indirect zones pointed to by the double. */
25758           b = (block_t) z << scale;
25759           bp = get_block(dev, b, NORMAL); /* get double indirect zone */
25760           for (i = 0; i < nr_indirects; i++) {
25761                 z1 = rd_indir(bp, i);
25762                 free_zone(dev, z1);
25763           }
25764
25765           /* Now free the double indirect zone itself. */
25766           put_block(bp, INDIRECT_BLOCK);
25767           free_zone(dev, z);
25768     }
25769
25770     /* Leave zone numbers for de(1) to recover file after an unlink(2).  */
25771   }


25774   /*===========================================================================*
25775    *                               remove_dir                                  *
25776    *===========================================================================*/
25777   PRIVATE int remove_dir(rldirp, rip, dir_name)
25778   struct inode *rldirp;                 /* parent directory */
25779   struct inode *rip;                    /* directory to be removed */
```



```
25780   char dir_name[NAME_MAX];              /* name of directory to be removed */
25781   {
25782     /* A directory file has to be removed. Five conditions have to met:
25783      *    - The file must be a directory
25784      *    - The directory must be empty (except for . and ..)
25785      *    - The final component of the path must not be . or ..
25786      *    - The directory must not be the root of a mounted file system
25787      *    - The directory must not be anybody's root/working directory
25788      */
25789
25790     int r;
25791     register struct fproc *rfp;
25792
25793     /* search_dir checks that rip is a directory too. */
25794     if ((r = search_dir(rip, "", (ino_t *) 0, IS_EMPTY)) != OK) return r;
25795
25796     if (strcmp(dir_name, ".") == 0 || strcmp(dir_name, "..") == 0)return(EINVAL);
25797     if (rip->i_num == ROOT_INODE) return(EBUSY); /* can't remove 'root' */
25798
25799     for (rfp = &fproc[INIT_PROC_NR + 1]; rfp < &fproc[NR_PROCS]; rfp++)
25800           if (rfp->fp_workdir == rip || rfp->fp_rootdir == rip) return(EBUSY);
25801                                 /* can't remove anybody's working dir */
25802
25803     /* Actually try to unlink the file; fails if parent is mode 0 etc. */
25804     if ((r = unlink_file(rldirp, rip, dir_name)) != OK) return r;
25805
25806     /* Unlink . and .. from the dir. The super user can link and unlink any dir,
25807      * so don't make too many assumptions about them.
25808      */
25809     (void) unlink_file(rip, NIL_INODE, dot1);
25810     (void) unlink_file(rip, NIL_INODE, dot2);
25811     return(OK);
25812   }


25815   /*===========================================================================*
25816    *                               unlink_file                                 *
25817    *===========================================================================*/
25818   PRIVATE int unlink_file(dirp, rip, file_name)
25819   struct inode *dirp;             /* parent directory of file */
25820   struct inode *rip;              /* inode of file, may be NIL_INODE too. */
25821   char file_name[NAME_MAX];       /* name of file to be removed */
25822   {
25823   /* Unlink 'file_name'; rip must be the inode of 'file_name' or NIL_INODE. */
25824
25825     ino_t numb;                   /* inode number */
25826     int   r;
25827
25828     /* If rip is not NIL_INODE, it is used to get faster access to the inode. */
25829     if (rip == NIL_INODE) {
25830           /* Search for file in directory and try to get its inode. */
25831           err_code = search_dir(dirp, file_name, &numb, LOOK_UP);
25832           if (err_code == OK) rip = get_inode(dirp->i_dev, (int) numb);
25833           if (err_code != OK || rip == NIL_INODE) return(err_code);
25834     } else {
25835           dup_inode(rip);         /* inode will be returned with put_inode */
25836     }
25837
25838     r = search_dir(dirp, file_name, (ino_t *) 0, DELETE);
25839
```

```
25840       if (r == OK) {
25841             rip->i_nlinks--;        /* entry deleted from parent's dir */
25842             rip->i_update |= CTIME;
25843             rip->i_dirt = DIRTY;
25844       }
25845
25846       put_inode(rip);
25847       return(r);
25848     }
```

++

src/fs/stadir.c

++

```
25900     /* This file contains the code for performing four system calls relating to
25901      * status and directories.
25902      *
25903      * The entry points into this file are
25904      *   do_chdir:  perform the CHDIR system call
25905      *   do_chroot: perform the CHROOT system call
25906      *   do_stat:   perform the STAT system call
25907      *   do_fstat:  perform the FSTAT system call
25908      */
25909
25910     #include "fs.h"
25911     #include <sys/stat.h>
25912     #include "file.h"
25913     #include "fproc.h"
25914     #include "inode.h"
25915     #include "param.h"
25916
25917     FORWARD _PROTOTYPE( int change, (struct inode **iip, char *name_ptr, int len));
25918     FORWARD _PROTOTYPE( int stat_inode, (struct inode *rip, struct filp *fil_ptr,
25919                               char *user_addr)                              );
25920
25921     /*===========================================================================*
25922      *                               do_chdir                                    *
25923      *===========================================================================*/
25924     PUBLIC int do_chdir()
25925     {
25926     /* Change directory.  This function is  also called by MM to simulate a chdir
25927      * in order to do EXEC, etc.  It also changes the root directory, the uids and
25928      * gids, and the umask.
25929      */
25930
25931       int r;
25932       register struct fproc *rfp;
25933
25934       if (who == MM_PROC_NR) {
25935             rfp = &fproc[slot1];
25936             put_inode(fp->fp_rootdir);
25937             dup_inode(fp->fp_rootdir = rfp->fp_rootdir);
25938             put_inode(fp->fp_workdir);
25939             dup_inode(fp->fp_workdir = rfp->fp_workdir);
25940
25941             /* MM uses access() to check permissions.  To make this work, pretend
25942              * that the user's real ids are the same as the user's effective ids.
25943              * FS calls other than access() do not use the real ids, so are not
25944              * affected.
```



```
25945              */
25946             fp->fp_realuid =
25947             fp->fp_effuid = rfp->fp_effuid;
25948             fp->fp_realgid =
25949             fp->fp_effgid = rfp->fp_effgid;
25950             fp->fp_umask = rfp->fp_umask;
25951             return(OK);
25952       }
25953
25954       /* Perform the chdir(name) system call. */
25955       r = change(&fp->fp_workdir, name, name_length);
25956       return(r);
25957     }


25960     /*===========================================================================*
25961      *                               do_chroot                                   *
25962      *===========================================================================*/
25963     PUBLIC int do_chroot()
25964     {
25965     /* Perform the chroot(name) system call. */
25966
25967       register int r;
25968
25969       if (!super_user) return(EPERM);       /* only su may chroot() */
25970       r = change(&fp->fp_rootdir, name, name_length);
25971       return(r);
25972     }


