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

Based on previous presentations by Jack Chen and Prof. Adam Bates

Purpose of MP2

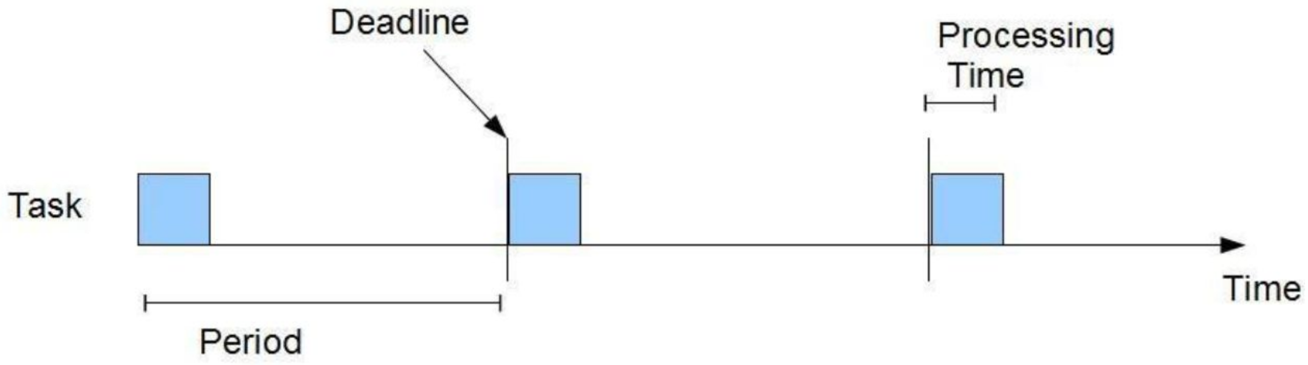
- Understand real time scheduling concepts
- Design a real time schedule module in the Linux kernel
- Learn how to use the kernel scheduling API, timer, procfs
- Test your scheduler by implementing a user level application

Introduction

- Real-time systems have requirements in terms of response time and predictability
    - Airbag in a car
    - Video surveillance systems
    - Audio production
- 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 has
  - *P,* Period,
  - *D,* Deadline, and
  - *C,* Processing Time (Runtime)



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 = higher priority
  - Run highest priority task
  - Preemptive

MP2 Overview

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

- RMS interface (via `procfs`)
  - 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:

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

    - Always assumes that

$$C_i < P_i$$

    - *Ci* is the runtime of task *i*
    - *Pi* is the period to deadline of task *i*

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

```
int main(int argc, char **argv) {
  REGISTER(pid, period, processing_time); // via /proc/mp2/status
  list = READ_STATUS(); // via /proc/mp2/status
  if (pid not in list) return 1;

  t0 = clock_gettime(); // to know when the first job wakes up
  YIELD(pid); // via /proc/mp2/status

  while (exists job) {
    wakeup_time = clock_gettime() - t0;
    do_job();
    process_time = clock_gettime() - wakeup_time;
    printf("wakeup: %d, process: %d\n", wakeup_time, process_time);
    YIELD(pid); // via /proc/mp2/status
  }

  DEREGISTER(pid); // via /proc/mp2/status
  return 0;
}
```

MP2 User Process Behavior

```
int main(int argc, char **argv) {
  REGISTER(pid, period, processing_time); // via /proc/mp2/status
  list = READ_STATUS(); // via /proc/mp2/status
  if (pid not in list) return 1;

  t0 = clock_gettime(); // to know when the first job wakes up
  YIELD(pid); // via /proc/mp2/status

  while (exists job) {
    wakeup_time = clock_gettime() - t0;
    do_job();
    process_time = clock_gettime() - wakeup_time;
    printf("wakeup: %d, process: %d\n", wakeup_time, process_time);
    YIELD(pid); // via /proc/mp2/status
  }

  DEREGISTER(pid); // via /proc/mp2/status
  return 0;
}
```

MP2 User Process Behavior

```
int main(int argc, char **argv) {
  REGISTER(pid, period, processing_time); // via /proc/mp2/status
  list = READ_STATUS(); // via /proc/mp2/status
  if (pid not in list) return 1;

  t0 = clock_gettime(); // to know when the first job wakes up
  YIELD(pid); // via /proc/mp2/status

  while (exists job) {
    wakeup_time = clock_gettime() - t0;
    do_job();
    process_time = clock_gettime() - wakeup_time;
    printf("wakeup: %d, process: %d\n", wakeup_time, process_time);
    YIELD(pid); // via /proc/mp2/status
  }

  DEREGISTER(pid); // via /proc/mp2/status
  return 0;
}
```

