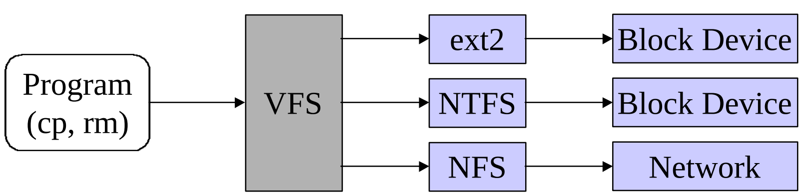
Lecture Overview

- Linux filesystem
  - Linux virtual filesystem (VFS) overview
    - Common file model
      - Superblock, inode, file, dentry
    - Object-oriented
  - Ext2 filesystem
    - Disk data structures
      - Superblock, block group, inodes
    - Memory data structures
    - Disk space management

The Linux Virtual Filesystem

- Virtual filesystem (VFS)
  - Provides an abstraction layer between the application program and the filesystem implementations
  - Provides support for many different kinds and types of filesystems
    - Disk-based, network, and special filesystems

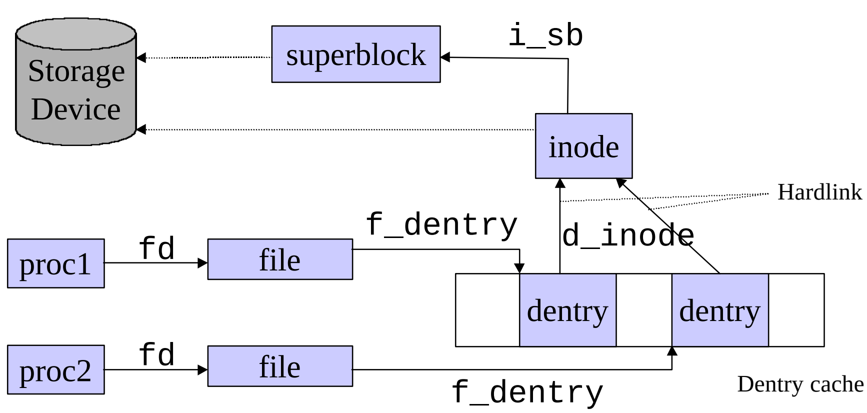


The Common File Model

- VFS introduces a *common file model* to represent all supported filesystems
- The common file model is specifically geared toward Unix filesystems, all other filesystems must map their own concepts into the common file model
  - For example, FAT filesystems do not have inodes
- The main components of the common file model are
  - *superblock* (information about mounted filesystem)
  - *inode* (information about a specific file)
  - *file* (information about an open file)
  - *dentry* (information about directory entry)

Common File Model Objects

- Interaction among objects



Object-Oriented Approach of VFS

- Each concept object has a set of defined operations that can be performed on the object (i.e., methods)
- VFS provides certain generic implementations for some operations
- Specific filesystem implementations must provide implementation specific operations definitions (i.e., inheritance and method overloading)
- There are no objects in C, though, so a table of function pointers is used for each object to provide its own version of the specific operations

Processes and Associated Files

- Each process has its own current working directory and its own root directory, this is stored in an `fs_struct` in the `fs` field of the process descriptor
- The open files of a process are stored in a `files_struct` in the `files` field of the process descriptor
  - When performing an `open()` system call, the file descriptor is actually an index into an array of the `file` objects in the `fd` array field of the process descriptors `files` field
    - For example, `current->files->fd[1]` is standard output for the process

The Ext2 Filesystem

- The first versions of Linux used the *Minix* filesystem
- Linux later introduced the *Extended Filesystem*, which as an improvement but offered unsatisfactory performance
- The *Second Extended Filesystem (Ext2)* was introduced in 1994

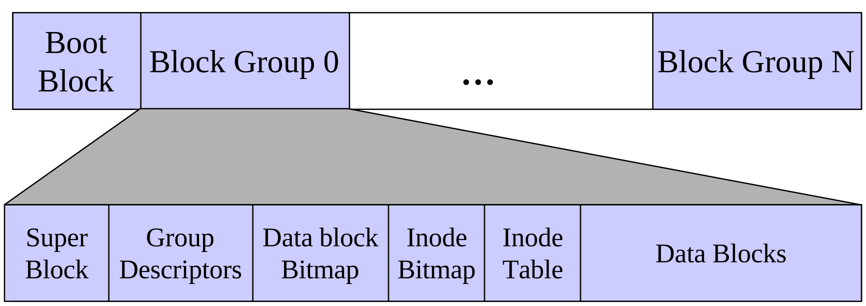
The Ext2 Filesystem Characteristics

- Configurable block size from 1024 to 4096 bytes
- Configurable number of inodes
- Partitions blocks into groups, where data blocks and inodes are stored in adjacent tracks
- Pre-allocates data blocks to regular files before they are used
- Supports "fast" symbolic links
- Implemented for robustness when updating disk structures
- Supports automatic consistency checking
- Supports immutable and append-only files