25975     /*===========================================================================*
25976      *                               change                                      *
25977      *===========================================================================*/
25978     PRIVATE int change(iip, name_ptr, len)
25979     struct inode **iip;             /* pointer to the inode pointer for the dir */
25980     char *name_ptr;                 /* pointer to the directory name to change to */
25981     int len;                        /* length of the directory name string */
25982     {
25983     /* Do the actual work for chdir() and chroot(). */
25984
25985       struct inode *rip;
25986       register int r;
25987
25988       /* Try to open the new directory. */
25989       if (fetch_name(name_ptr, len, M3) != OK) return(err_code);
25990       if ( (rip = eat_path(user_path)) == NIL_INODE) return(err_code);
25991
25992       /* It must be a directory and also be searchable. */
25993       if ( (rip->i_mode & I_TYPE) != I_DIRECTORY)
25994             r = ENOTDIR;
25995       else
25996             r = forbidden(rip, X_BIT);      /* check if dir is searchable */
25997
25998       /* If error, return inode. */
25999       if (r != OK) {
26000             put_inode(rip);
26001             return(r);
26002       }
26003
26004       /* Everything is OK.  Make the change. */
```

```
26005     put_inode(*iip);              /* release the old directory */
26006     *iip = rip;                   /* acquire the new one */
26007     return(OK);
26008   }


26011   /*===========================================================================*
26012    *                                do_stat                                    *
26013    *===========================================================================*/
26014   PUBLIC int do_stat()
26015   {
26016   /* Perform the stat(name, buf) system call. */
26017
26018     register struct inode *rip;
26019     register int r;
26020
26021     /* Both stat() and fstat() use the same routine to do the real work.  That
26022      * routine expects an inode, so acquire it temporarily.
26023      */
26024     if (fetch_name(name1, name1_length, M1) != OK) return(err_code);
26025     if ( (rip = eat_path(user_path)) == NIL_INODE) return(err_code);
26026     r = stat_inode(rip, NIL_FILP, name2); /* actually do the work.*/
26027     put_inode(rip);               /* release the inode */
26028     return(r);
26029   }


26032   /*===========================================================================*
26033    *                                do_fstat                                   *
26034    *===========================================================================*/
26035   PUBLIC int do_fstat()
26036   {
26037   /* Perform the fstat(fd, buf) system call. */
26038
26039     register struct filp *rfilp;
26040
26041     /* Is the file descriptor valid? */
26042     if ( (rfilp = get_filp(fd)) == NIL_FILP) return(err_code);
26043
26044     return(stat_inode(rfilp->filp_ino, rfilp, buffer));
26045   }


26048   /*===========================================================================*
26049    *                                stat_inode                                 *
26050    *===========================================================================*/
26051   PRIVATE int stat_inode(rip, fil_ptr, user_addr)
26052   register struct inode *rip;     /* pointer to inode to stat */
26053   struct filp *fil_ptr;           /* filp pointer, supplied by 'fstat' */
26054   char *user_addr;                /* user space address where stat buf goes */
26055   {
26056   /* Common code for stat and fstat system calls. */
26057
26058     struct stat statbuf;
26059     mode_t mo;
26060     int r, s;
26061
26062     /* Update the atime, ctime, and mtime fields in the inode, if need be. */
26063     if (rip->i_update) update_times(rip);
26064
```



```
26065     /* Fill in the statbuf struct. */
26066     mo = rip->i_mode & I_TYPE;
26067     s = (mo == I_CHAR_SPECIAL || mo == I_BLOCK_SPECIAL);  /* true iff special */
26068     statbuf.st_dev = rip->i_dev;
26069     statbuf.st_ino = rip->i_num;
26070     statbuf.st_mode = rip->i_mode;
26071     statbuf.st_nlink = rip->i_nlinks & BYTE;
26072     statbuf.st_uid = rip->i_uid;
26073     statbuf.st_gid = rip->i_gid & BYTE;
26074     statbuf.st_rdev = (dev_t) (s ? rip->i_zone[0] : NO_DEV);
26075     statbuf.st_size = rip->i_size;
26076
26077     if (rip->i_pipe == I_PIPE) {
26078           statbuf.st_mode &= ~I_REGULAR;  /* wipe out I_REGULAR bit for pipes */
26079           if (fil_ptr != NIL_FILP && fil_ptr->filp_mode & R_BIT)
26080                   statbuf.st_size -= fil_ptr->filp_pos;
26081     }
26082
26083     statbuf.st_atime = rip->i_atime;
26084     statbuf.st_mtime = rip->i_mtime;
26085     statbuf.st_ctime = rip->i_ctime;
26086
26087     /* Copy the struct to user space. */
26088     r = sys_copy(FS_PROC_NR, D, (phys_bytes) &statbuf,
26089                  who, D, (phys_bytes) user_addr, (phys_bytes) sizeof(statbuf));
26090     return(r);
26091   }



++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                  src/fs/protect.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

26100   /* This file deals with protection in the file system.  It contains the code
26101    * for four system calls that relate to protection.
26102    *
26103    * The entry points into this file are
26104    *   do_chmod:  perform the CHMOD system call
26105    *   do_chown:  perform the CHOWN system call
26106    *   do_umask:  perform the UMASK system call
26107    *   do_access: perform the ACCESS system call
26108    *   forbidden: check to see if a given access is allowed on a given inode
26109    */
26110
26111   #include "fs.h"
26112   #include <unistd.h>
26113   #include <minix/callnr.h>
26114   #include "buf.h"
26115   #include "file.h"
26116   #include "fproc.h"
26117   #include "inode.h"
26118   #include "param.h"
26119   #include "super.h"
26120
26121   /*===========================================================================*
26122    *                                do_chmod                                   *
26123    *===========================================================================*/
26124   PUBLIC int do_chmod()
```

```
26125   {
26126   /* Perform the chmod(name, mode) system call. */
26127
26128     register struct inode *rip;
26129     register int r;
26130
26131     /* Temporarily open the file. */
26132     if (fetch_name(name, name_length, M3) != OK) return(err_code);
26133     if ( (rip = eat_path(user_path)) == NIL_INODE) return(err_code);
26134
26135     /* Only the owner or the super_user may change the mode of a file.
26136      * No one may change the mode of a file on a read-only file system.
26137      */
26138     if (rip->i_uid != fp->fp_effuid && !super_user)
26139           r = EPERM;
26140     else
26141           r = read_only(rip);
26142
26143     /* If error, return inode. */
26144     if (r != OK)  {
26145           put_inode(rip);
26146           return(r);
26147     }
26148
26149     /* Now make the change. Clear setgid bit if file is not in caller's grp */
26150     rip->i_mode = (rip->i_mode & ~ALL_MODES) | (mode & ALL_MODES);
26151     if (!super_user && rip->i_gid != fp->fp_effgid)rip->i_mode &= ~I_SET_GID_BIT;
26152     rip->i_update |= CTIME;
26153     rip->i_dirt = DIRTY;
26154
26155     put_inode(rip);
26156     return(OK);
26157   }


26160   /*===========================================================================*
26161    *                                do_chown                                   *
26162    *===========================================================================*/
26163   PUBLIC int do_chown()
26164   {
26165   /* Perform the chown(name, owner, group) system call. */
26166
26167     register struct inode *rip;
26168     register int r;
26169
26170     /* Temporarily open the file. */
26171     if (fetch_name(name1, name1_length, M1) != OK) return(err_code);
26172     if ( (rip = eat_path(user_path)) == NIL_INODE) return(err_code);
26173
26174     /* Not permitted to change the owner of a file on a read-only file sys. */
26175     r = read_only(rip);
26176     if (r == OK) {
26177           /* FS is R/W.  Whether call is allowed depends on ownership, etc. */
26178           if (super_user) {
26179                   /* The super user can do anything. */
26180                   rip->i_uid = owner;     /* others later */
26181           } else {
26182                   /* Regular users can only change groups of their own files. */
26183                   if (rip->i_uid != fp->fp_effuid) r = EPERM;
26184                   if (rip->i_uid != owner) r = EPERM;     /* no giving away */
```



