Lecture Overview

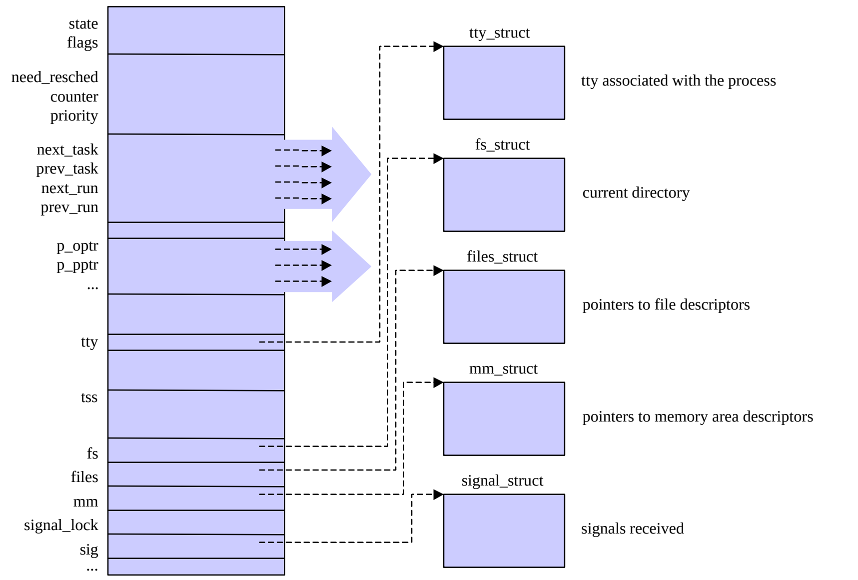
- Overview of Linux processes
  - Based on version 2.2 of the Linux kernel
  - Introduce process properties
  - Introduce kernel process structures
  - Discuss process creation and destruction

- A closer examination of these topics should be helpful as you start to delve deeper into the kernel in your programming assignments

Linux Processes

- Linux also refers to a process as a "task"
- Linux represents each process as a process descriptor of type `task_struct`
  - Contains all information related to a single process
  - Not all information is contained directly in the `task_struct`, instead it includes pointers to other data structures, which may point to other data structures, and so on
- Each process has its own process descriptor
  - Because of this strict one-to-one relationship, process descriptor addresses uniquely identify process (*process descriptor pointer*)

Linux Process Descriptor



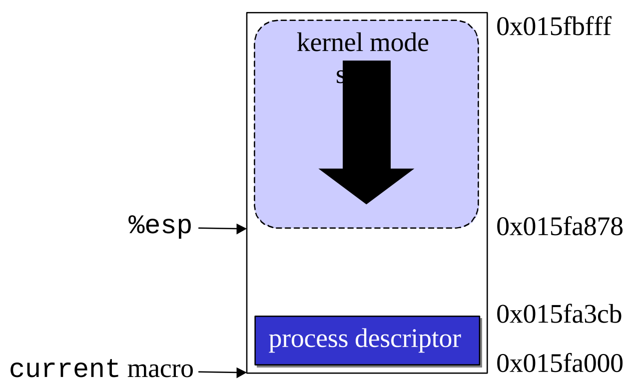
Linux Process State

- The state field of the process descriptor
  - Describes what is currently happening to the process
  - Consists of an array of mutually exclusive flags
  - Possible states include
    - `TASK_RUNNING` - running to waiting to run
    - `TASK_INTERRUPTIBLE` - suspended
    - `TASK_UNINTERRUPTIBLE` - suspended
    - `TASK_STOPPED` - execution has stopped
    - `TASK_ZOMBIE` - terminated

Linux Task Array

- All process descriptors are contained a global task array in kernel address space, called `task`
  - The elements of the `task` array are pointers to process descriptors; null indicates an unused entry
  - As a result using an array of pointers, process descriptors are stored in dynamic memory rather than permanent kernel memory
- Each `task` array entry actually contains two different data structures in a single 8 KB block for each process
  - A process descriptor and the kernel mode stack
  - These are cached after use to save allocation costs

