CS423 MP2 Walkthrough

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

Overview

- You will develop a **Rate Monotonic Scheduler** for Linux using Linux Kernel Modules
- You will implement bound-based **Admission control** for the Rate Monotonic Scheduler
- You will learn the basic kernel API of the **Linux CPU Scheduler**
- You will use the **slab allocator api** to improve performance of object memory allocation in the kernel
- You will **implement a simple application** to test your Real-Time Scheduler

'schedule()' is the scheduler function. This is GOOD CODE! There probably won't be any reason to change this, as it should work well.

Linux 0.01 Comment

“And you have to realize that there are not very many things that have aged as well as the scheduler. Which is just another proof that scheduling is easy.”

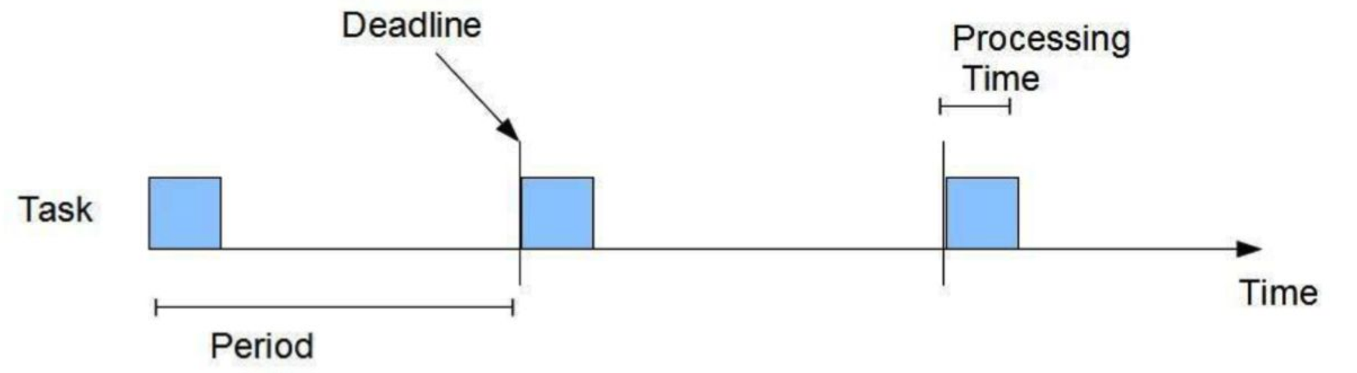
Linus Torvalds, 2001

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 = higher priority
  - Run highest priority task
  - Preemptive
- If B has shorter period than A, then B could preempt A when A is running.