MP2 User Process Behavior

```
int main(int argc, char **argv) {
  REGISTER(pid, period, processing_time); // via /proc/mp2/status
  list = READ_STATUS(); // via /proc/mp2/status
  if (pid not in list) return 1;

  t0 = clock_gettime(); // to know when the first job wakes up
  YIELD(pid); // via /proc/mp2/status

  while (exists job) {
    wakeup_time = clock_gettime() - t0;
    do_job();
    process_time = clock_gettime() - wakeup_time;
    printf("wakeup: %d, process: %d\n", wakeup_time, process_time);
    YIELD(pid); // via /proc/mp2/status
  }

  DEREGISTER(pid); // via /proc/mp2/status
  return 0;
}
```

MP2 User Process Behavior

```
int main(int argc, char **argv) {
  REGISTER(pid, period, processing_time); // via /proc/mp2/status
  list = READ_STATUS(); // via /proc/mp2/status
  if (pid not in list) return 1;

  t0 = clock_gettime(); // to know when the first job wakes up
  YIELD(pid); // via /proc/mp2/status

  while (exists job) {
    wakeup_time = clock_gettime() - t0;
    do_job();
    process_time = clock_gettime() - wakeup_time;
    printf("wakeup: %d, process: %d\n", wakeup_time, process_time);
    YIELD(pid); // via /proc/mp2/status
  }

  DEREGISTER(pid); // via /proc/mp2/status
  return 0;
}
```

MP2 User Process Behavior

```
int main(int argc, char **argv) {
  REGISTER(pid, period, processing_time); // via /proc/mp2/status
  list = READ_STATUS(); // via /proc/mp2/status
  if (pid not in list) return 1;

  t0 = clock_gettime(); // to know when the first job wakes up
  YIELD(pid); // via /proc/mp2/status

  while (exists job) {
    wakeup_time = clock_gettime() - t0;
    do_job();
    process_time = clock_gettime() - wakeup_time;
    printf("wakeup: %d, process: %d\n", wakeup_time, process_time);
    YIELD(pid); // via /proc/mp2/status
  }

  DEREGISTER(pid); // via /proc/mp2/status
  return 0;
}
```

MP2 User Process Behavior

```
int main(int argc, char **argv) {
  REGISTER(pid, period, processing_time); // via /proc/mp2/status
  list = READ_STATUS(); // via /proc/mp2/status
  if (pid not in list) return 1;

  t0 = clock_gettime(); // to know when the first job wakes up
  YIELD(pid); // via /proc/mp2/status

  while (exists job) {
    wakeup_time = clock_gettime() - t0;
    do_job();
    process_time = clock_gettime() - wakeup_time;
    printf("wakeup: %d, process: %d\n", wakeup_time, process_time);
    YIELD(pid); // via /proc/mp2/status
  }

  DEREGISTER(pid); // via /proc/mp2/status
  return 0;
}
```

MP2 Process State

- A process in MP2 can be in one of three states
  1. READY: a new job is ready to be scheduled
  2. RUNNING: a job is currently running and using the CPU
  3. 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 Extended PCB

```c
struct mp2_task_struct {
  struct task_struct *linux_task;
  struct timer_list wakeup_timer;
  struct list_head list;
  pid_t pid;
  unsigned long period_ms;
  unsigned long runtime_ms;
  unsigned long deadline_jiff;
  enum task_state state;
};
```

MP2 Scheduling Logic

- What happens when userapp sends YIELD?
  - Find the calling task
  - Change the state of the calling task to SLEEPING
  - Calculate the time when next period begins
  - Set the timer
  - Wake up dispatching thread
  - Put the calling task to sleep (in Linux scheduler)

MP2 Scheduling Logic

- What happens when a wakeup timer expires?
    - 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
  - When dispatching thread wakes up, find highest priority READY task
  - Preempt the currently running task
  - Set the state of new 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 Campuswire and we will help you (when we see it...)
- If your kernel freezes you might be asking too much from kmalloc (some other things could also happen)