Linux Task Array Entry



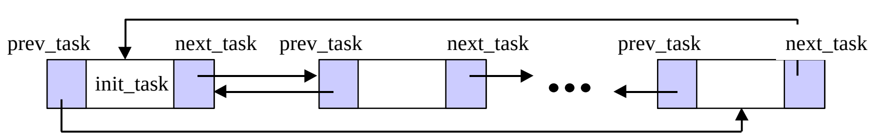

```
union task_union {
    struct task_struct task;
    unsigned long stack[2048];
};
```

Linux Task Array Entry

- The pairing of the processor descriptor and the kernel mode stack offers some benefits
  - The kernel can easily obtain the process descriptor pointer of the currently executing process from the value of the `%esp` register
    - The memory block is 8 KB or 2^{13} bytes long, so all the kernel has to do is mask out the least significant 13 bits of `%esp` to get the process descriptor pointer, this is done by the current macro
    - You might see the macro used inline like, `current->pid`
  - The pairing is also beneficial when using multiple processors since the current process for each is determined similarly

Linux Process Lists

- Linux maintains many different lists of processes for many different purposes
- Process list
  - The process list contains all existing process descriptors
  - It is a circular doubly linked list
  - The head of the list is the `init_task` descriptor, which is the first element of the last array (process 0 or swapper process)
  - The macros `SET_LINKS`/`REMOVE_LINKS` modify the list



Linux Process Lists

- Running list
  - The OS often looks for a new process to run on the CPU
  - It is possible to scan the entire process list for processes in the `TASK_RUNNING` state, but this is inefficient
  - The OS maintains a *runqueue* of all `TASK_RUNNING` processes
  - This list is a circular doubly linked list like the process list and has the `init_task` process descriptor as its head also
  - `add_to_runqueue()`/`del_from_runqueue()` modify the list
  - `wake_up_process()` makes a process runnable

Linux Process Lists

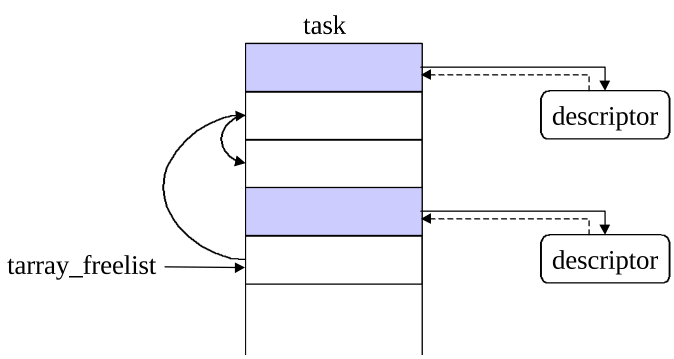
- PID hash table
  - Each process has an associated *process identifier (PID)*, which users use to identify a process
    - There is a `pid` field in the process descriptor
    - A PID is a number from 0 to 32767
  - For efficient look up of processes by PID, the OS maintains a PID hash table
  - The hash table using chaining to handle collisions, so each entry in the hash table forms a doubly linked list
    - The fields are `pidhash_next` and `pidhash_previous` in the process descriptor
  - `hash_pid()`/`unhash_pid()` modify the hash table, `find_task_by_pid()` searches the hash table

Linux Process Lists

- List of free task entries
  - The `task` array entries are used and freed every time a process is created or destroyed
  - A list of free `task` array entries is maintained for efficiency starting with the `tarray_freelist` variable
  - Each free entry in the `task` array points to another free entry, while the last entry points to null
  - Destroying a process puts its entry at the head of the list
  - Each process descriptor also contains a pointer to its entry in the `task` array to make deletion more efficient

Linux Process Lists

- List of free task entries
  - `get_free_taskslot()`/`add_free_taskslot()` modify the list

