MP2: Rate-Monotonic CPU Scheduling

Peizhe Liu

CS 423: Operating System Design

Fall 2025

Important Dates

- **MP2 is released today.**
- **The due date is 10/28.**
- **MP2 will likely cost you several days (probably more than MP1).**
- **NO LATE SUBMISSION. We found that GitHub allows it, but we won’t.**

Goals

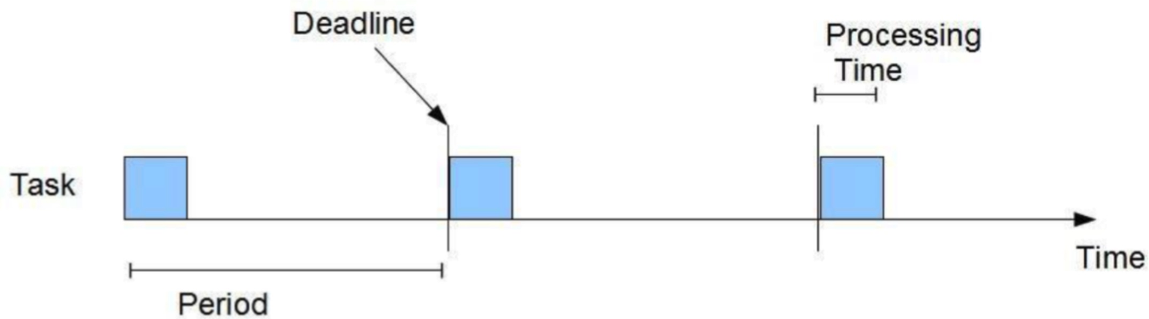
- Learn the basics of real-time CPU scheduling.
- Develop a Rate-Monotonic Scheduler for Linux.
- Utilize more kernel APIs, like scheduler, slab allocator, kthread.
- Test your scheduler with a user app.

What you need

- Your MP0 environment.
- VSCode+clangd setup (strongly recommended).
- Instructions on the course website.
- Accept your assignment on GitHub classroom and start right away.

Periodic Tasks Model

- Real time systems often require response time and predictability.
  - Surveillance camera that captures a frame per 30ms
- Liu and Layland Periodic Task Model to provide that timing guarantee.
- Every tasks carries a processing time and period.
- Every period, the task must run once, and must finish before the next period starts.



Rate-Monotonic Scheduler

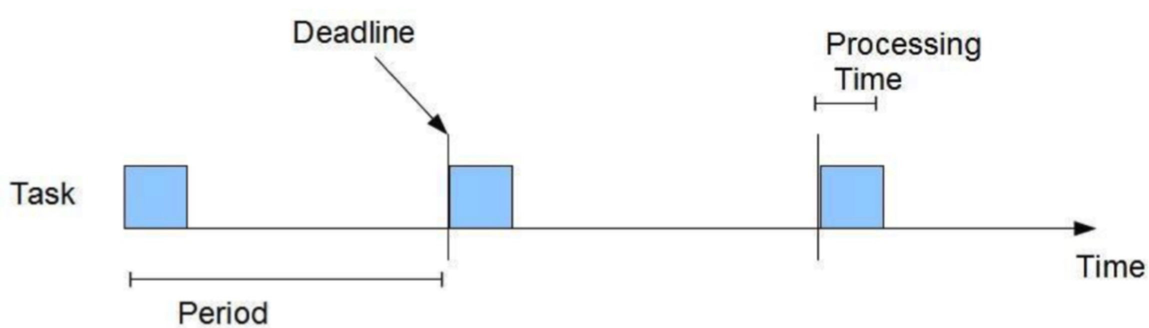
- **An algorithm to implement the Periodic Tasks Model (methods VS rules).**
- **Static priority: the shorter the period, the higher the priority.**
- **Preemptive: higher priority task will preempt lower priority task.**
- **Utilization bound:** $$\sum_{i \in T} \frac{C_i}{P_i} \leq 0.693$$
  - **T: all tasks in the system, including the incoming one**
  - **C: processing time.**
  - **P: period.**

