CS 423Operating System Design: Introduction to Linux Kernel Programming (MP2 Walkthrough)

Jack Chen

Some content taken from a previous year's walkthrough by Prof. Adam Bates

Purpose of MP2

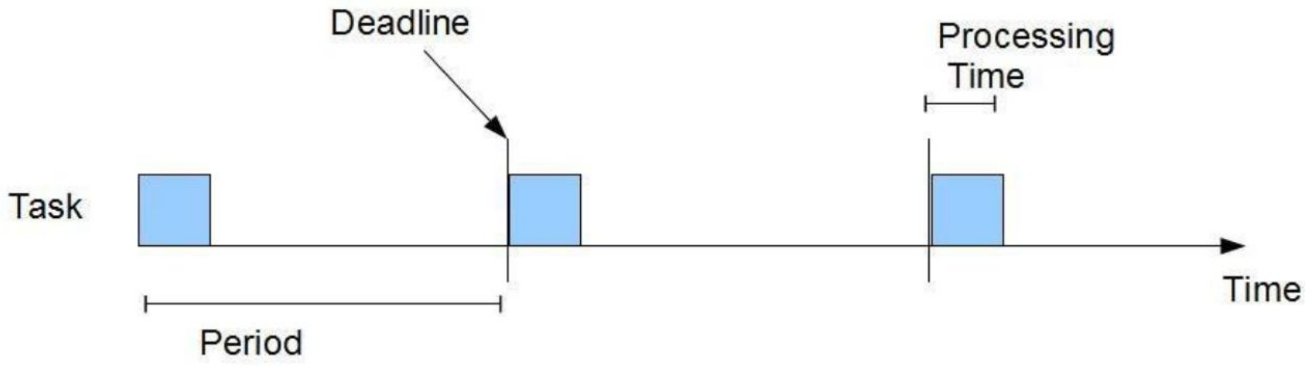
- Understand real time scheduling concepts
- Design a real time schedule module in the Linux kernel
- Learn how to use the kernel scheduling API, timer, procfs
- Test your scheduler by implementation a user level application

Introduction

- Real-time systems have requirements in terms of response time and predictability
  - Air bag in a car
  - Video surveillance systems
- We will be dealing with periodic tasks
  - Constant period
  - Constant running time
- We will assume tasks are independent

Periodic Tasks Model

- Liu and Layland [1973] model, each task i has
  - Period P_i
  - Deadline D_i
  - Runtime C_i



Rate Monotonic Scheduler (RMS)

- A static scheduler has complete information about all the incoming tasks
  - Arrival time
  - Deadline
  - Runtime
  - Etc.

- RMS assigns higher priority for tasks with higher rate/shorter period
  - Shorter period results higher priority
  - It always picks the task with the highest priority
  - It is preemptive

MP2 Overview

- We will implement RMS with an admission control policy as a kernel module

- The scheduler provides the following interface
  - Registration: save process info like pid, etc.

  - Yield: process notifies RMS that it has completed its period

  - De-Registration: process notifies RMS that it has completed all its tasks

Admission Control

- We only register a process if it passes admission control
- The module will answer this question every time:
  - Can the new set of processes still be scheduled on a single processor?
  - Yes if and only if:

$$\boxed{\sum_{i \in T} \frac{C_i}{P_i} \leq 0.693}$$

  - Always assumes that

$$C_i < P_i$$

  - Ci is the runtime of a task
  - Pi is the period to deadline

Admission Control

Floating point operations are very expensive in the kernel.
You should NOT use them.

Instead use Fixed-Point arithmetic.

MP2 User Process Behavior

```
void main (void)
{
	// Proc filesystem
	Register(PID, Period, ProcessTime);
	// Proc filesystem: Verify the process was admitted
	List = Read_Status();
	if (! process in the list) exit(1);

	Yield(PID); // Send yield to Proc filesystem
	while (exist jobs)
	{
		//wakeup_time = t0 - gettimeofday() and factorial
computation
		do_job();
		Yield(PID); // Send yield to Proc filesystem
	}
	Unregister(PID); // Send yield to Proc filesystem
}
```

MP2 User Process Behavior

```
void main (void)
{
	// Proc filesystem
	Register(PID, Period, ProcessTime);
	// Proc filesystem: Verify the process was admitted
	List = Read_Status();
	if (! process in the list) exit(1);

	Yield(PID); // Send yield to Proc filesystem
	while (exist jobs)
	{
		//wakeup_time = t0 - gettimeofday() and factorial
computation
		do_job();
		Yield(PID); // Send yield to Proc filesystem
	}
	Unregister(PID); // Send yield to Proc filesystem
}
```