Ext2 Disk Data Structures

- The first block in all Ext2 partitions is always reserved for the boot sector
- The remainder of the partition is split into *block groups*
  - All block groups are the same size and are stored sequentially on the disk
  - Block groups reduce file fragmentation, since the kernel tries to keep the data blocks belonging to a file in the one block group if possible
  - The next slide illustrates the block group structure

Block Group Disk Data Structure



- It should come as no surprise that the VFS concepts map easily to the Ext2 structure
- Only the superblock and group descriptors in block group 0 are used by the kernel
- Block group size depends on partition and block size
  - 8GB partition with 4KB block, has 32k bits in block bitmap or 128MB; therefore 64 block groups are needed

Superblock Disk Data Structure

- The superblock is stored in an `ext2_super_block` structure
- Contains
  - Total number of inodes
  - Filesystem size in blocks
  - Free block counter
  - Free inode counter
  - Block size
  - Blocks per group
  - Inodes per group
  - 128-bit filesystem identifier
  - Mount counter
  - etc.

Group Descriptor Disk Data Structure

- Each block group has its own group descriptor, an `ext2_group_desc` structure
- Contains
  - Block number of block bitmap
  - Block number of inode bitmap
  - Block number of first inode table block
  - Number of free blocks in group
  - Number of free inodes in group
  - Number of directories in group
  - etc.

Inode Table Disk Data Structure

- The inode table consists of a series of consecutive blocks, each packed with inodes of the structure `ext2_inode`
- All inodes are the same size (128 bytes in Linux 2.2)
- An inode contains
  - File type and access rights
  - Owner and group identifiers
  - File length in bytes
  - Number of data blocks in the file
  - Various timestamp attributes
  - An array of (usually 15) data block pointers
  - etc.

Example Inode File Types

- Regular file
  - Need data blocks when it starts to have data
- Directory file
  - Special kind of file whose data blocks store filenames with corresponding inode numbers (actually it contains structures of type `ext2_dir_entry_2`)
    - Each directory structure contains inode number, entry length, name length, file type, and file name
    - Variable length structure, padded to be a multiple of 4
- Symbolic link
  - Up to 60 characters are stored in the data block pointer array of the inode structure for “fast” symbolic links
  - If longer than 60 characters, then a data block is required

Ext2 Memory Data Structures

- For efficiency, most information stored in disk data structures is copied into RAM when the filesystem is mounted
- Consider how often data structures change
  - Whenever a new file is created
  - Whenever a file needs more disk blocks
  - Whenever access times need to be updated
- Some in-memory data structures differ from on-disk data structures

Ext2 Memory Data Structures

Corresponding data structures and caching policies

Type	***Disk structure***	***Memory structure***	***Caching***
Superblock	`ext2_super_block`	`ext2_sb_info`	Always
Group descriptor	`ext2_group_desc`	`ext2_group_desc`	Always
Block bitmap	Bit array in block	Bit array in buffer	Fixed
Inode bitmap	Bit array in block	Bit array in buffer	Fixed
Inode	`ext2_inode`	`ext2_inode_info`	Dynamic
Data block	Unspecified	Buffer	Dynamic
Free inode	`ext2_inode`	None	Never
Free block	Unspecified	None	Never

Superblock Memory Data Structure

- An `ext2_sb_info` structure pointer is placed in the VFS superblock data structure when an Ext2 filesystem is mounted
  - This memory data structure contains most of the information from the disk data structure for the Ext2 superblock
  - Contains data related to mount state, options, etc.
  - Also contains a block bitmap cache and an inode bitmap cache
    - It is not feasible to keep all disk bitmaps in memory, so it is necessary to cache some and leave the rest on disk
    - Uses a LRU algorithm over (usually) 8 cache entries

Inode Memory Data Structure

- An `ext2_inode_info` structure pointer is placed in the VFS inode data structure
  - Contains most of the fields in the Ext2 disk inode structure
  - Information for block preallocation
  - Flag to indicate whether I/O operations should be done synchronously

Ext2 Operations

- Ext2 superblock operations
  - Essentially, specific implementations are provided for all VFS operations (except 2)
- Ext2 inode, directory, and file operations
  - Many operations have specific implementations, but in many cases the generic VFS operations are sufficient

Creating a Filesystem

- Ext2 filesystems are created with the utility program `/sbin/mke2fs`
  - Default options: block 1024 bytes, one inode for each group of 4096 bytes, 5% reserved blocks
  - It performs these actions
    - Initializes superblock and group descriptors
    - Creates a list of defective blocks
    - For each block group, reserves all blocks needed to store superblock, descriptors, bitmaps, and inode table
    - Initializes all bitmaps to zero
    - Initializes all inode tables
    - Creates root directory
    - Creates lost+found directory
    - Updates inode bitmap and data bitmap of block group where the above directories were added
    - Groups defective blocks in the lost+found directory

Creating a Filesystem

- Consider a filesystem created on a 1.4MB floppy disk
  - A single group descriptor is sufficient, 72 (5% of 1440) reserved blocks, 360 inodes in 45 blocks