MP2 Overview

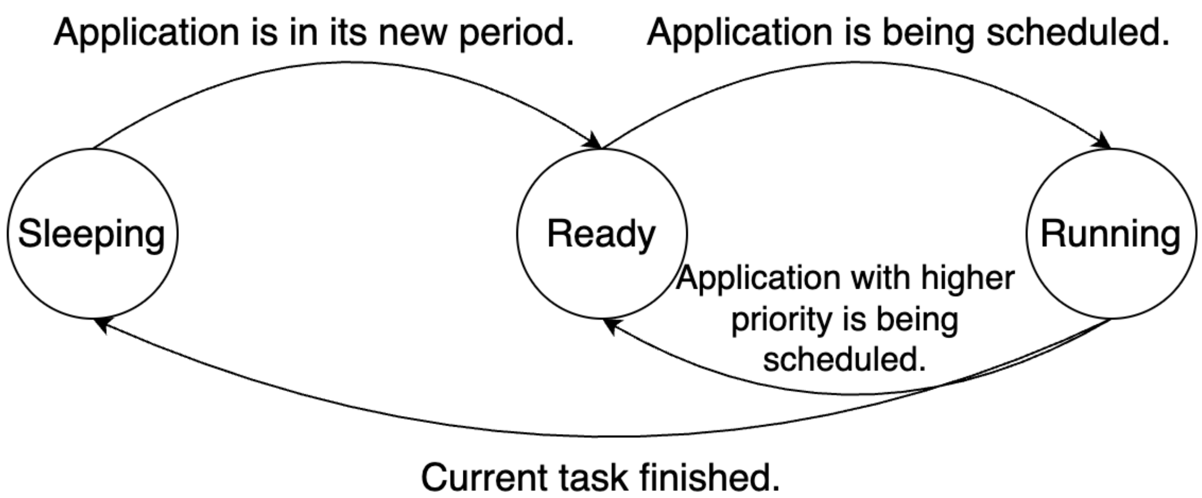
- Kernel module: implements RMS.
  - Communicates with your user application via proc file write.
  - Prints list of applications via proc file read.
  - Handles the real scheduling via Linux scheduler APIs.
- User application: simulates a RT application for your RMS.
  - Registers itself to your kernel module.
  - May run multiple instances.

RMS States

- **Ready: application is currently in its new period. Ready to be scheduled.**
- **Running: application is currently running.**
- **Sleeping: application has finished running in its current period.**

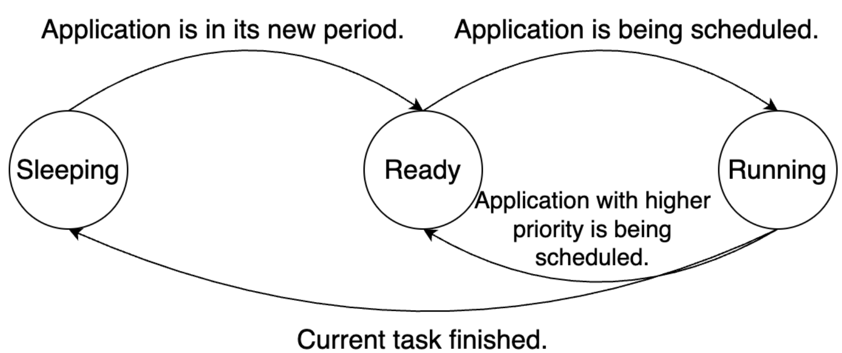


State Transitions

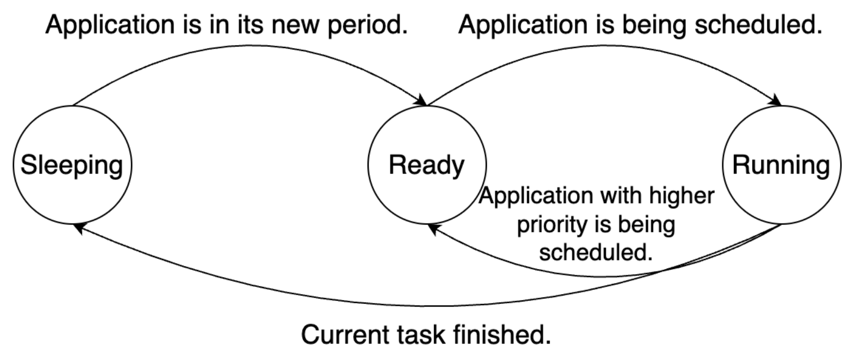


Implement RMS

- **New period: kernel timer**
- **Being scheduled/preempt: dispatcher kthread**
- **Task finished: userapp yield via proc file write**