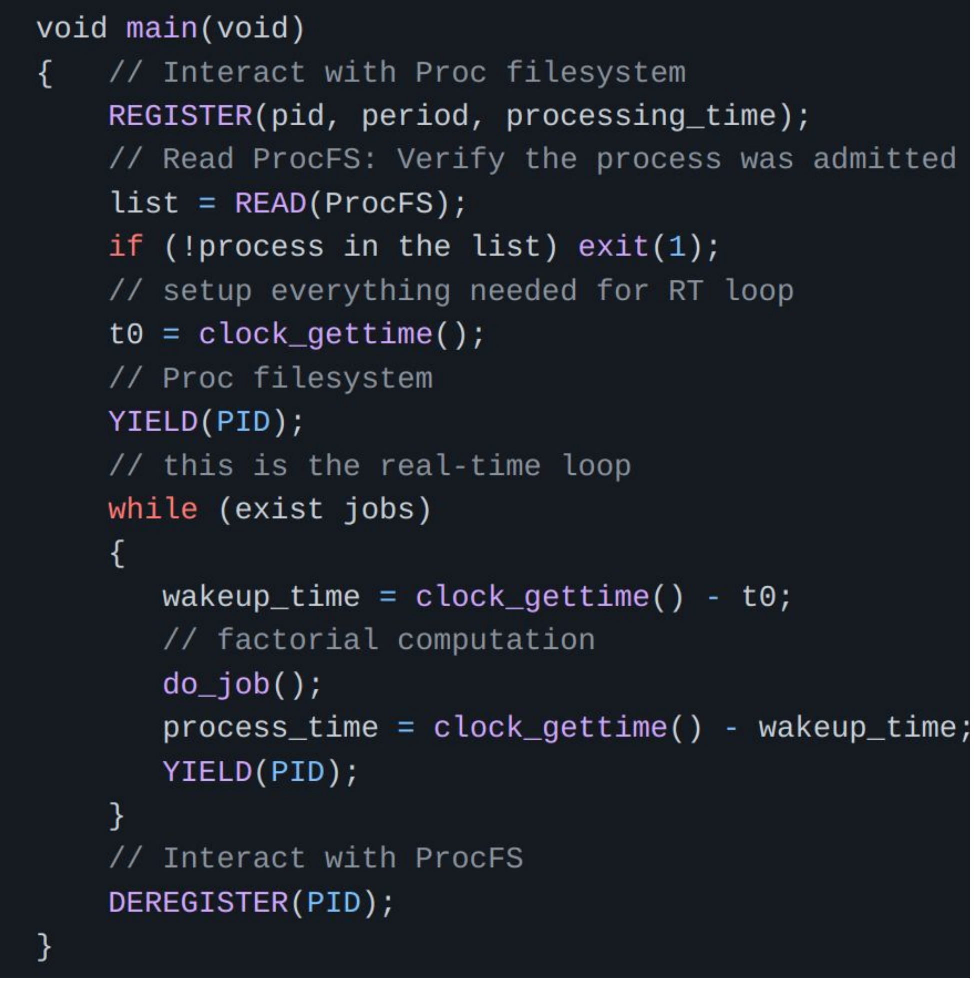
Periodic Tasks Model

- Liu and Layland [1973] model: each task has
    - P, Period,
    - D, Deadline, and
    - C, Processing Time (Runtime)