Block	***Content***
0	Boot block
1	Superblock
2	Block containing single block group descriptor
3	Data block bitmap
4	Inode bitmap
5-49	Inode table (inodes up to 10 are reserved, inode 11 is lost+found)
50	Root directory
51	lost+found directory
52-62	Reserved blocks preallocated for lost+found directory
63-1439	Free block

Ext2 Managing Disk Space

- The goals for disk space management are twofold
  - Make every effort to avoid file fragmentation
    - Increases average time of file operations
    - Similar problems as associated with memory allocation
  - Make every effort to be time-efficient
    - Conversion between file offset and logical block number must be performed quickly
    - Need to limit accesses to disk data structures

Ext2 Managing Disk Space

- Allocating inodes
  - Occurs in `ext2_new_inode()`
  - Requires the parent inode and the mode (i.e., type) of the file to be created
  - If the inode is for a directory
    - Forward search from the parent’s block group for a block group with free space and a low directory-to-inode ratio
    - If that fails, searches for block groups with above average free space and chooses the one with the fewest directories
  - If the inode is for any other type
    - Forward search from the parent’s block group for a free inode
  - Updates inode bitmap, decrements inode counters, puts the inode into the superblock’s dirty list

Ext2 Managing Disk Space

- Releasing inodes
  - Occurs in ext2_free_inode()
  - Requires inode to deallocate
  - Is called after inode is removed from the inode has table, after the last hard link has been deleted, and after the file is truncated to 0
  - Computes the index of the block group using the inode number and number of inodes per block group
  - Releases all pages in the page cache associated with inode (e.g., for memory mapped I/O)
  - Updates inode bitmap, increments inode counters, puts the inode into the superblock’s dirty list

Ext2 Managing Disk Space

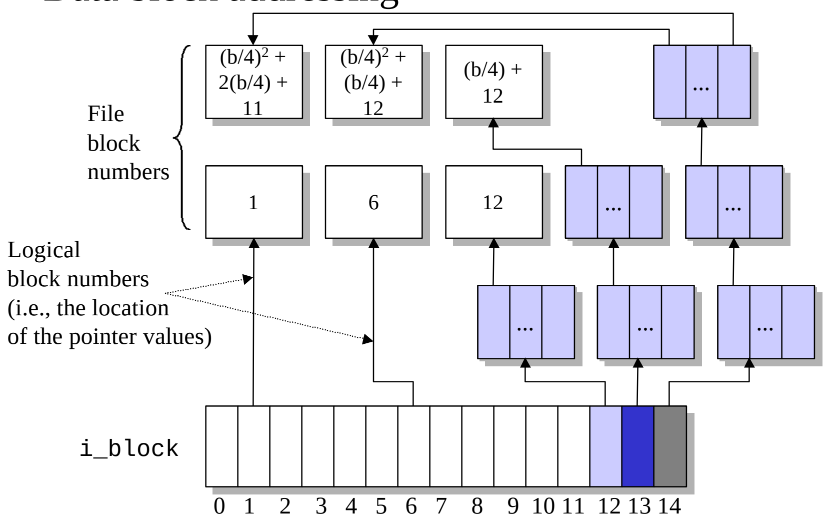
- Data block addressing
  - A non-empty regular file consists of a group of data blocks
    - The blocks can be referred to by their relative position inside the file (*file block number*) or their position inside the disk partition (*logical block number*)
  - Deriving the logical block number from an offset *f* inside a file is a two-step process
    - Derive from *f* the file block number
      - This is easy, divide *f* by block size and round down to an integer
    - Translate the file block number to the logical block number
      - This is not so easy

Ext2 Managing Disk Space

- Data block addressing
  - Recall that an inode has an array of 15 block pointer
  - The first 12 entries actually point to data blocks
  - The 13th entry points a disk block that contains pointers to data blocks for the file, i.e., a single level of indirection
  - The 14th entry points to a disk block that contains pointers to disk blocks that contain pointers to data blocks, i.e., two levels of indirection
  - The 15th entry points to a disk block that contains pointers to disk blocks that contain pointers to disk blocks that contain pointers data blocks, i.e., three levels of indirection
  - Use an algorithm to convert the file block number into the indices (i.e., logical block number) to find the physical block

Ext2 Managing Disk Space

- Data block addressing



Ext2 Managing Disk Space

- Allocating data blocks
  - Occurs in ext2_getblk()
  - Requires an inode for the request and a “goal”
    - The goal is a preferred logical block number
    - The preferred logical block number is the previously allocated block number plus one or any of the previously allocated block numbers plus one or a logical block number in the inode’s block group
      - This is an attempt to reduce file fragmentation
  - Performs pre-allocation of blocks
  - Updates the various bookkeeping records

Ext2 Managing Disk Space

- Releasing data blocks
  - Occurs in ext2_truncate()
  - Requires an inode
  - Walks i_block to get all of the data blocks to free them
  - Updates the various bookkeeping records