```
26185                   if (fp->fp_effgid != group) r = EPERM;
26186           }
26187     }
26188     if (r == OK) {
26189           rip->i_gid = group;
26190           rip->i_mode &= ~(I_SET_UID_BIT | I_SET_GID_BIT);
26191           rip->i_update |= CTIME;
26192           rip->i_dirt = DIRTY;
26193     }
26194
26195     put_inode(rip);
26196     return(r);
26197   }


26200   /*===========================================================================*
26201    *                                do_umask                                   *
26202    *===========================================================================*/
26203   PUBLIC int do_umask()
26204   {
26205   /* Perform the umask(co_mode) system call. */
26206     register mode_t r;
26207
26208     r = ~fp->fp_umask;            /* set 'r' to complement of old mask */
26209     fp->fp_umask = ~(co_mode & RWX_MODES);
26210     return(r);                    /* return complement of old mask */
26211   }


26214   /*===========================================================================*
26215    *                                do_access                                  *
26216    *===========================================================================*/
26217   PUBLIC int do_access()
26218   {
26219   /* Perform the access(name, mode) system call. */
26220
26221     struct inode *rip;
26222     register int r;
26223
26224     /* First check to see if the mode is correct. */
26225     if ( (mode & ~(R_OK | W_OK | X_OK)) != 0 && mode != F_OK)
26226           return(EINVAL);
26227
26228     /* Temporarily open the file whose access is to be checked. */
26229     if (fetch_name(name, name_length, M3) != OK) return(err_code);
26230     if ( (rip = eat_path(user_path)) == NIL_INODE) return(err_code);
26231
26232     /* Now check the permissions. */
26233     r = forbidden(rip, (mode_t) mode);
26234     put_inode(rip);
26235     return(r);
26236   }


26239   /*===========================================================================*
26240    *                                forbidden                                  *
26241    *===========================================================================*/
26242   PUBLIC int forbidden(rip, access_desired)
26243   register struct inode *rip;     /* pointer to inode to be checked */
26244   mode_t access_desired;  /* RWX bits */
```

```
26245   {
26246   /* Given a pointer to an inode, 'rip', and the access desired, determine
26247    * if the access is allowed, and if not why not.  The routine looks up the
26248    * caller's uid in the 'fproc' table.  If access is allowed, OK is returned
26249    * if it is forbidden, EACCES is returned.
26250    */
26251
26252     register struct inode *old_rip = rip;
26253     register struct super_block *sp;
26254     register mode_t bits, perm_bits;
26255     int r, shift, test_uid, test_gid;
26256
26257     if (rip->i_mount == I_MOUNT)  /* The inode is mounted on. */
26258           for (sp = &super_block[1]; sp < &super_block[NR_SUPERS]; sp++)
26259                   if (sp->s_imount == rip) {
26260                           rip = get_inode(sp->s_dev, ROOT_INODE);
26261                           break;
26262                   } /* if */
26263
26264     /* Isolate the relevant rwx bits from the mode. */
26265     bits = rip->i_mode;
26266     test_uid = (fs_call == ACCESS ? fp->fp_realuid : fp->fp_effuid);
26267     test_gid = (fs_call == ACCESS ? fp->fp_realgid : fp->fp_effgid);
26268     if (test_uid == SU_UID) {
26269           /* Grant read and write permission.  Grant search permission for
26270            * directories.  Grant execute permission (for non-directories) if
26271            * and only if one of the 'X' bits is set.
26272            */
26273           if ( (bits & I_TYPE) == I_DIRECTORY ||
26274                bits & ((X_BIT << 6) | (X_BIT << 3) | X_BIT))
26275                   perm_bits = R_BIT | W_BIT | X_BIT;
26276           else
26277                   perm_bits = R_BIT | W_BIT;
26278     } else {
26279           if (test_uid == rip->i_uid) shift = 6;          /* owner */
26280           else if (test_gid == rip->i_gid ) shift = 3;    /* group */
26281           else shift = 0;                                 /* other */
26282           perm_bits = (bits >> shift) & (R_BIT | W_BIT | X_BIT);
26283     }
26284
26285     /* If access desired is not a subset of what is allowed, it is refused. */
26286     r = OK;
26287     if ((perm_bits | access_desired) != perm_bits) r = EACCES;
26288
26289     /* Check to see if someone is trying to write on a file system that is
26290      * mounted read-only.
26291      */
26292     if (r == OK)
26293           if (access_desired & W_BIT) r = read_only(rip);
26294
26295     if (rip != old_rip) put_inode(rip);
26296
26297     return(r);
26298   }


26301   /*===========================================================================*
26302    *                              read_only                                    *
26303    *===========================================================================*/
26304   PUBLIC int read_only(ip)
```



```
26305     struct inode *ip;               /* ptr to inode whose file sys is to be cked */
26306     {
26307     /* Check to see if the file system on which the inode 'ip' resides is mounted
26308      * read only.  If so, return EROFS, else return OK.
26309      */
26310
26311       register struct super_block *sp;
26312
26313       sp = ip->i_sp;
26314       return(sp->s_rd_only ? EROFS : OK);
26315     }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                  src/fs/time.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

26400     /* This file takes care of those system calls that deal with time.
26401      *
26402      * The entry points into this file are
26403      *   do_utime:  perform the UTIME system call
26404      *   do_time:   perform the TIME system call
26405      *   do_stime:  perform the STIME system call
26406      *   do_tims:   perform the TIMES system call
26407      */
26408
26409     #include "fs.h"
26410     #include <minix/callnr.h>
26411     #include <minix/com.h>
26412     #include "file.h"
26413     #include "fproc.h"
26414     #include "inode.h"
26415     #include "param.h"
26416
26417     PRIVATE message clock_mess;
26418
26419     /*===========================================================================*
26420      *                                do_utime                                   *
26421      *===========================================================================*/
26422     PUBLIC int do_utime()
26423     {
26424     /* Perform the utime(name, timep) system call. */
26425
26426       register struct inode *rip;
26427       register int len, r;
26428
26429       /* Adjust for case of NULL 'timep'. */
26430       len = utime_length;
26431       if (len == 0) len = m.m2_i2;
26432
26433       /* Temporarily open the file. */
26434       if (fetch_name(utime_file, len, M1) != OK) return(err_code);
26435       if ( (rip = eat_path(user_path)) == NIL_INODE) return(err_code);
26436
26437       /* Only the owner of a file or the super_user can change its time. */
26438       r = OK;
26439       if (rip->i_uid != fp->fp_effuid && !super_user) r = EPERM;
```

```
26440     if (utime_length == 0 && r != OK) r = forbidden(rip, W_BIT);
26441     if (read_only(rip) != OK) r = EROFS;  /* not even su can touch if R/O */
26442     if (r == OK) {
26443           if (utime_length == 0) {
26444                   rip->i_atime = clock_time();
26445                   rip->i_mtime = rip->i_atime;
26446           } else {
26447                   rip->i_atime = utime_actime;
26448                   rip->i_mtime = utime_modtime;
26449           }
26450           rip->i_update = CTIME;  /* discard any stale ATIME and MTIME flags */
26451           rip->i_dirt = DIRTY;
26452     }
26453
26454     put_inode(rip);
26455     return(r);
26456   }


