Operating Systems
COMS W4118
Lecture 20

Alexander Roth

2015 – 04 – 08

1 Examining the Kernel Memory

- We are now in the process stack for the kernel
- Suppose we have four processes running.
- Each process has a task struct that is a linked list connecting it to the next process.
- You can only have a fixed number of processes.
- There is also a *run queue* that links together some of the processes.
- If a processes are in the run queue, their state is set to `TASK_RUNNING`, which is the value of 0.
- `TASK_RUNNING` means the process is eligible to run, not that it is currently running.
- Run queues in 2.6 are organized into a huge number of queues, with a given priority, so the big O notation is constant time.
- In 3.x, we use a CFS algorithm (Completely Fair Scheduler). It tries to give more priority on I/O intensive processes.
- Think of the run queue as linking all the processes that are able to run.

2 Task States

- Each task struct has an integer field called `tick`, which is how many clock ticks it has gone through.
- We can assign a timeout value to the amount of ticks that a process can go through.

- There is a flag within the `task_struct` called `NEED_RESCHED`, which is set whenever a timeout occurs.
- Every interrupt has a common routine when it is about to return.
- The common routine is right before the system goes back in the user mode.
- We check the `NEED_RESCHED` flag.
- We call the function `schedule()` to switch tasks.
- Eventually, `schedule()` calls a function called `context_switch()`
- Context switch is system-dependent, and it saves the current process information and then go to the next task that was picked to run.
- We have a list of runnable processes that the system will look through with the scheduling algorithm to determine which process to run next.
- Timer interrupts have the ability to preempt other processes.
- A running process can go to sleep and enter the sleeping state.
- When does a running process enter into a blocked sleeping state?
- There are certain situations when the kernel makes processes uninterruptible.
- A blocking system call sends the running task into a sleeping process.
- Suppose we are blocking a task that is reading from a keyboard.
- The read system call would set the task to interruptible while it waits for the keyboard.
- There must be some data structure that knows what process task struct is waiting on it.
- That is why there is a wait queue.
- When the system receive information from the keyboard, the task struct will be woken up from the wait queue on `wake_up()`
- We run the command `try_to_wake_up()`.
- When an interruptible process gets a signal, it wakes up but a field is set to go to the signal handler.