CS423 Spring 2026 MP2: Rate-Monotonic CPU Scheduling

Assignment Due: 3/23 11:59PM CT.

This document will guide you through your MP2 for CS 423 Operating System Design. In this MP, you will learn how to create a simple Linux kernel scheduler from scratch.

This MP will require you to read, write and debug C-based kernel code in depth. It may take you **several days** (probably more than MP1) to complete.

Overview

- In this MP you will learn the basics of Real-Time CPU Scheduling
- 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 to improve performance of object memory allocation in the kernel
- You will implement a simple application to test your Real-Time Scheduler

Before you start

What you need

- You should have successfully completed MP1. MP2 will use Proc Filesystem, Timers and List which were introduced in MP1.
- You should be able to read, write and debug program codes written in C language.
- (Recommended) You may have a code editor that supports Linux kernel module development - for example, VSCode, Neovim, or GNU Emacs (or Nano).
- (Recommended) You may use the Linux Kernel Documentation to search the Linux kernel documents for concepts and high-level descriptions.
- (Recommended) You may use the Elixir Cross Referencer to search the Linux kernel codebase for function definitions and use cases.
- (Recommended) You may use clangd to help you navigate through your code. We have introduced clangd during MP0, and probably you have already tried it in MP1.

Compile and Test Your Code

Before you start, you should modify your Makefile. Our provided Makefile does not work in your development VM. Specifically, you should change the path to your downloaded kernel source.

```
KERNEL_SRC:= <PATH_TO_YOUR_KERNEL_SOURCE>
```

For example:

```
KERNEL_SRC:= /home/paizhang/cs423/linux-5.15.165
```

Do not modify anything else other than the kernel source path. Our autograder will replace your Makefile, so if you did something fancy like adding additional source files, your MP won't compile and you will receive a zero.

Then, you can compile your module in both your development VM and MP **QEMU** VM.

```
# compile your kernel module
make

# clean your kernel module
make clean
```

Compiling in the development VM offers several advantages, including faster speed and enabling you to configure clangd by doing a compilation with bear. Please refer back to the MP0 doc.

To test your kernel module, you can try loading, unloading, and running it in the MP0 **QEMU** VM. The following commands may be helpful:

```
# inserting kernel module
insmod mp2.ko

# removing kernel module
rmmod mp2.ko

# print the kernel debug/printed messages
dmesg
```

Introduction

Several systems that we use every day have strict requirements in terms of **response time** and **predictability** for the safety or enjoyment of their users. For example, a surveillance system needs to record video of a restricted area, the video camera must capture a video frame every 30 milliseconds. If the capture is not properly scheduled, the video quality will be severely degraded.

For this reason, the Real-Time systems area has developed several algorithms and models to provide this precise timing guarantees as close to mathematical certainty as needed. One of the most common models used is the **Periodic Task Model**.

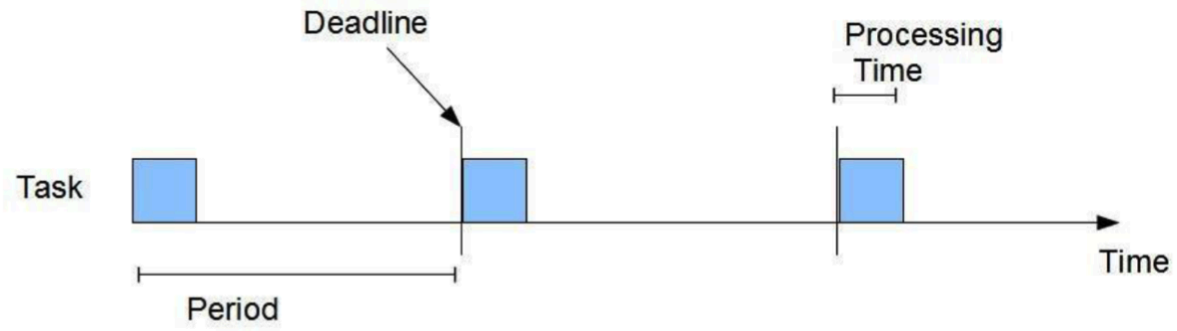

Figure 1: Liu and Layland Periodic Task Model

A Periodic Task, as defined by the Liu and Layland model, is a task in which a job is dispatched by the task after every period P, and must be completed before the beginning of the next period, referred to as deadline D. As part of the model, each job requires a certain processing time C. Figure 1 illustrates this model. In this MP, you will develop a CPU scheduler for the Liu and Layland Periodic Task Model. The scheduler will be based on the **Rate-Monotonic Scheduler (RMS)**. The RMS is a static priority scheduler, in which the priorities are assigned based on the period of the job: the shorter the period, the higher the priority. This scheduler is preemptive, which means that a task with a higher priority will always preempt a task with a lower priority until its processing time has been used.