26459   /*===========================================================================*
26460    *                                do_time                                    *
26461    *===========================================================================*/
26462   PUBLIC int do_time()
26463
26464   {
26465   /* Perform the time(tp) system call. */
26466
26467     reply_l1 = clock_time();      /* return time in seconds */
26468     return(OK);
26469   }


26472   /*===========================================================================*
26473    *                                do_stime                                   *
26474    *===========================================================================*/
26475   PUBLIC int do_stime()
26476   {
26477   /* Perform the stime(tp) system call. */
26478
26479     register int k;
26480
26481     if (!super_user) return(EPERM);
26482     clock_mess.m_type = SET_TIME;
26483     clock_mess.NEW_TIME = (long) tp;
26484     if ( (k = sendrec(CLOCK, &clock_mess)) != OK) panic("do_stime error", k);
26485     return(OK);
26486   }


26489   /*===========================================================================*
26490    *                                do_tims                                    *
26491    *===========================================================================*/
26492   PUBLIC int do_tims()
26493   {
26494   /* Perform the times(buffer) system call. */
26495
26496     clock_t t[5];
26497
26498     sys_times(who, t);
26499     reply_t1 = t[0];
```



```
26500     reply_t2 = t[1];
26501     reply_t3 = t[2];
26502     reply_t4 = t[3];
26503     reply_t5 = t[4];
26504     return(OK);
26505   }
```

++

src/fs/misc.c

++

```
26600   /* This file contains a collection of miscellaneous procedures.  Some of them
26601    * perform simple system calls.  Some others do a little part of system calls
26602    * that are mostly performed by the Memory Manager.
26603    *
26604    * The entry points into this file are
26605    *   do_dup:      perform the DUP system call
26606    *   do_fcntl:    perform the FCNTL system call
26607    *   do_sync:     perform the SYNC system call
26608    *   do_fork:     adjust the tables after MM has performed a FORK system call
26609    *   do_exec:     handle files with FD_CLOEXEC on after MM has done an EXEC
26610    *   do_exit:     a process has exited; note that in the tables
26611    *   do_set:      set uid or gid for some process
26612    *   do_revive:   revive a process that was waiting for something (e.g. TTY)
26613    */
26614
26615   #include "fs.h"
26616   #include <fcntl.h>
26617   #include <unistd.h>     /* cc runs out of memory with unistd.h :-( */
26618   #include <minix/callnr.h>
26619   #include <minix/com.h>
26620   #include <minix/boot.h>
26621   #include "buf.h"
26622   #include "file.h"
26623   #include "fproc.h"
26624   #include "inode.h"
26625   #include "dev.h"
26626   #include "param.h"
26627
26628
26629   /*===========================================================================*
26630    *                                do_dup                                     *
26631    *===========================================================================*/
26632   PUBLIC int do_dup()
26633   {
26634   /* Perform the dup(fd) or dup2(fd,fd2) system call. These system calls are
26635    * obsolete.  In fact, it is not even possible to invoke them using the
26636    * current library because the library routines call fcntl().  They are
26637    * provided to permit old binary programs to continue to run.
26638    */
26639
26640     register int rfd;
26641     register struct filp *f;
26642     struct filp *dummy;
26643     int r;
26644
```

```
26645     /* Is the file descriptor valid? */
26646     rfd = fd & ~DUP_MASK;          /* kill off dup2 bit, if on */
26647     if ((f = get_filp(rfd)) == NIL_FILP) return(err_code);
26648
26649     /* Distinguish between dup and dup2. */
26650     if (fd == rfd) {                       /* bit not on */
26651           /* dup(fd) */
26652           if ( (r = get_fd(0, 0, &fd2, &dummy)) != OK) return(r);
26653     } else {
26654           /* dup2(fd, fd2) */
26655           if (fd2 < 0 || fd2 >= OPEN_MAX) return(EBADF);
26656           if (rfd == fd2) return(fd2);    /* ignore the call: dup2(x, x) */
26657           fd = fd2;                /* prepare to close fd2 */
26658           (void) do_close();       /* cannot fail */
26659     }
26660
26661     /* Success. Set up new file descriptors. */
26662     f->filp_count++;
26663     fp->fp_filp[fd2] = f;
26664     return(fd2);
26665   }

26667   /*===========================================================================*
26668    *                                do_fcntl                                   *
26669    *===========================================================================*/
26670   PUBLIC int do_fcntl()
26671   {
26672   /* Perform the fcntl(fd, request, ...) system call. */
26673
26674     register struct filp *f;
26675     int new_fd, r, fl;
26676     long cloexec_mask;             /* bit map for the FD_CLOEXEC flag */
26677     long clo_value;                /* FD_CLOEXEC flag in proper position */
26678     struct filp *dummy;
26679
26680     /* Is the file descriptor valid? */
26681     if ((f = get_filp(fd)) == NIL_FILP) return(err_code);
26682
26683     switch (request) {
26684        case F_DUPFD:
26685           /* This replaces the old dup() system call. */
26686           if (addr < 0 || addr >= OPEN_MAX) return(EINVAL);
26687           if ((r = get_fd(addr, 0, &new_fd, &dummy)) != OK) return(r);
26688           f->filp_count++;
26689           fp->fp_filp[new_fd] = f;
26690           return(new_fd);
26691
26692        case F_GETFD:
26693           /* Get close-on-exec flag (FD_CLOEXEC in POSIX Table 6-2). */
26694           return( ((fp->fp_cloexec >> fd) & 01) ? FD_CLOEXEC : 0);
26695
26696        case F_SETFD:
26697           /* Set close-on-exec flag (FD_CLOEXEC in POSIX Table 6-2). */
26698           cloexec_mask = 1L << fd;       /* singleton set position ok */
26699           clo_value = (addr & FD_CLOEXEC ? cloexec_mask : 0L);
26700           fp->fp_cloexec = (fp->fp_cloexec & ~cloexec_mask) | clo_value;
26701           return(OK);
26702
26703        case F_GETFL:
26704           /* Get file status flags (O_NONBLOCK and O_APPEND). */
```



```
26705           fl = f->filp_flags & (O_NONBLOCK | O_APPEND | O_ACCMODE);
26706           return(fl);
26707
26708        case F_SETFL:
26709           /* Set file status flags (O_NONBLOCK and O_APPEND). */
26710           fl = O_NONBLOCK | O_APPEND;
26711           f->filp_flags = (f->filp_flags & ~fl) | (addr & fl);
26712           return(OK);
26713
26714        case F_GETLK:
26715        case F_SETLK:
26716        case F_SETLKW:
26717           /* Set or clear a file lock. */
26718           r = lock_op(f, request);
26719           return(r);
26720
26721        default:
26722           return(EINVAL);
26723     }
26724   }


