# Operating Systems II

Internal Kernel Data Structures

# Overview

- Look at some of the internal data structures that make up a kernel.
  - In particular examine in detail the information you may want to record for each task in a system
- High level look at some of the main algorithms used inside the kernel.
- Building block for later examining each of the different subsystems of the kernel.

# Task Structure

- `state (typically an int/long)`
  - current state of the process
  - RUNNING – currently executing.
  - READY – capable of executing, waiting for time slice.
  - INTERRUPTIBLE - waiting for an external event. Can be reactivated by a signal.
  - UNINTERRUPTIBLE - waiting for an external event.
  - STOPPED - process is halted.
  - ZOMBIE - terminated process, but must be kept in the process table.

# Task Structure (cont.)

- `flags (typically int/long)`
  - bit mask of the system status. (STARTING, EXITING, SUPERPRIV, DUMPCORE, SIGNALED, MEMALLOC, VFORK, USEDFPU)
  - flags used for the accounting of processes and do not influence the mode of system operation.
    - Useful for profiling applications!

# Task Structure (cont.)

- `ptrace (typically int/long)`
  - indicates the process is being monitored by another process. (PTRACED, TRACESYS)
- `sigpending (typically int)`
  - set when a signal is handed over to this process.
- `addr_limit (memory struct)`
  - describes the address space that is possible to access.

# Task Structure (cont.)

- `exec_domain (struct)`
  - description of another platform that is emulated to execute the process.
- `lock (struct)`
  - synchronization lock for the task structure.
- `rt_priority (typically int)`
  - priority of task.
- `nice (typically int)`
  - nice value for the process.

# Task Structure (scheduling)

- `counter (typically long)`
  - contains the time in clock cycles the process can run before scheduling. Next process chosen by task with highest counter value.
- `policy (typically int)`
  - scheduling policy to use (ROUND_ROBIN, FIFO, PRIORITY).
  - Allows multiple scheduling algorithms to be used in the operating system.

# Task Structure (signals)

- `blocked (typically long)`
  - bit mask of the signals blocked for the process.
- `sig (struct *)`
  - refers to the corresponding signal handling routines.
- `pending (struct)`
  - list of signals that are pending service.

# Task Structure (process relations)

- The kernel often uses a doubly linked list of processes.
  - `run_list (struct *)`
- `next_task (struct *)`
  - next process
- `prev_task (struct *)`
  - previous process

# Task Structure (process relations)

- Family relations between tasks are recorded via references.
  - `p_opptr` - original parent
  - `p_pptr` - parent
  - `p_cptr` - youngest child
  - `p_ysptr` - younger sibling
  - `p_osptr` - older sibling
- Using these references you can traverse ALL levels of related tasks!

# Task Structure (Memory)

- `mem_manage (struct *)`
  - describes the positions of all the parts of process in memory.
  - `start_code, end_code, start_data, end_data;`
    - Position of program
  - `start_stack, start_mmap`
    - Position of execution info
  - `arg_start, arg_end, env_start, env_end`
    - Program argument/environment information

# Task Structure (process id)

- Processes in system require some unique identifier for referencing the process.
  - `pid` - process id
  - `pgrp` - process group
  - `tgid` - thread group id (parent pid)
- These values are typically integers.

# Task Structure (user/group ids)

- Used to identify ownership of the process and permissions
- More user/group id types increase flexibility – also increases complexity.
  - `uid, gid` - plain user/group id
  - `euid, egid` - effective user/group id
  - `suid, sgid` - set user/group id
  - `fsuid, fsgid` - filesystem user/group id

# Task Structure (files)

- `files_info (struct *)` contains:
  - `count`
    - reference count of processes pointing to structure
  - `mask`
    - permissions of new files that are created.
  - `root (struct *)`
    - root directory for the process. WHY?
  - `pwd (struct *)`
    - the working directory of the process.

# Task Structure (files cont.)

- `files (struct *)` - contains information of all the files open by the process.
  - `count` - count of references to file
  - `max_count` - maximum # of file references
  - `fd (struct *)` - file descriptors of open files.

# Task Structure (timing)

- `per_cpu_utime[NR_OF_CPUS]`
  - amount of user time executing per CPU
- `per_cpu_stime[NR_OF_CPUS]`
  - amount of system time executing per CPU
- `times (struct)`
  - sum of all execution time (including children)
- `start_time`
  - time when the process was generated

# Task Structure (timing cont.)

- Support of interval timers.
  - It contains values for when the timers are to be triggered and for the interval between them.
  - `it_value`
    - When the initial timer is triggered.
  - `it_incr`
    - Interval for subsequent timer is triggered.
- Can have multiple timers for flexibility.

# Task Structure (IPC)

- `semsleep (struct *)`
  - Reference to a semaphore the process is sleeping on.
- `semhold (struct *)`
  - list of semaphores occupied by process. Needed to be released when the process terminates.

# Task Structure (misc)

- `command`
  - Name of the program executable. (i.e. **a.out**)
- `dumpable`
  - Should the process do a memory dump on certain signals.

# Task Structure (misc cont.)

- `rlim[RLIM_NLIMITS]`
  - Resource limits for process.
    - Memory, CPU time, ...
  - Get/Set methods for each type of resource.
- `exit_code, exit_signal`
  - The exit code and signal that terminated the program.