Problem Description

In this module, we'll implement a Real-Time CPU scheduler based on the **Rate-Monotonic Scheduler (RMS) for a Single-Core Processor**. Our implementation will be in the form of a **Linux Kernel module**. For communication between the scheduler and user-space applications, we'll utilize the Proc filesystem. Specifically, we'll use a **single Proc filesystem entry** for all interactions: `/proc/mp2/status`, which should be readable and writable by any user. Our scheduler will support the following three operations through the Proc filesystem:

- **Registration**: This operation allows the application to inform the Kernel module of its intent to use the RMS scheduler. The application provides its registration parameters to the kernel module (*PID, Period, Processing Time*). (The *Processing Time* is referred to as *time-slice* in our lectures)
- **Yield**: This operation signals the RMS scheduler that the application has completed its period. After yielding, the application will *block until the next period*.
- **De-Registration**: This operation lets the application inform the RMS scheduler that it has *finished* using the RMS scheduler.

Once registered, we refer to this Real-Time user-space application as a task for our Rate-Monotonic Scheduler. There are three RMS states **(READY, SLEEPING, RUNNING)** in our scheduler. The state transition diagram is as follows:

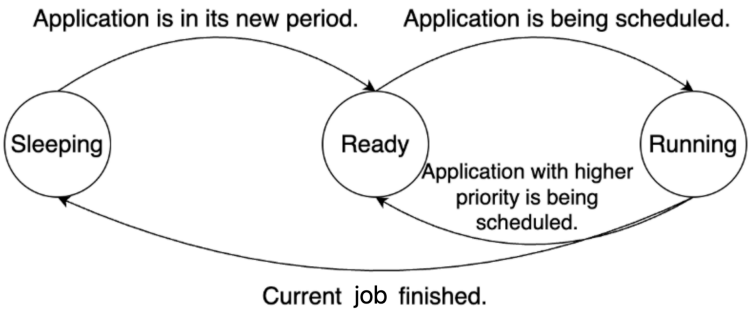

Figure 2: MP2 RMS State Transitions

There are four state transitions in total:

1. **Sleeping to Ready.** This is when the task is in a new period.
2. **Ready to Running.** This is when the task is being scheduled to run.
3. **Running to Ready.** This is when the task is being preempted by another task with higher priority.
4. **Running to Sleeping.** This is when the task has just finished its job in the current period and hence yielding the CPU.

For transition 1, the task knows it is in a new period via a kernel timer. When this wakeup timer expires, the timer interrupt handler should change the RMS state of the task to READY and should wake up the dispatching thread.

For transition 2 and 3, **the kernel dispatching thread (you will implement as part of our RMS scheduler)** is responsible for handling the context switches–scheduling a new task or putting a task to sleep–on behalf of our RMS scheduler based on the priority. The dispatching thread will be woken up as needed. Each run it will make at most one scheduling operation, before going back to sleep.

For transition 4, the task itself sends a YIELD message, and the proc filesystem write callback should put the corresponding task to sleep, change its RMS state to SLEEPING, set up the wakeup timer, and wake up the dispatching thread which does the eventual context switch.

Except the first transition, context switches will happen on all other three transitions. Our scheduler will **depend on the Linux Scheduler** for context switching. There is one key difference between Linux CFS & our RMS. CFS does not actually use a dispatching thread for context switching–the schedule() API is called directly by the running task to find and switch to the next task. RMS will need a dispatching thread only to make scheduling decisions between the registered periodic application tasks, but it still depends on Linux APIs to execute the actual context switches. Hence, the Linux Scheduler APIs will be used for executing the low-level scheduling operations. The APIs you will use for this are `set_current_state`, `schedule`, `sched_setattr_nocheck`, and `wake_up_process`.

To use the correct API to execute a context switch, we first need to understand “context”. **Context means where the current task (not limited to our MP2 task, also includes interrupt, softirq, kernel thread, etc.) originated from.** Let’s analyze these context switches’ current contexts:

- **When a task is being scheduled to run (transition 2).** The task is not running, it is being woken up by the dispatcher thread, so it is not in that task's context. We are trying to wake the task up.
- **When the task is being preempted by another task (transition 3).** The task is being preempted by the dispatcher thread, so it is not in that task's context. We are trying to move the task to the READY state.
- **When the task is yielding the CPU (transition 4).** The task sends a YIELD message to our handler and thus this is in that task's context. We are trying to put the task to sleep.