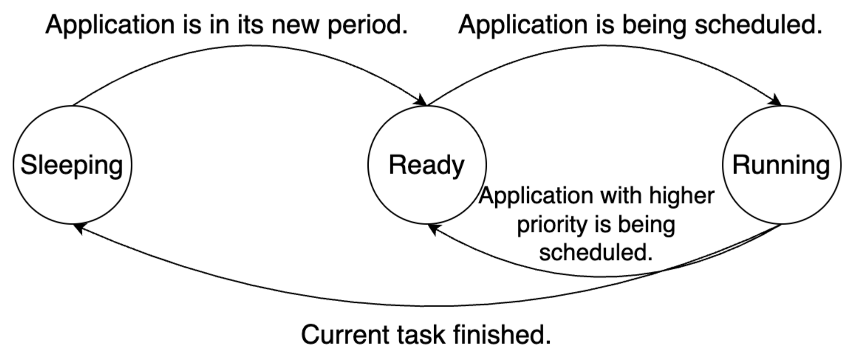


Kernel Timer



- **Every application got its own timer.**
  - **Set to its period time.**
- **When timer fired:**
  - **State transits from SLEEPING to READY.**
  - **Wake up the dispatcher kthread.**
  - *Technically, it cannot fire at READY or RUNNING state…*

Dispatcher KThread



- **Wake up by timer or yield.**
  - **One app got its new period, or app finished running.**
  - **i.e.: Ready for schedule new application.**
- **From applications in Ready state, pick the lowest period one and run it.**
  - **Implements priority property of RMS.**
  - **State transits from READY to RUNNING.**
- **Preempt the running one if necessary.**
  - **Implements preemption property of RMS.**
  - **State transites from RUNNING to READY.**

Work with Linux Scheduler

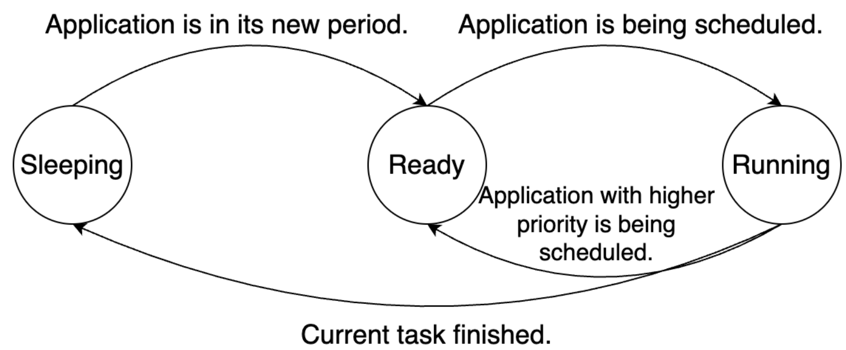
- "Bypass" scheduler, to put an application in sleep or wake it up immediately.
- Utilize kernel scheduling policy…

```c
struct sched_attr attr;
attr.sched_policy = SCHED_NORMAL;
attr.sched_priority = 0;
sched_setattr_nocheck(task, &attr);
```

```c
#include <uapi/linux/sched/types.h>

struct sched_attr attr;
wake_up_process(task);
attr.sched_policy = SCHED_FIFO;
attr.sched_priority = 99;
sched_setattr_nocheck(task, &attr);
```

UserApp Yield



- **UserApp will yield when its task was done.**
  - **This is done via proc file write.**
- **When UserApp yields:**
  - **State transits from RUNNING to SLEEPING.**
  - **Wake up the dispatcher kthread.**
  - *Also technically, it cannot yield at SLEEPING or READY state…*

That's MP2 RMS!

- Old APIs: timer, proc file write.
- New APIs: kthread, scheduler.
- We directly give you the code how to work with scheduler.