MP2 User Process Behavior

```
void main (void)
{
        // Proc filesystem
        Register(PID, Period, ProcessTime);
        // Proc filesystem: Verify the process was admitted
        List = Read_Status();
        if (! process in the list) exit(1);

        Yield(PID); // Send yield to Proc filesystem
        while (exist jobs)
        {
                //wakeup_time = t0 - gettimeofday() and factorial
computation
                do_job();
                Yield(PID); // Send yield to Proc filesystem
        }
        Unregister(PID); // Send yield to Proc filesystem
}
```

MP2 User Process Behavior

```
void main (void)
{
	// Proc filesystem
	Register(PID, Period, ProcessTime);
	// Proc filesystem: Verify the process was admitted
	List = Read_Status();
	if (! process in the list) exit(1);

	Yield(PID); // Send yield to Proc filesystem
	while (exist jobs)
	{
		//wakeup_time = t0 - gettimeofday() and factorial
computation
		do_job();
		Yield(PID); // Send yield to Proc filesystem
	}
	Unregister(PID); // Send yield to Proc filesystem
}
```

MP2 User Process Behavior

```
void main (void)
{
        // Proc filesystem
        Register(PID, Period, ProcessTime);
        // Proc filesystem: Verify the process was admitted
        List = Read_Status();
        if (! process in the list) exit(1);

        Yield(PID); // Send yield to Proc filesystem
        while (exist jobs)
        {
                //wakeup_time = t0 - gettimeofday() and factorial
computation
                do_job();
                Yield(PID); // Send yield to Proc filesystem
        }
        Unregister(PID); // Send yield to Proc filesystem
}
```

MP2 User Process Behavior

```
void main (void)
{
        // Proc filesystem
        Register(PID, Period, ProcessTime);
        // Proc filesystem: Verify the process was admitted
        List = Read_Status();
        if (! process in the list) exit(1);

        Yield(PID); // Send yield to Proc filesystem
        while (exist jobs)
        {
                //wakeup_time = t0 - gettimeofday() and factorial
computation
                do_job();
                Yield(PID); // Send yield to Proc filesystem
        }
        Unregister(PID); // Send yield to Proc filesystem
}
```

MP2 Process State

- A process in MP2 can be in one of three states
  a. READY: a new job is ready to be scheduled
  b. RUNNING: a job is currently running and using the CPU
  c. SLEEPING: job has finished execution and process is waiting for the next period

- Those are states we should explicitly define in MP2 as they are specific to our scheduler.

MP2 Extending the PCB

We should extend PCB to hold MP2 specific information

```
struct mp2_task_struct {
   struct task_struct *linux_task;
   struct list_head task_node;
   struct timer_list tasl_timer;
   pid_t pid;
   unsigned long period_ms;
   unsigned long compute_time_ms;
   unsigned long deadline_jiff;
   int task_state;
};
```

MP2 Scheduling Logic

- What happens when userapp sends YIELD?
  (What does it actually mean when sending YIELD?)
  - Find the calling task
  - Change the state of the calling task to SLEEPING
  - Calculate the time when next period begins
  - Set the timer
    - What should happen if current deadline has passed, but no other tasks are preempting the currently running task?
  - Wake up dispatching thread
  - Put the calling task to sleep (in linux scheduler)

MP2 Scheduling Logic

- What happens when a task is expired?
  - Change the task to READY
  - Wake up the dispatching thread

MP2 Scheduling Logic

- What should dispatching thread do? Dispatching thread handles our main scheduling logic.

    - Trigger context switch
    - As soon as the context switch wakes up, find the READY task with highest priority
    - Preempt the currently running task
    - Set the state of new running task to RUNNING

MP2 Scheduling Logic

- We are using a kernel thread to handle our main scheduling logic
- You will need to explicitly put the kernel thread to sleep when you’re done with your work
- You also need to explicitly check for signals
  - Check if should stop working
  - kthread_should_stop()

MP2 Scheduler API

- schedule() - trigger the kernel scheduler
- wake_up_process (struct task_struct *)
- sched_setscheduler(): set scheduling parameters
  - FIFO for real time scheduling, NORMAL for regular processes, etc.
- set_current_state()
- set_task_state()

MP2 Scheduler API Example

- To sleep and trigger a context switch
```
set_current_state(TASK_INTERRUPTIBLE);
schedule();
```
- To wake up a process
```
struct task_struct * sleeping_task;
……
wake_up_process(sleeping_task);
```

MP2 Final Notes

- Develop things incrementally, follow the mp2 description
- Test things one at a time
  - Try to test one feature after you are done with it
  - Use git commits to organize your developments. When things go wildly wrong, you can rollback to where it once worked.
- Use fixed point arithmetic. Don't use double or float
- Use global variables for persistent state
- Remember to cleanup everything
- If you get permission denied during login, you might have produced too many kernel logs. Post privately on piazza and I will help you (when I see it...)
- If your kernel freezes you might be asking too much from kmalloc (some other things could also happen)