When it is not in the context (transition 2 and 3), use `sched_setattr_nocheck` to modify the attribute and leverage the Linux scheduler to wake up (with `wake_up_process`) or sleep the task as soon as possible. When we are in the context of the running task (transition 4), we can use `set_current_state` and `schedule` to put the current running task; this is also the case when the dispatching thread is putting itself to sleep after making one scheduling decision.

Our Rate-Monotonic scheduler will register a new periodic application only if the application's scheduling parameters clear the *admission control*. The admission control determines if the new application can be scheduled alongside other already admitted periodic applications without any deadline misses for the registered tasks.

Additionally, an application running in the system should be able to query which applications are registered and also query the scheduling parameters of each registered application. When the entry (`/proc/mp2/status`) is read by an application, the kernel module must return an unordered list comprising the PID, Period, and Processing Time of each registered application in the following **exact format**:

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

For example:

```
401: 100000, 423
400: 200000, 523
404: 150000, 416
```

You will also craft a simple test application for our scheduler. This application will be a single-threaded periodic app that loops over a job dedicated to calculating factorials; in other words, this period app simulates the dispatch of one job in each iteration of its loop. This periodic app should first register with the scheduler (via admission control). It must specify its scheduling parameters during registration: the job Period and Processing Time, both in **milliseconds**.

After attempting registration, the application should read the Proc filesystem entry to confirm that its PID is listed, indicating that it made it through admission control. Following this, it should notify the scheduler of its readiness by dispatching a YIELD message via the Proc filesystem. The app should then initiate the Real-Time Loop and start executing the periodic jobs. One job is analogous to one iteration of the Real-Time Loop. At the loop's conclusion, i.e. after the completion of all its periodic jobs, the app must de-register itself with the scheduler. This is done using the Proc filesystem. More precisely:

1. The periodic application must register itself with the scheduler (to try and make it through admission control). During the registration process, it must specify its scheduling parameters: The Period and Processing Time of each job, both expressed in milliseconds.
2. After the registration, the application must read the `/proc/mp2/status` entry to ensure that its PID is listed. This means the task is accepted; else the task must terminate.
3. After this, the application must signal the scheduler that it is ready to start by sending a YIELD message to `/proc/mp2/status`.
4. Then the application must initiate the Real-Time Loop, and begin the execution of the periodic jobs. One job is equivalent to one iteration of the Real-Time Loop. After each job, the process should send a YIELD message to `/proc/mp2/status`.
5. At the end of the Real-Time Loop, the application must de-register itself after finishing all its periodic jobs via `/proc/mp2/status`.

Here's the **pseudo code** for the Periodic Application:

```
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);
}
```

To determine the processing time of a job, run the application using the Linux scheduler first and measure the average execution time of one iteration of the Real-Time Loop; try and run several iterations to compute this average. Your application can perform a simple computation in do_job(); calculating the factorial of a fixed number is recommended. This MP focuses on Real-Time scheduling, so it works best if do_job() takes a predictable time to execute.

Implementation Challenges

Scheduling typically encounters three pivotal challenges that must synchronize; otherwise, there might be occurrences of zombie processes or processes disobeying the RMS policy:

1. **First Challenge**: It involves waking your application when it's ready to run. Rate Monotonic upholds a strict scheduling policy, not allowing the job of an application to run before its period. Hence, the application *must sleep until the beginning of the period without any busy waiting*, avoiding the waste of valuable resources. Consequently, our applications will exhibit one of 3 RMS states in the kernel:
    - **READY**: The application has reached the beginning of the period, and a new job is ready to be scheduled.
    - **RUNNING**: The application is currently executing a job and using the CPU.
    - **SLEEPING**: The application has completed the job for the current period and is awaiting the next one.
2. **Second Challenge**: It involves preempting a currently running application that has a lower priority as soon as a new application with a higher priority is READY. This necessitates triggering a context switch, achievable through the Linux Scheduler API.
3. **Third Challenge**: It involves preempting an application that has completed its current job. For this, we assume that the application is always well-behaved and notifies the scheduler via the YIELD message upon completing its job for the current period. After receiving a YIELD message via the Proc filesystem, the RMS scheduler must put the application to sleep until its next period. This involves setting up a timer, preempting the current application, and context switching the CPU to the next READY application with the highest priority.

Considering these challenges, it is evident that the Process Control Block of each task needs augmenting, so it can include the application's RMS state **(READY, SLEEPING, RUNNING)**, a wake-up timer, and the task's scheduling parameters, including the application period (denoting priority in RMS). The scheduler should also maintain a list as a run queue[1] of all the registered tasks, enabling the selection of the correct task to schedule during any preemption point. An *additional challenge* is minimizing performance overhead in the CPU scheduler. Avoid Floating Point arithmetic as it's resource-intensive.