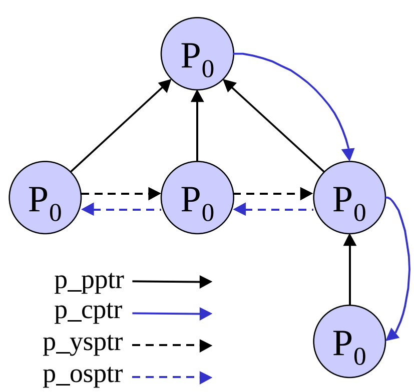


Linux Process Lists

- Parent/child relationships
  - Each process descriptor maintains a pointer to its parent, sibling, and child process descriptors
    - `p_opptr` (*original parent*) points to the creating process or the *init* process (process 1) if the parent has terminated
    - `p_pptr` (*parent*) coincides with p_opptr except in some cases, such as when another process is monitoring the child process
    - `p_cptr` (*child*) points to the process' youngest child
    - `p_ysptr` (*younger sibling*) points to the process' next younger sibling
    - `p_osptr` (*older sibling*) points to the process' next older sibling

Linux Process Lists

- Parent/child relationships



Linux Process Lists

- The runqueue groups processes in state TASK_RUNNING
- Processes in state TASK_STOPPED or TASK_ZOMBIE are not linked in specific lists since there is no need
- Processes in TASK_INTERRUPTIBLE and TASK_UNINTERRUPTIBLE are divided into many classes of list, these lists are *wait queues*

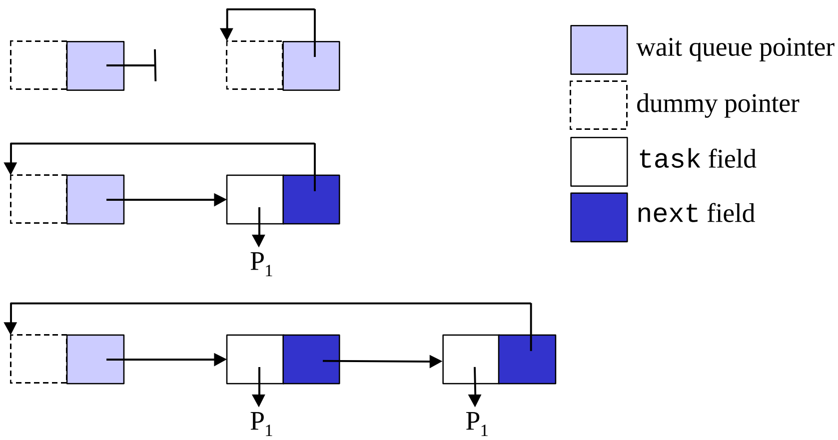
Linux Process Lists

- Wait queues have several uses in the kernel where processes must wait for some event to occur
  - Interrupt handling, process synchronization, timing
- Wait queues implement conditional waits on events
  - A specific queue is for a specific type of event
- A wait queue a structure and wait queues are identified by a *wait queue pointers*

```
struct wait_queue {
    struct task_struct *task;
    struct wait_queue *next;
};
```

Linux Process Lists

- Wait queues are somewhat complex, because they use a dummy pointer for efficiency



Linux Process Lists

- `init_waitqueue()` initializes a wait queue pointer, modifying the pointer to point to the dummy address
- `add_wait_queue()`/`remove_wait_queue()` modify the wait queue
- To wait on a specific wait queue, call `sleep_on()`

```
void sleep_on(struct wait_queue **p) {
    struct wait_queue wait;
    current->state = TASK_UNINTERRUPTIBLE;
    wait.task = current;
    add_wait_queue(p, &wait);
    schedule();
    remove_wait_queue(p, &wait);
}
```

  - Similarly for `interruptible_sleep_on()`, `sleep_on_timeout()`, and `interruptible_sleep_on_timeout()`