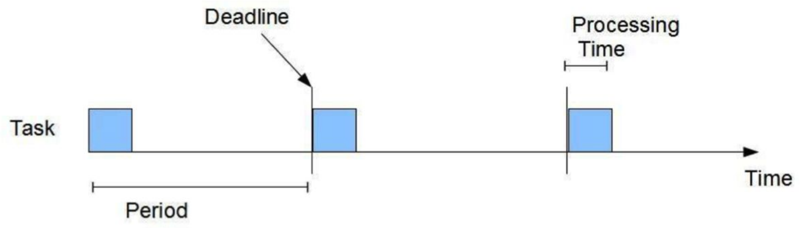


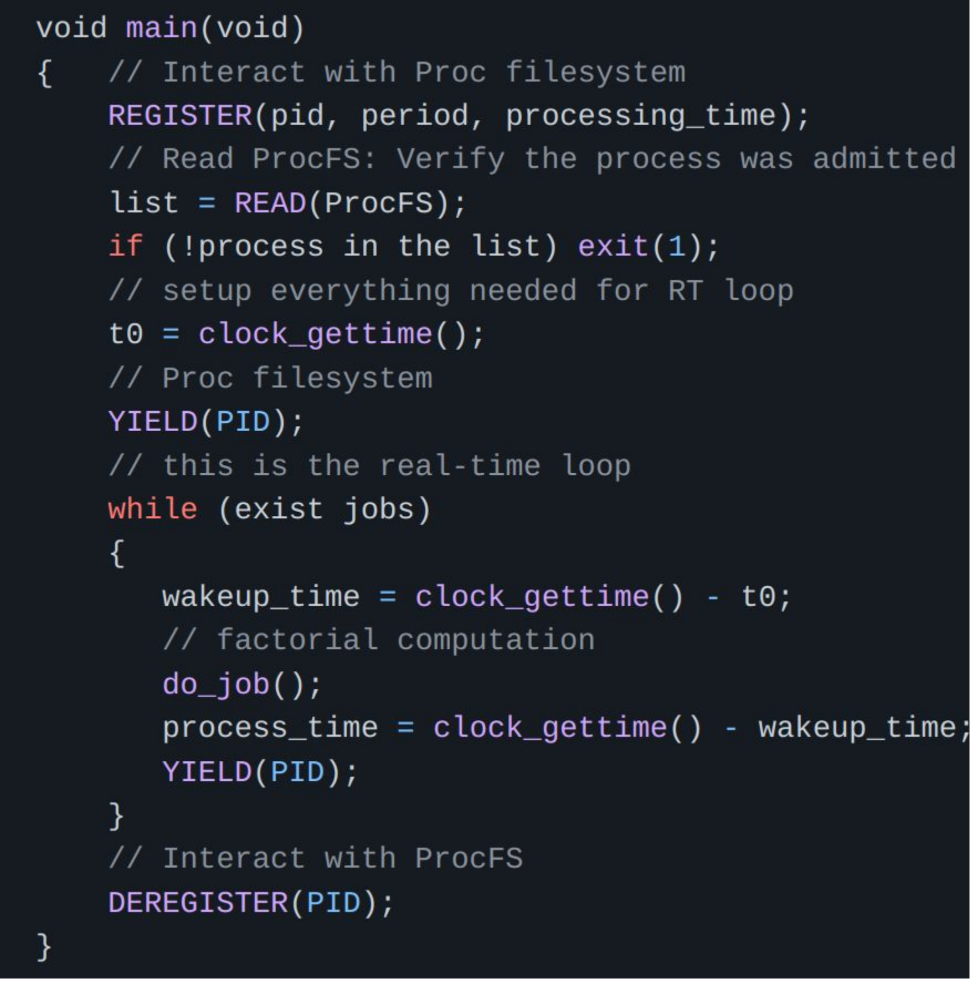
MP2 Overview

- You will implement RMS with an admission control policy as a kernel module (like MP1)
- 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
- Let’s look at the user program with Periodic Tasks Model

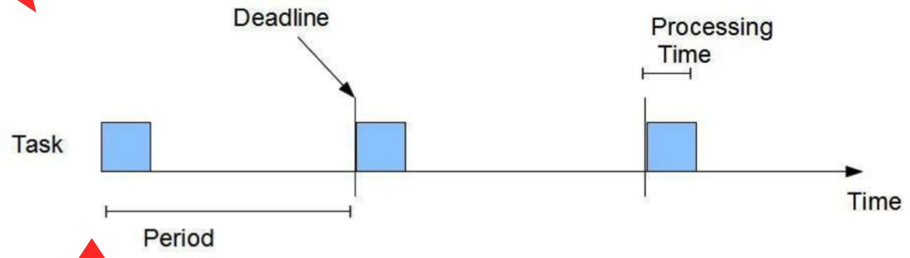


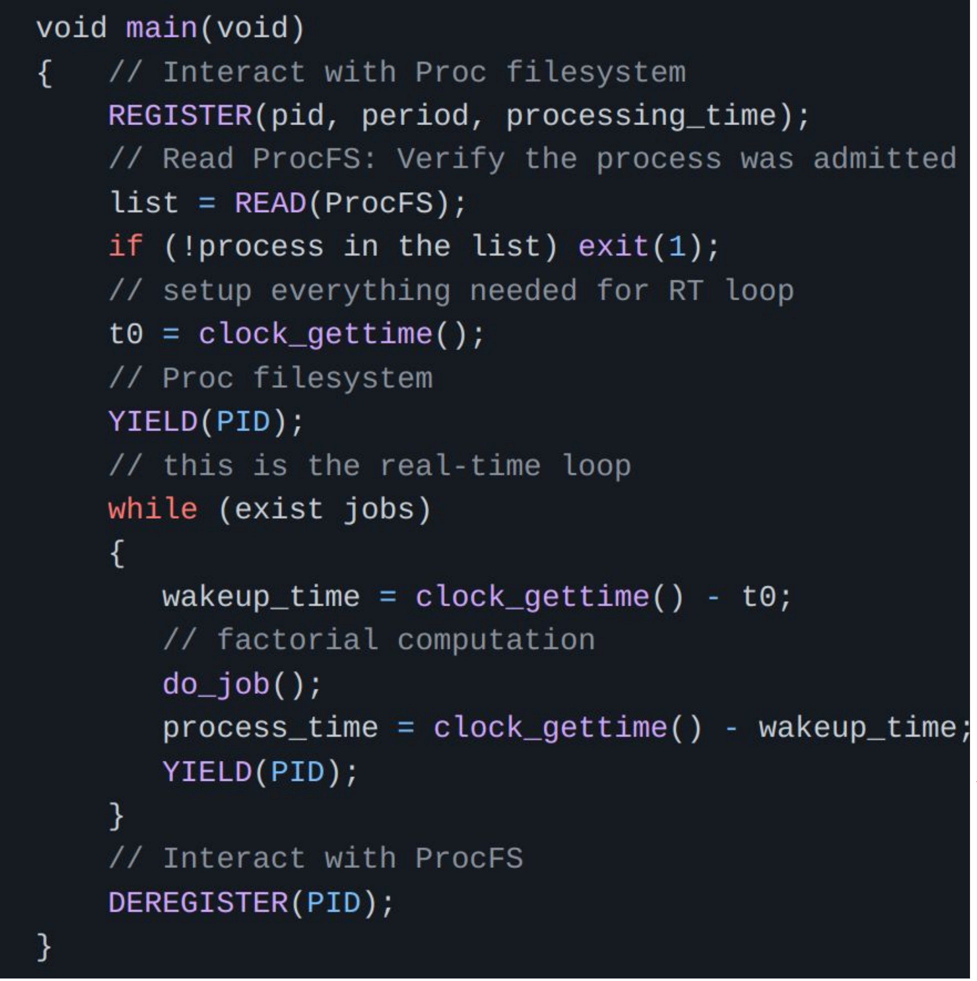
```
void main(void)
{   // Interact with Proc filesystem
    REGISTER(pid, period, processing_time);
    // Read ProcFS: Verify the process was admitted
    list = READ(ProcFS);
    if (!process in the list) exit(1);
    // setup everything needed for RT loop
    t0 = clock_gettime();
    // Proc filesystem
    YIELD(PID);
    // this is the real-time loop
    while (exist jobs)
    {
       wakeup_time = clock_gettime() - t0;
       // factorial computation
       do_job();
       process_time = clock_gettime() - wakeup_time;
       YIELD(PID);
    }
    // Interact with ProcFS
    DEREGISTER(PID);
}
```