[1] Typically, most schedulers use a fancier data structure to implement a runQ (Linux uses a RB-tree). However, for MP2 a linked list will suffice for a runQ because we plan to test MP2 with only a handful of tasks/applications.

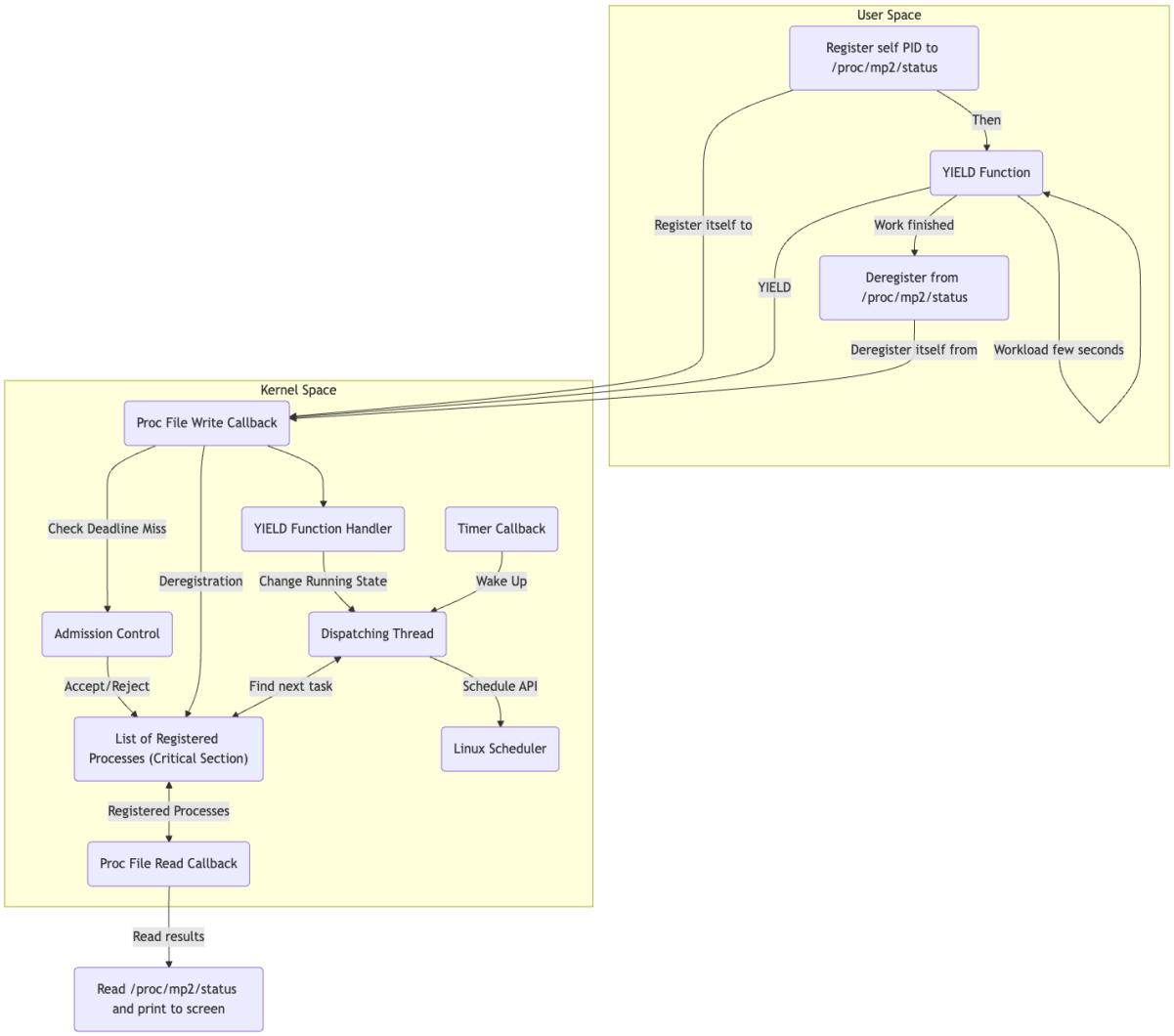

Figure 3: MP2 Architecture Overview

Implementation Overview

In this section, we will guide you through the implementation process. Figure 2 shows the basic architecture of your scheduler.

1. Start

First, implement an empty ('Hello World!') Linux Kernel Module. You may also reuse some of the generic functions you implemented for MP1 as necessary, such as the linked list helper functions.