26727   /*===========================================================================*
26728    *                                do_sync                                    *
26729    *===========================================================================*/
26730   PUBLIC int do_sync()
26731   {
26732   /* Perform the sync() system call.  Flush all the tables. */
26733
26734     register struct inode *rip;
26735     register struct buf *bp;
26736
26737     /* The order in which the various tables are flushed is critical.  The
26738      * blocks must be flushed last, since rw_inode() leaves its results in
26739      * the block cache.
26740      */
26741
26742     /* Write all the dirty inodes to the disk. */
26743     for (rip = &inode[0]; rip < &inode[NR_INODES]; rip++)
26744           if (rip->i_count > 0 && rip->i_dirt == DIRTY) rw_inode(rip, WRITING);
26745
26746     /* Write all the dirty blocks to the disk, one drive at a time. */
26747     for (bp = &buf[0]; bp < &buf[NR_BUFS]; bp++)
26748           if (bp->b_dev != NO_DEV && bp->b_dirt == DIRTY) flushall(bp->b_dev);
26749
26750     return(OK);           /* sync() can't fail */
26751   }


26754   /*===========================================================================*
26755    *                                do_fork                                    *
26756    *===========================================================================*/
26757   PUBLIC int do_fork()
26758   {
26759   /* Perform those aspects of the fork() system call that relate to files.
26760    * In particular, let the child inherit its parent's file descriptors.
26761    * The parent and child parameters tell who forked off whom. The file
26762    * system uses the same slot numbers as the kernel.  Only MM makes this call.
26763    */
26764
```

```
26765     register struct fproc *cp;
26766     int i;
26767
26768     /* Only MM may make this call directly. */
26769     if (who != MM_PROC_NR) return(EGENERIC);
26770
26771     /* Copy the parent's fproc struct to the child. */
26772     fproc[child] = fproc[parent];
26773
26774     /* Increase the counters in the 'filp' table. */
26775     cp = &fproc[child];
26776     for (i = 0; i < OPEN_MAX; i++)
26777           if (cp->fp_filp[i] != NIL_FILP) cp->fp_filp[i]->filp_count++;
26778
26779     /* Fill in new process id. */
26780     cp->fp_pid = pid;
26781
26782     /* A child is not a process leader. */
26783     cp->fp_sesldr = 0;
26784
26785     /* Record the fact that both root and working dir have another user. */
26786     dup_inode(cp->fp_rootdir);
26787     dup_inode(cp->fp_workdir);
26788     return(OK);
26789   }


26792   /*===========================================================================*
26793    *                                do_exec                                    *
26794    *===========================================================================*/
26795   PUBLIC int do_exec()
26796   {
26797   /* Files can be marked with the FD_CLOEXEC bit (in fp->fp_cloexec).  When
26798    * MM does an EXEC, it calls FS to allow FS to find these files and close them.
26799    */
26800
26801     register int i;
26802     long bitmap;
26803
26804     /* Only MM may make this call directly. */
26805     if (who != MM_PROC_NR) return(EGENERIC);
26806
26807     /* The array of FD_CLOEXEC bits is in the fp_cloexec bit map. */
26808     fp = &fproc[slot1];           /* get_filp() needs 'fp' */
26809     bitmap = fp->fp_cloexec;
26810     if (bitmap == 0) return(OK);  /* normal case, no FD_CLOEXECs */
26811
26812     /* Check the file desriptors one by one for presence of FD_CLOEXEC. */
26813     for (i = 0; i < OPEN_MAX; i++) {
26814           fd = i;
26815           if ( (bitmap >> i) & 01) (void) do_close();
26816     }
26817
26818     return(OK);
26819   }


26822   /*===========================================================================*
26823    *                                do_exit                                    *
26824    *===========================================================================*/
```



```
26825   PUBLIC int do_exit()
26826   {
26827   /* Perform the file system portion of the exit(status) system call. */
26828
26829     register int i, exitee, task;
26830     register struct fproc *rfp;
26831     register struct filp *rfilp;
26832     register struct inode *rip;
26833     int major;
26834     dev_t dev;
26835     message dev_mess;
26836
26837     /* Only MM may do the EXIT call directly. */
26838     if (who != MM_PROC_NR) return(EGENERIC);
26839
26840     /* Nevertheless, pretend that the call came from the user. */
26841     fp = &fproc[slot1];           /* get_filp() needs 'fp' */
26842     exitee = slot1;
26843
26844     if (fp->fp_suspended == SUSPENDED) {
26845           task = -fp->fp_task;
26846           if (task == XPIPE || task == XPOPEN) susp_count--;
26847           pro = exitee;
26848           (void) do_unpause();    /* this always succeeds for MM */
26849           fp->fp_suspended = NOT_SUSPENDED;
26850     }
26851
26852     /* Loop on file descriptors, closing any that are open. */
26853     for (i = 0; i < OPEN_MAX; i++) {
26854           fd = i;
26855           (void) do_close();
26856     }
26857
26858     /* Release root and working directories. */
26859     put_inode(fp->fp_rootdir);
26860     put_inode(fp->fp_workdir);
26861     fp->fp_rootdir = NIL_INODE;
26862     fp->fp_workdir = NIL_INODE;
26863
26864     /* If a session leader exits then revoke access to its controlling tty from
26865      * all other processes using it.
26866      */
26867     if (!fp->fp_sesldr) return(OK);               /* not a session leader */
26868     fp->fp_sesldr = FALSE;
26869     if (fp->fp_tty == 0) return(OK);              /* no controlling tty */
26870     dev = fp->fp_tty;
26871
26872     for (rfp = &fproc[LOW_USER]; rfp < &fproc[NR_PROCS]; rfp++) {
26873           if (rfp->fp_tty == dev) rfp->fp_tty = 0;
26874
26875           for (i = 0; i < OPEN_MAX; i++) {
26876                 if ((rfilp = rfp->fp_filp[i]) == NIL_FILP) continue;
26877                 if (rfilp->filp_mode == FILP_CLOSED) continue;
26878                 rip = rfilp->filp_ino;
26879                 if ((rip->i_mode & I_TYPE) != I_CHAR_SPECIAL) continue;
26880                 if ((dev_t) rip->i_zone[0] != dev) continue;
26881                 dev_mess.m_type = DEV_CLOSE;
26882                 dev_mess.DEVICE = dev;
26883                 major = (dev >> MAJOR) & BYTE;  /* major device nr */
26884                 task = dmap[major].dmap_task;   /* device task nr */
```

```
26885                   (*dmap[major].dmap_close)(task, &dev_mess);
26886                   rfilp->filp_mode = FILP_CLOSED;
26887           }
26888     }
26889     return(OK);
26890   }


26893   /*===========================================================================*
26894    *                              do_set                                       *
26895    *===========================================================================*/
26896   PUBLIC int do_set()
26897   {
26898   /* Set uid_t or gid_t field. */
26899
26900     register struct fproc *tfp;
26901
26902     /* Only MM may make this call directly. */
26903     if (who != MM_PROC_NR) return(EGENERIC);
26904
26905     tfp = &fproc[slot1];
26906     if (fs_call == SETUID) {
26907           tfp->fp_realuid = (uid_t) real_user_id;
26908           tfp->fp_effuid =  (uid_t) eff_user_id;
26909     }
26910     if (fs_call == SETGID) {
26911           tfp->fp_effgid =  (gid_t) eff_grp_id;
26912           tfp->fp_realgid = (gid_t) real_grp_id;
26913     }
26914     return(OK);
26915   }