Linux Process Usage Limits

- All processes have an associated set of *usage limits*
- Linux recognizes the following limits
  - RLIMIT_CPU, RLIMIT_FSIZE, RLIMIT_DATA, RLIMIT_STACK, RLIMIT_CORE, RLIMIT_RSS (page frames), RLIMIT_NPROC, RLIMIT_NOFILE, RLIMIT_MEMLOCK, and RLIMIT_AS (address space)
- Process limits are stored in the `rlim` field of the process descriptor; `rlim` is an array of `rlimit`

```
struct rlimit {
    long rlim_cur;
    long rlim_max;
};
```

  - To check a limit, `current->rlim[RLIMIT_CPU].rlim_cur`
  - Most limits are set to `RLIMIT_INFINITY`

Linux Process Creation

- When creating a process, most Unix-based operating systems create the child process as a copy of the parent process
  - This is inefficient
- Linux makes process creating more efficient using three different mechanisms
  - Copy-on-write
  - Lightweight processes
  - `vfork()` system call

Linux Process Creation

- Linux creates lightweight processes using the `__clone()` function
  - Is actually a wrapper for a hidden `clone()` function
  - It takes four parameters, a function to execute, an argument pointer, sharing flags, and the child stack
  - Both fork() and vfork() are implemented in Linux using clone() using different parameters

Linux Process Creation

- Kernel threads
  - Traditional Unix systems delegate some tasks to intermittently running processes
    - Flushing disk caches, swapping out unused page frames, servicing network connections, etc.
  - It is more efficient to service these tasks asynchronously
  - Since many of these tasks can only run in kernel mode, Linux introduces the notion of *kernel threads*
    - Each kernel thread executes a single specific kernel function
    - Each kernel thread only executes in kernel mode
    - Each kernel thread has a limited address space

Linux Process Creation

- Kernel threads
  - `kernel_thread()` is used to create a kernel thread

```
int kernel_thread(int (*fn)(void *), void *arg,
                  unsigned long flags)
{
    pid_t p;
    p = clone(0, flags | CLONE_VM);
    if (p)
        return p;
    else {
        fn(arg);
        exit();
    }
}
```

Linux Process Creation

- Process 0 (*swapper process*)
  - Is a kernel thread that is the ancestor of all processes
  - It is created from scratch during the initialization phase of Linux by the `start_kernel()` function
  - start_kernel() initializes all data structures needed by the kernel, enables interrupts, and creates an additional kernel thread, process 1 (*init process*)
  - After creating the init process, the swapper process executes `cpu_idle()`, which essentially executes `hlt` assembly instructions repeatedly
    - The swapper process is only selected when there are no other processes in `TASK_RUNNING` state

Linux Process Creation

- Process 1 (*init process*)
  - The init process initially shares all per-process data structures with the swapper process
  - The init process, once scheduled, starts executing the `init()` function
  - `init()` process creates four more kernel threads to flush dirty disk buffers, swap out pages, and reclaim memory
  - Then `init()` invokes `execve()` to load the executable *init* program; at this point the init process becomes a regular process
  - The init process never terminates

Linux Process Destruction

- Processes die when the explicitly call `exit()`, when they complete `main()`, or when a signal is not or cannot be handled
- `do_exit()` handles process termination by removing most references in the kernel to the process
  - Updates process status flag, removes process from any queues, releases data structures, set the exit code, updates parent/child relationships, invokes the scheduler to select another process for execution
- Child processes become children of init process

Linux Process Switching

- Hardware context
  - Linux must save/reload CPU registers when switching processes
  - Some information is stored in the kernel mode stack, other information is stored in the *Task State Segment* (TSS)
- Hardware support
  - The Intel 80x86 architecture includes the TSS used specifically to store the hardware context
- Linux code
  - The *switch_to* macro actually performs the process switch
- Saving floating point registers
  - There is hardware support to lazily save floating point registers, i.e., they are only saved when necessary

Changes in Linux 2.4

- There is no longer a `tasks` array; this raises the previously hard-coded limit on the number of processes
- Wait queues are enhanced and now use a more generic `list_head` data type to create lists
- `clone()` now allows you to clone the parent PID
- Process switching data is now stored more fully in the process descriptor data structure