Proc File Write

- **Register: RT application registers itself with its PID, Period, Processing Time.**

  `R,PID,PERIOD,COMPUTATION`

- **Yield: RT application has finished its period, yield the CPU.**

  `Y,PID`

- **De-register: RT application has finished and de-registers itself.**

  `D,PID`

Proc File Read

- **Print all registered applications with its PID, period, and process time.**
- **Reuse your MP1 code, certainly.**

```
<pid 1>: <period 1>, <processing time 1>
<pid 2>: <period 2>, <processing time 2>
...
<pid n>: <period n>, <processing time n>
```

Register and De-Register

- **Data structure: custom PCB and linked list.**
  - **Locking can be pretty challenging. Let’s see…**
  - **Remember, clean up after you use.**
- **Utilization bound-based admission control:** $\sum_{i \in T} \frac{C_i}{P_i} \leq 0.693$
  - **Caution! FP arithmetic is very expensive in kernel!**
  - **Kernel does not save FP registers during context switch.**

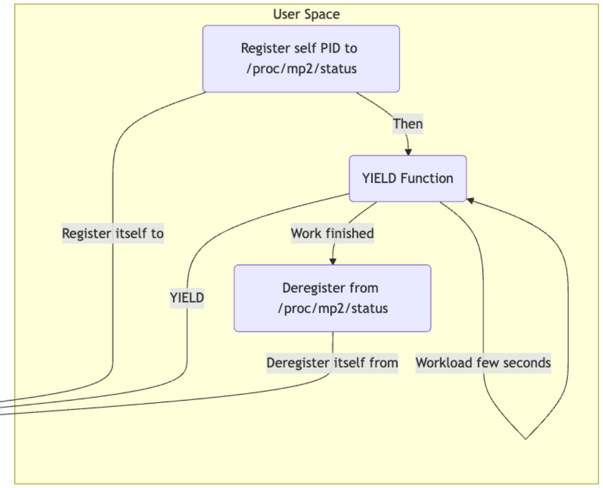
User Application

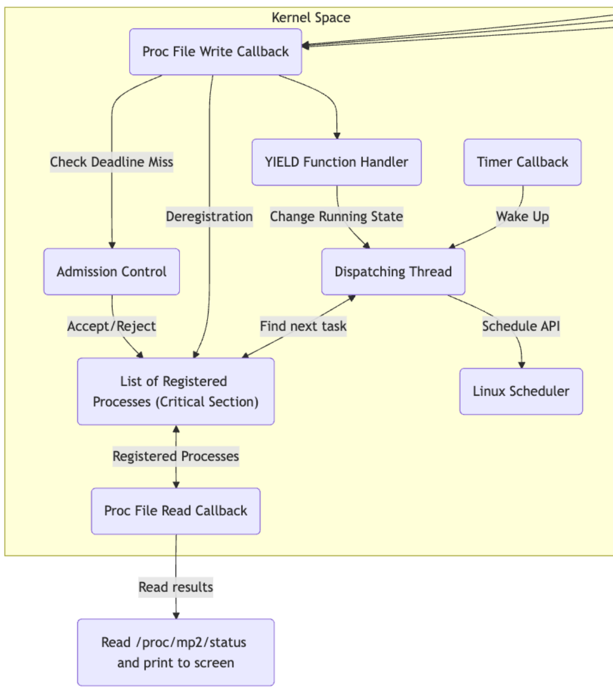
- **Register itself to the scheduler (double check to ensure it was registered).**
- **Yield immediately (signal the scheduler it is ready).**
- **Work loop:**
  - **Do some “real time work”**
  - **Yield**
- **Processing time should match with actual time.**

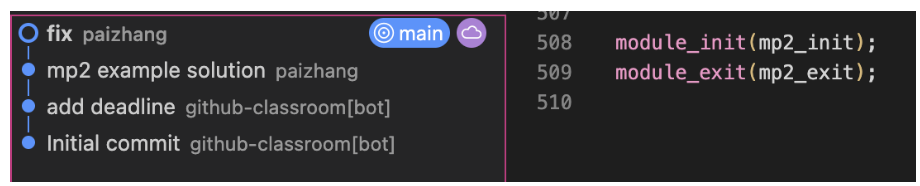
```
void main(void)
{   // Interact with Proc filesystem
    REGISTER(pid, period, processing_time);
    // Read ProcFS: Verify the process was admitted
    list = READ(ProcFS);
    if (!process in the list) exit(1);
    // setup everything needed for RT loop
    t0 = clock_gettime();
    // Proc filesystem
    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);
}
```

Pro tips:

1. Good tools can be helpful.
2. Develop and test incrementally.
3. Try conventional commits.







Total: ~500 lines

My best advise: Start right away!