26918   /*===========================================================================*
26919    *                              do_revive                                    *
26920    *===========================================================================*/
26921   PUBLIC int do_revive()
26922   {
26923   /* A task, typically TTY, has now gotten the characters that were needed for a
26924    * previous read.  The process did not get a reply when it made the call.
26925    * Instead it was suspended.  Now we can send the reply to wake it up.  This
26926    * business has to be done carefully, since the incoming message is from
26927    * a task (to which no reply can be sent), and the reply must go to a process
26928    * that blocked earlier.  The reply to the caller is inhibited by setting the
26929    * 'dont_reply' flag, and the reply to the blocked process is done explicitly
26930    * in revive().
26931    */
26932
26933   #if !ALLOW_USER_SEND
26934     if (who >= LOW_USER) return(EPERM);
26935   #endif
26936
26937     revive(m.REP_PROC_NR, m.REP_STATUS);
26938     dont_reply = TRUE;            /* don't reply to the TTY task */
26939     return(OK);
26940   }
```



```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/fs/device.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

27000   /* When a needed block is not in the cache, it must be fetched from the disk.
27001    * Special character files also require I/O.  The routines for these are here.
27002    *
27003    * The entry points in this file are:
27004    *   dev_io:     perform a read or write on a block or character device
27005    *   dev_opcl:   perform generic device-specific processing for open & close
27006    *   tty_open:   perform tty-specific processing for open
27007    *   ctty_open:  perform controlling-tty-specific processing for open
27008    *   ctty_close: perform controlling-tty-specific processing for close
27009    *   do_setsid:  perform the SETSID system call (FS side)
27010    *   do_ioctl:   perform the IOCTL system call
27011    *   call_task:  procedure that actually calls the kernel tasks
27012    *   call_ctty:  procedure that actually calls task for /dev/tty
27013    */
27014
27015   #include "fs.h"
27016   #include <fcntl.h>
27017   #include <minix/callnr.h>
27018   #include <minix/com.h>
27019   #include "dev.h"
27020   #include "file.h"
27021   #include "fproc.h"
27022   #include "inode.h"
27023   #include "param.h"
27024
27025   PRIVATE message dev_mess;
27026   PRIVATE major, minor, task;
27027
27028   FORWARD _PROTOTYPE( void find_dev, (Dev_t dev)                          );
27029
27030   /*===========================================================================*
27031    *                              dev_io                                       *
27032    *===========================================================================*/
27033   PUBLIC int dev_io(op, nonblock, dev, pos, bytes, proc, buff)
27034   int op;                       /* DEV_READ, DEV_WRITE, DEV_IOCTL, etc. */
27035   int nonblock;                 /* TRUE if nonblocking op */
27036   dev_t dev;                    /* major-minor device number */
27037   off_t pos;                    /* byte position */
27038   int bytes;                    /* how many bytes to transfer */
27039   int proc;                     /* in whose address space is buff? */
27040   char *buff;                   /* virtual address of the buffer */
27041   {
27042   /* Read or write from a device.  The parameter 'dev' tells which one. */
27043
27044     find_dev(dev);              /* load the variables major, minor, and task */
27045
27046     /* Set up the message passed to task. */
27047     dev_mess.m_type   = op;
27048     dev_mess.DEVICE   = (dev >> MINOR) & BYTE;
27049     dev_mess.POSITION = pos;
27050     dev_mess.PROC_NR  = proc;
27051     dev_mess.ADDRESS  = buff;
27052     dev_mess.COUNT    = bytes;
27053     dev_mess.TTY_FLAGS = nonblock; /* temporary kludge */
27054
```

```
27055     /* Call the task. */
27056     (*dmap[major].dmap_rw)(task, &dev_mess);
27057
27058     /* Task has completed.  See if call completed. */
27059     if (dev_mess.REP_STATUS == SUSPEND) {
27060           if (op == DEV_OPEN) task = XPOPEN;
27061           suspend(task);          /* suspend user */
27062     }
27063
27064     return(dev_mess.REP_STATUS);
27065   }


27068   /*===========================================================================*
27069    *                              dev_opcl                                     *
27070    *===========================================================================*/
27071   PUBLIC void dev_opcl(task_nr, mess_ptr)
27072   int task_nr;                    /* which task */
27073   message *mess_ptr;              /* message pointer */
27074   {
27075   /* Called from the dmap struct in table.c on opens & closes of special files.*/
27076
27077     int op;
27078
27079     op = mess_ptr->m_type;        /* save DEV_OPEN or DEV_CLOSE for later */
27080     mess_ptr->DEVICE = (mess_ptr->DEVICE >> MINOR) & BYTE;
27081     mess_ptr->PROC_NR = fp - fproc;
27082
27083     call_task(task_nr, mess_ptr);
27084
27085     /* Task has completed.  See if call completed. */
27086     if (mess_ptr->REP_STATUS == SUSPEND) {
27087           if (op == DEV_OPEN) task_nr = XPOPEN;
27088           suspend(task_nr);       /* suspend user */
27089     }
27090   }

27092   /*===========================================================================*
27093    *                              tty_open                                     *
27094    *===========================================================================*/
27095   PUBLIC void tty_open(task_nr, mess_ptr)
27096   int task_nr;
27097   message *mess_ptr;
27098   {
27099   /* This procedure is called from the dmap struct in table.c on tty opens. */
27100
27101     int r;
27102     dev_t dev;
27103     int flags, proc;
27104     register struct fproc *rfp;
27105
27106     dev = (dev_t) mess_ptr->DEVICE;
27107     flags = mess_ptr->COUNT;
27108     proc = fp - fproc;
27109
27110     /* Add O_NOCTTY to the flags if this process is not a session leader, or
27111      * if it already has a controlling tty, or if it is someone elses
27112      * controlling tty.
27113      */
27114     if (!fp->fp_sesldr || fp->fp_tty != 0) {
```



```
27115           flags |= O_NOCTTY;
27116     } else {
27117           for (rfp = &fproc[LOW_USER]; rfp < &fproc[NR_PROCS]; rfp++) {
27118                 if (rfp->fp_tty == dev) flags |= O_NOCTTY;
27119           }
27120     }
27121
27122     r = dev_io(DEV_OPEN, mode, dev, (off_t) 0, flags, proc, NIL_PTR);
27123
27124     if (r == 1) {
27125           fp->fp_tty = dev;
27126           r = OK;
27127     }
27128
27129     mess_ptr->REP_STATUS = r;
27130   }


27133   /*===========================================================================*
27134    *                              ctty_open                                    *
27135    *===========================================================================*/
27136   PUBLIC void ctty_open(task_nr, mess_ptr)
27137   int task_nr;
27138   message *mess_ptr;
27139   {
27140   /* This procedure is called from the dmap struct in table.c on opening
27141    * /dev/tty, the magic device that translates to the controlling tty.
27142    */
27143
27144     mess_ptr->REP_STATUS = fp->fp_tty == 0 ? ENXIO : OK;
27145   }


27148   /*===========================================================================*
27149    *                              ctty_close                                   *
27150    *===========================================================================*/
27151   PUBLIC void ctty_close(task_nr, mess_ptr)
27152   int task_nr;
27153   message *mess_ptr;
27154   {
27155   /* Close /dev/tty. */
27156
27157     mess_ptr->REP_STATUS = OK;
27158   }