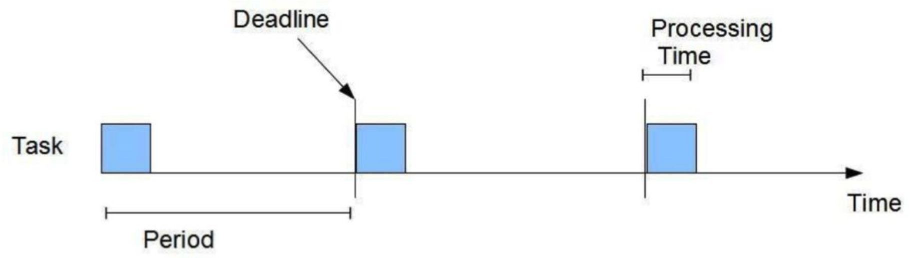


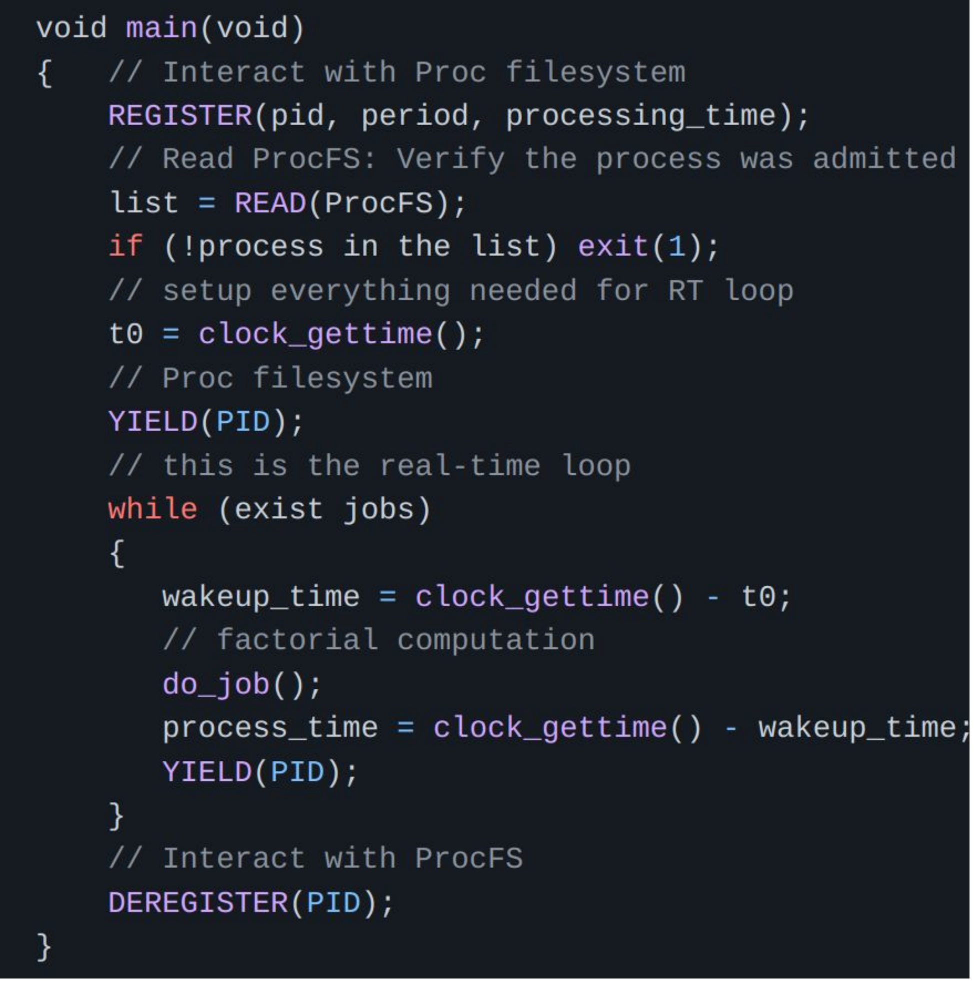
```
void main(void)
{   // Interact with Proc filesystem
    REGISTER(pid, period, processing_time);
    // Read ProcFS: Verify the process was admitted
    list = READ(ProcFS);
    if (!process in the list) exit(1);
    // setup everything needed for RT loop
    t0 = clock_gettime();
    // Proc filesystem
    YIELD(PID);
    // this is the real-time loop
    while (exist jobs)
    {
       wakeup_time = clock_gettime() - t0;
       // factorial computation
       do_job();
       process_time = clock_gettime() - wakeup_time;
       YIELD(PID);
    }
    // Interact with ProcFS
    DEREGISTER(PID);
}
```



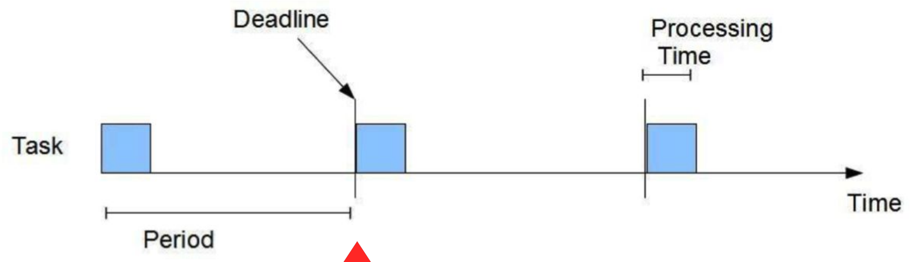


```
void main(void)
{   // Interact with Proc filesystem
    REGISTER(pid, period, processing_time);
    // Read ProcFS: Verify the process was admitted
    list = READ(ProcFS);
    if (!process in the list) exit(1);
    // setup everything needed for RT loop
    t0 = clock_gettime();
    // Proc filesystem
    YIELD(PID);
    // this is the real-time loop
    while (exist jobs)
    {
       wakeup_time = clock_gettime() - t0;
       // factorial computation
       do_job();
       process_time = clock_gettime() - wakeup_time;
       YIELD(PID);
    }
    // Interact with ProcFS
    DEREGISTER(PID);
}
```





```
void main(void)
{   // Interact with Proc filesystem
    REGISTER(pid, period, processing_time);
    // Read ProcFS: Verify the process was admitted
    list = READ(ProcFS);
    if (!process in the list) exit(1);
    // setup everything needed for RT loop
    t0 = clock_gettime();
    // Proc filesystem
    YIELD(PID);
    // this is the real-time loop
    while (exist jobs)
    {
       wakeup_time = clock_gettime() - t0;
       // factorial computation
       do_job();
       process_time = clock_gettime() - wakeup_time;
       YIELD(PID);
    }
    // Interact with ProcFS
    DEREGISTER(PID);
}
```



Timer trigger

do_job()

1. Estimate compute time
    a. Start with a **small factorial** computation.
    b. Measure the **time** it takes to compute.
    c. If the measured time is less than the desired process time, **increase the number** of factorial computations and go back to step b.