# Process Table

- Tasks within the system are often organized via doubly-linked lists.
  - This permits moving the task structures between different lists that signify different meanings/states.
- Tasks can be accessed via the `next_task` and `prev_task` in the Task structure.

# Process Table (cont.)

- You can use the macro `init_task` to access the first task in the list.
- You can access the current task using the `get_current` macro.
  - Current task can be in any state!
- The maximum number of tasks in the system is restricted to some global value: `max_threads`.
  - This contributes towards some security.

# Files and Inodes

- `inode (struct)`
  - Contains view of the “system” on the file.
  - Exactly 1 inode for each file used.
    - Info includes: physical location; ...
- `file (struct)`
  - Contains view of a “process” on the file.
  - 1 struct for each file used.
    - Info includes: usage mode; current position in file; ...

# Inode Structure

- `device` - partition (device) identifier
- `location` - location in the partition
- `uid` - file ownership
- `gid` - group ownership
- `size` - size of file in bytes
- `mtime` - last modified time
- `atime` - last accessed time
- `ctime` - file creation time

# File structure

- `f_mode` - mode file was opened.
- `f_pos` - position for next operation.
- `f_count` - reference counter.
- `f_flags` - control file access.
- `fs_dentry (struct *)` - refers to directory.
- `f_operations (struct *)` - structure referring to all file operations.

# Dynamic Memory Management

- Memory is often managed on a page basis. Each page is $2^n$ bytes.
- To request a free page
  - `alloc_pages` or `get_free_pages`
- To free acquired pages
  - `free_pages`
- The recommened way of acquiring memory is to use `get_zeroed_page()`.
  - Why?

# Wait Queues

- All wait queues are cyclic and as such utilize the same underlying queue data structure
  - `struct list head {`
    `struct list_head *next, *prev}`
- Methods for adding and removing are synchronized.
  - `add_wait_queue`
  - `remove_wait_queue`

# Wait Queues (cont)

- Processes enter queue blocked on an event.
- There is a different queue for every event.
- Higher level functions include:
  - `sleep_on` - wait indefinitely
  - `sleep_on_timeout` - wait up to a given time
  - `interruptible_sleep_on` - wait indefinitely, but process can be interrupted
  - `uninterruptable_sleep_on_timeout` – wait up to a time period, but otherwise cannot be interrupted

# Wait Queues (cont.)

- Set the process state and then enter the queue.
- Processes can exit the queue via higher level functions
  - `wake_up` – remove process from queue, uninterruptable until completed.
  - `wake_up_interruptible` – remove process from queue, interruptable as soon as possible.

# Semaphores

- Kernel level semaphores, NOT for user programs :)
- `struct semaphore`
  - `count`
  - `list_head* wait`
- **Down** - if count <= 0, enter the wait queue, otherwise continue...
- **Up** - release semaphore and increment count

# System Timers

- `struct timer_list`
  - `struct list_head list` - list maintenance.
  - `long expires` - when the timer alarms.
  - `long data` - argument for the function.
  - `void (*function)(unsigned long)` - function to call on alarm.
- `add_timer`, `del_timer`, and `mod_timer` to manipulate list.

# Main Algorithms

- Signals
- Hardware Interrupts
- Software Interrupts
- Timer Interrupts
- Scheduler

# Signals

- Signals are used to inform processes about certain events.
  - Synchronization, abort, or simply change state...
- Signals typically sent through
  - `send_sig_info(int sig, struct siginfo *info, struct task *t)`
- `sig` – signal number
- `info` – additional details of signal (sender, ...)
- `t` – the task to signal.

# Signals (cont.)

- The kernel controls and protects who can signal.
- If the signal is ok, then it is passed to the task through the task structure (`pending` and `sigpending`)
- Though the task could block the signal through the task structure (`blocked`)

# Signals (cont.)

- The signal is NOT procesed immediately. It is dealt with when the process is moved into the run state.
- When the scheduler moves the task into the run state (and before returning to user mode), the routine `return_from_sys_call` checks for signals and if so calls `do_signal()` to perform the action.
- What if signal handler is a user defined function?

# Interrupts

- What is an interrupt?
- There are two common techniques an OS can employ to deal with interrupts.
- **Technique #1:** The system halts until ALL aspects of the interrupt are handled immediately.
  - System is uninterruptible during full processing

# Interrupts (cont.)

- **Technique #2:** Interrupt is split into 2 stages
  - Hardware interrupt – Stage 1: On receiving an interrupt, any immediate actions necessary are taken. (uninterruptible)
  - Software interrupt - Stage 2: processing the data of the interrupt is performed. (interruptible)

# Interrupts (cont.)

- Why the 2 techniques?
- Technique #1 provides a simple to implement solution.
  - Best choice for small kernels
- Technique #2 allows the system to better prioritize handling of interrupts.
  - Best choice for hard real-time or systems with lots of interrupts

# Timer Interrupts

- Internally the kernel keeps track of time in two formats.
  - **Ticks (long jiffies)** – needed for task scheduling
  - **Wall clock time (struct time)** – needed for interacting with user
- `do_timer` is the interrupt routine for time
  - Stage 1 updates jiffies
  - Stage 2 updates xtime
- What else must this interrupt routine perform?
  - Update time used per task; Update system load; …

# Scheduler

- The `schedule` routine performs several key operations:
  1. Store current process info and profiling information.
  2. Check for any pending stage 2 interrupts and process them.
  3. Determine the next process to schedule.
  4. Make the next process the current one.