27161   /*===========================================================================*
27162    *                              do_setsid                                    *
27163    *===========================================================================*/
27164   PUBLIC int do_setsid()
27165   {
27166   /* Perform the FS side of the SETSID call, i.e. get rid of the controlling
27167    * terminal of a process, and make the process a session leader.
27168    */
27169     register struct fproc *rfp;
27170
27171     /* Only MM may do the SETSID call directly. */
27172     if (who != MM_PROC_NR) return(ENOSYS);
27173
27174     /* Make the process a session leader with no controlling tty. */
```

```
27175     rfp = &fproc[slot1];
27176     rfp->fp_sesldr = TRUE;
27177     rfp->fp_tty = 0;
27178   }


27181   /*===========================================================================*
27182    *                              do_ioctl                                     *
27183    *===========================================================================*/
27184   PUBLIC int do_ioctl()
27185   {
27186   /* Perform the ioctl(ls_fd, request, argx) system call (uses m2 fmt). */
27187
27188     struct filp *f;
27189     register struct inode *rip;
27190     dev_t dev;
27191
27192     if ( (f = get_filp(ls_fd)) == NIL_FILP) return(err_code);
27193     rip = f->filp_ino;            /* get inode pointer */
27194     if ( (rip->i_mode & I_TYPE) != I_CHAR_SPECIAL
27195          && (rip->i_mode & I_TYPE) != I_BLOCK_SPECIAL) return(ENOTTY);
27196     dev = (dev_t) rip->i_zone[0];
27197     find_dev(dev);
27198
27199     dev_mess= m;
27200
27201     dev_mess.m_type  = DEV_IOCTL;
27202     dev_mess.PROC_NR = who;
27203     dev_mess.TTY_LINE = minor;
27204
27205     /* Call the task. */
27206     (*dmap[major].dmap_rw)(task, &dev_mess);
27207
27208     /* Task has completed.  See if call completed. */
27209     if (dev_mess.REP_STATUS == SUSPEND) {
27210           if (f->filp_flags & O_NONBLOCK) {
27211                 /* Not supposed to block. */
27212                 dev_mess.m_type = CANCEL;
27213                 dev_mess.PROC_NR = who;
27214                 dev_mess.TTY_LINE = minor;
27215                 (*dmap[major].dmap_rw)(task, &dev_mess);
27216                 if (dev_mess.REP_STATUS == EINTR) dev_mess.REP_STATUS = EAGAIN;
27217           } else {
27218                 suspend(task);          /* User must be suspended. */
27219           }
27220     }
27221     return(dev_mess.REP_STATUS);
27222   }


27225   /*===========================================================================*
27226    *                              find_dev                                     *
27227    *===========================================================================*/
27228   PRIVATE void find_dev(dev)
27229   dev_t dev;                    /* device */
27230   {
27231   /* Extract the major and minor device number from the parameter. */
27232
27233     major = (dev >> MAJOR) & BYTE;        /* major device number */
27234     minor = (dev >> MINOR) & BYTE;        /* minor device number */
```



```
27235     if (major >= max_major) {
27236           major = minor = 0;              /* will fail with ENODEV */
27237     }
27238     task = dmap[major].dmap_task; /* which task services the device */
27239   }


27242   /*===========================================================================*
27243    *                              call_task                                    *
27244    *===========================================================================*/
27245   PUBLIC void call_task(task_nr, mess_ptr)
27246   int task_nr;                  /* which task to call */
27247   message *mess_ptr;            /* pointer to message for task */
27248   {
27249   /* All file system I/O ultimately comes down to I/O on major/minor device
27250    * pairs.  These lead to calls on the following routines via the dmap table.
27251    */
27252
27253     int r, proc_nr;
27254     message local_m;
27255
27256     proc_nr = mess_ptr->PROC_NR;
27257
27258     while ((r = sendrec(task_nr, mess_ptr)) == ELOCKED) {
27259           /* sendrec() failed to avoid deadlock. The task 'task_nr' is
27260            * trying to send a REVIVE message for an earlier request.
27261            * Handle it and go try again.
27262            */
27263           if ((r = receive(task_nr, &local_m)) != OK) break;
27264
27265           /* If we're trying to send a cancel message to a task which has just
27266            * sent a completion reply, ignore the reply and abort the cancel
27267            * request. The caller will do the revive for the process.
27268            */
27269           if (mess_ptr->m_type == CANCEL && local_m.REP_PROC_NR == proc_nr)
27270                 return;
27271
27272           /* Otherwise it should be a REVIVE. */
27273           if (local_m.m_type != REVIVE) {
27274                 printf(
27275                 "fs: strange device reply from %d, type = %d, proc = %d\n",
27276                       local_m.m_source,
27277                       local_m.m_type, local_m.REP_PROC_NR);
27278                 continue;
27279           }
27280
27281           revive(local_m.REP_PROC_NR, local_m.REP_STATUS);
27282     }
27283
27284     /* The message received may be a reply to this call, or a REVIVE for some
27285      * other process.
27286      */
27287     for (;;) {
27288           if (r != OK) panic("call_task: can't send/receive", NO_NUM);
27289
27290           /* Did the process we did the sendrec() for get a result? */
27291           if (mess_ptr->REP_PROC_NR == proc_nr) break;
27292
27293           /* Otherwise it should be a REVIVE. */
27294           if (mess_ptr->m_type != REVIVE) {
```

```
27295                   printf(
27296                   "fs: strange device reply from %d, type = %d, proc = %d\n",
27297                           mess_ptr->m_source,
27298                           mess_ptr->m_type, mess_ptr->REP_PROC_NR);
27299                   continue;
27300           }
27301           revive(mess_ptr->REP_PROC_NR, mess_ptr->REP_STATUS);
27302
27303           r = receive(task_nr, mess_ptr);
27304     }
27305   }


27308   /*===========================================================================*
27309    *                              call_ctty                                    *
27310    *===========================================================================*/
27311   PUBLIC void call_ctty(task_nr, mess_ptr)
27312   int task_nr;                    /* not used - for compatibility with dmap_t */
27313   message *mess_ptr;              /* pointer to message for task */
27314   {
27315   /* This routine is only called for one device, namely /dev/tty.  Its job
27316    * is to change the message to use the controlling terminal, instead of the
27317    * major/minor pair for /dev/tty itself.
27318    */
27319
27320     int major_device;
27321
27322     if (fp->fp_tty == 0) {
27323           /* No controlling tty present anymore, return an I/O error. */
27324           mess_ptr->REP_STATUS = EIO;
27325           return;
27326     }
27327     major_device = (fp->fp_tty >> MAJOR) & BYTE;
27328     task_nr = dmap[major_device].dmap_task;       /* task for controlling tty */
27329     mess_ptr->DEVICE = (fp->fp_tty >> MINOR) & BYTE;
27330     call_task(task_nr, mess_ptr);
27331   }


27334   /*===========================================================================*
27335    *                              no_dev                                       *
27336    *===========================================================================*/
27337   PUBLIC void no_dev(task_nr, m_ptr)
27338   int task_nr;                    /* not used - for compatibility with dmap_t */
27339   message *m_ptr;                 /* message pointer */
27340   {
27341   /* No device there. */
27342
27343     m_ptr->REP_STATUS = ENODEV;
27344   }
```



```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/fs/utility.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