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: **Processing Time less than Period**

$$C_i < P_i$$

  - 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.

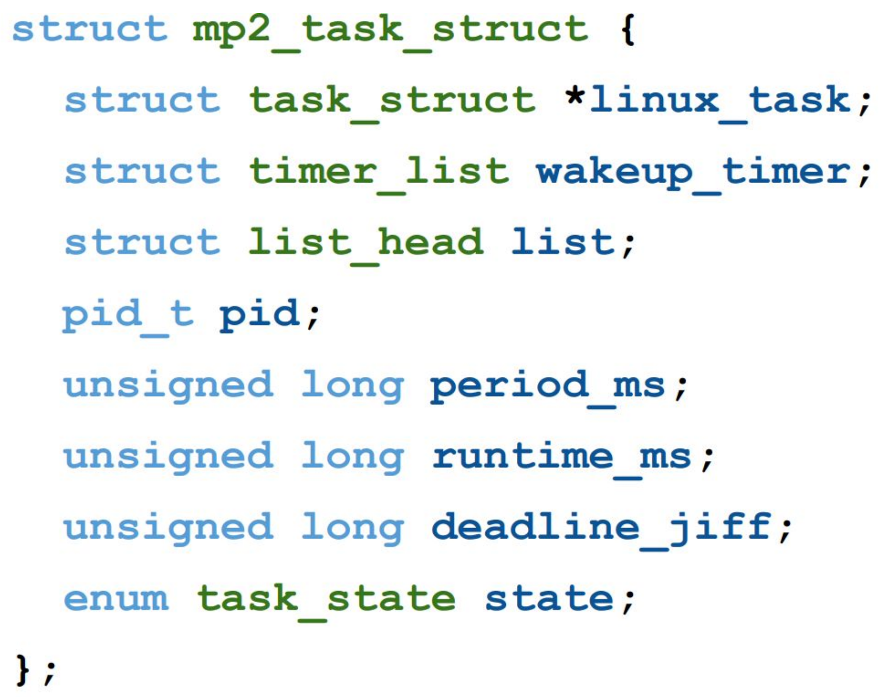
Instead use Fixed-Point arithmetic.

MP2 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.

Extended PCB



```
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;
};
```

Not limited to this style

What happens when userapp sends YIELD?

- Find the **calling task**
  - *Iterate the list, like MP1*
- Change the state of the **calling task to SLEEPING**
- **Calculate the time** when next period begins
- Set the **timer**
  - *Like MP1*
- Wake up **dispatching thread**
  - *wake_up_process()*
- Put the calling task **to sleep** (in Linux scheduler)
  - *set_current_state(); schedule();*

MP2 Scheduling Logic

- What happens when a wakeup timer expires?
  - Change the task to **READY**
    - Check the macro *from_timer()* in linux/timer.h
  - **Wake up** the dispatching thread
    - *wake_up_process()*

What should dispatching thread do?

- When dispatching thread wakes up, **find highest priority** READY task
  - Yes, MP1 list again!
- **Preempt** the currently running task
  - Code block in README 6a
- Set the state of new task to **RUNNING**
  - Also, code block in README 6a
- Put dispatching thread to **sleep**
  - *set_current_state(); schedule();*

dispatching thread is a kernel thread

- You will need to explicitly put the kernel thread to sleep when you're done with your work
    - *set_current_state();schedule()*
- You also need to explicitly check for signals
    - Check if should stop working (signals)
        - *kthread_should_stop()*

Slab Allocator API

- You will use the **slab allocator api** to improve performance of object memory allocation in the kernel
  - Check “linux/slab.h”
  - Something like **kmem_cache_*()**

Some tips

1. Develop things incrementally, follow the mp2 description
   a. Try to test one feature after you are done with it
2. Use git commits to organize your developments. When things go wrong, you can rollback to where it once worked.
3. Use fixed point arithmetic. Don’t use double or float
4. Ask on Piazza and come to Office Hour
5. Start early!