27400   /* This file contains a few general purpose utility routines.
27401    *
27402    * The entry points into this file are
27403    *   clock_time:  ask the clock task for the real time
27404    *   copy:        copy a block of data
27405    *   fetch_name:  go get a path name from user space
27406    *   no_sys:      reject a system call that FS does not handle
27407    *   panic:       something awful has occurred;  MINIX cannot continue
27408    *   conv2:       do byte swapping on a 16-bit int
27409    *   conv4:       do byte swapping on a 32-bit long
27410    */
27411
27412   #include "fs.h"
27413   #include <minix/com.h>
27414   #include <minix/boot.h>
27415   #include <unistd.h>
27416   #include "buf.h"
27417   #include "file.h"
27418   #include "fproc.h"
27419   #include "inode.h"
27420   #include "param.h"
27421
27422   PRIVATE int panicking;          /* inhibits recursive panics during sync */
27423   PRIVATE message clock_mess;
27424
27425   /*===========================================================================*
27426    *                              clock_time                                   *
27427    *===========================================================================*/
27428   PUBLIC time_t clock_time()
27429   {
27430   /* This routine returns the time in seconds since 1.1.1970.  MINIX is an
27431    * astrophysically naive system that assumes the earth rotates at a constant
27432    * rate and that such things as leap seconds do not exist.
27433    */
27434
27435     register int k;
27436
27437     clock_mess.m_type = GET_TIME;
27438     if ( (k = sendrec(CLOCK, &clock_mess)) != OK) panic("clock_time err", k);
27439
27440     return( (time_t) clock_mess.NEW_TIME);
27441   }


27444   /*===========================================================================*
27445    *                              fetch_name                                   *
27446    *===========================================================================*/
27447   PUBLIC int fetch_name(path, len, flag)
27448   char *path;                     /* pointer to the path in user space */
27449   int len;                        /* path length, including 0 byte */
27450   int flag;                       /* M3 means path may be in message */
27451   {
27452   /* Go get path and put it in 'user_path'.
27453    * If 'flag' = M3 and 'len' <= M3_STRING, the path is present in 'message'.
27454    * If it is not, go copy it from user space.
```

```
27455     */
27456
27457     register char *rpu, *rpm;
27458     int r;
27459
27460     /* Check name length for validity. */
27461     if (len <= 0) {
27462           err_code = EINVAL;
27463           return(EGENERIC);
27464     }
27465
27466     if (len > PATH_MAX) {
27467           err_code = ENAMETOOLONG;
27468           return(EGENERIC);
27469     }
27470
27471     if (flag == M3 && len <= M3_STRING) {
27472           /* Just copy the path from the message to 'user_path'. */
27473           rpu = &user_path[0];
27474           rpm = pathname;         /* contained in input message */
27475           do { *rpu++ = *rpm++; } while (--len);
27476           r = OK;
27477     } else {
27478           /* String is not contained in the message.  Get it from user space. */
27479           r = sys_copy(who, D, (phys_bytes) path,
27480                   FS_PROC_NR, D, (phys_bytes) user_path, (phys_bytes) len);
27481     }
27482     return(r);
27483   }


27486   /*===========================================================================*
27487    *                              no_sys                                       *
27488    *===========================================================================*/
27489   PUBLIC int no_sys()
27490   {
27491   /* Somebody has used an illegal system call number */
27492
27493     return(EINVAL);
27494   }


27497   /*===========================================================================*
27498    *                              panic                                        *
27499    *===========================================================================*/
27500   PUBLIC void panic(format, num)
27501   char *format;                   /* format string */
27502   int num;                        /* number to go with format string */
27503   {
27504   /* Something awful has happened.  Panics are caused when an internal
27505    * inconsistency is detected, e.g., a programming error or illegal value of a
27506    * defined constant.
27507    */
27508
27509     if (panicking) return;        /* do not panic during a sync */
27510     panicking = TRUE;             /* prevent another panic during the sync */
27511     printf("File system panic: %s ", format);
27512     if (num != NO_NUM) printf("%d",num);
27513     printf("\n");
27514     (void) do_sync();             /* flush everything to the disk */
```



```
27515     sys_abort(RBT_PANIC);
27516   }


27519   /*===========================================================================*
27520    *                              conv2                                        *
27521    *===========================================================================*/
27522   PUBLIC unsigned conv2(norm, w)
27523   int norm;                       /* TRUE if no swap, FALSE for byte swap */
27524   int w;                          /* promotion of 16-bit word to be swapped */
27525   {
27526   /* Possibly swap a 16-bit word between 8086 and 68000 byte order. */
27527
27528     if (norm) return( (unsigned) w & 0xFFFF);
27529     return( ((w&BYTE) << 8) | ( (w>>8) & BYTE));
27530   }


27533   /*===========================================================================*
27534    *                              conv4                                        *
27535    *===========================================================================*/
27536   PUBLIC long conv4(norm, x)
27537   int norm;                       /* TRUE if no swap, FALSE for byte swap */
27538   long x;                         /* 32-bit long to be byte swapped */
27539   {
27540   /* Possibly swap a 32-bit long between 8086 and 68000 byte order. */
27541
27542     unsigned lo, hi;
27543     long l;
27544
27545     if (norm) return(x);                  /* byte order was already ok */
27546     lo = conv2(FALSE, (int) x & 0xFFFF);  /* low-order half, byte swapped */
27547     hi = conv2(FALSE, (int) (x>>16) & 0xFFFF);    /* high-order half, swapped */
27548     l = ( (long) lo <<16) | hi;
27549     return(l);
27550   }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                              src/fs/putk.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

27600   /* FS must occasionally print some message.  It uses the standard library
27601    * routine prink().  (The name "printf" is really a macro defined as "printk").
27602    * Printing is done by calling the TTY task directly, not going through FS.
27603    */
27604
27605   #include "fs.h"
27606   #include <minix/com.h>
27607
27608   #define BUF_SIZE          100   /* print buffer size */
27609
27610   PRIVATE int buf_count;          /* # characters in the buffer */
27611   PRIVATE char print_buf[BUF_SIZE];       /* output is buffered here */
27612   PRIVATE message putch_msg;      /* used for message to TTY task */
27613
27614   FORWARD _PROTOTYPE( void flush, (void)                                );
```

```
27615
27616   /*===========================================================================*
27617    *                                putk                                       *
27618    *===========================================================================*/
27619   PUBLIC void putk(c)
27620   int c;
27621   {
27622   /* Accumulate another character.  If 0 or buffer full, print it. */
27623
27624     if (c == 0 || buf_count == BUF_SIZE) flush();
27625     if (c == '\n') putk('\r');
27626     if (c != 0) print_buf[buf_count++] = c;
27627   }


27630   /*===========================================================================*
27631    *                                flush                                      *
27632    *===========================================================================*/
27633   PRIVATE void flush()
27634   {
27635   /* Flush the print buffer by calling TTY task. */
27636
27637
27638     if (buf_count == 0) return;
27639     putch_msg.m_type = DEV_WRITE;
27640     putch_msg.PROC_NR  = 1;
27641     putch_msg.TTY_LINE = 0;
27642     putch_msg.ADDRESS  = print_buf;
27643     putch_msg.COUNT = buf_count;
27644     call_task(TTY, &putch_msg);
27645     buf_count = 0;
27646   }
```