2. Proc Filesystem

After this, you will implement the Proc Filesystem entry. The write callback function should have a switch to separate each type of message (REGISTRATION, YIELD, DE-REGISTRATION). At

this point, you may leave the functions empty or simply print a message using `printk()`, but you will implement their full functionality during Steps 7 and 8.

Each message type is encoded as a single-operation character, which is used to perform the switch operation. This allows you to receive various types of messages with a single Proc filesystem entry and provide a unified interface. Please use the **exact string format shown below** in each example message, for each of the Proc Filesystem message types:

- For REGISTRATION: `R,PID,PERIOD,COMPUTATION`
  - Example: `R,123,1000,500`, the unit of `period` and `computation` is in `milliseconds`. `PID` is the process ID of the process.
- For YIELD: `Y,PID`
  - Example: `Y,123` where `PID` is the process ID of the process.
- For DE-REGISTRATION: `D,PID`
  - Example: `D,123` where `PID` is the process ID of the process.

You should be able to test your code at this point.

3. Process Control Block

You should augment the Process Control Block (PCB). You will not directly modify the Linux PCB (struct task_struct) but instead declare a separate data structure that also points to the corresponding PCB of each task.

Create a new struct for each registered application, which includes a pointer of type struct task_struct. In Linux, this is the data structure that represents the PCB and it is defined in `linux/sched.h`. Also, we recommend you index your list by PID. To obtain the task_struct associated with a given PID, we have provided you with a helper function in `mp2_given.h`.

Add any other information you need to keep–the current RMS state of the (application) task, including the period, the processing time, a wake-up timer for the task, etc. Your data structure may look something like this:

```
struct mp2_task_struct {
    struct task_struct* linux_task;
    struct timer_list wakeup_timer;
    ...
}
```

4. Registration

Now you can implement *registration*. Do not worry about admission control at this point, we will implement admission control in Step 8. To implement registration use the empty registration function from Step 2.

In this function, you must allocate and initialize a new `mp2_task_struct`; use the slab allocator for this memory allocation. The slab allocator is an allocation caching layer that improves allocation performance and reduces memory fragmentation in scenarios that require intensive

allocation and deallocation of objects of the same size (e.g creation of a new PCB after a `fork()`). As part of your kernel module initialization you must create a new cache of size `sizeof(mp2_task_struct)`. Subsequently, this new cache will be used by the slab allocator to allocate a new instance of `mp2_task_struct`.

The registration function must initialize the task in SLEEPING state. However, we will let the task continue running until the application sends the YIELD message. In other words, we do not enforce any RMS scheduling until the application enters the Real-Time loop. This allows necessary initializations before the application can enter the real-time loop.

In the registration function, you will also need to insert this new structure into the list of tasks. This step is very similar to what you did in MP1.

Also, at this point, you should implement the Read callback (described in the early sections) of the Proc Filesystem entry.

5. De-registration

You should implement *de-registration*. The de-registration requires you to remove the task from the list and free all data structures allocated during registration. Again, this is very similar to what you did in MP1. You should be able to test your code by trying to register and de-register some tasks; the read callback must not list a de-registered application.

6. Implementing the Kernel Dispatching Thread

We recommend you define a global variable with the current task (`struct mp2_task_struct* mp2_current`). This will simplify your implementation. This practice is common and it is even used by the Linux kernel. In general, the current task is also the currently running task, and this global variable can be set to NULL when there is no running task; the setting and clearing of this global variable will not necessarily be instantaneous as the rest of this section shows.

As soon as the dispatching thread wakes up, it should find **a new task in the READY RMS state with the highest priority (i.e., the shortest period)** from the list of registered tasks.

If the current task is in RUNNING state, the dispatching thread must handle the following scenarios:

- If the newly chosen task has a higher priority than the current task, it must preempt the current task and context switch to the chosen task. It must also set their RMS states to READY and RUNNING, respectively.
- If the newly chosen task doesn't have a higher priority than the current task, then it should not preempt the current task and leave the chosen task in the READY RMS state.
- If there are no tasks in the READY RMS state, it should leave the current task running.

If the current task is in SLEEPING RMS state, it means that the yield handler (described in Step 7) has already changed the task's RMS state from RUNNING to SLEEPING, and has woken up the dispatching thread. The dispatching thread must handle the following scenarios:

- If there is a newly chosen task, the dispatching thread preempts the current task, and context switches to the chosen task. It must set the chosen task to RUNNING RMS state, but leave the current task in SLEEPING RMS state.
- If there are no tasks in the READY state, it does nothing.

If the current task is NULL, the dispatching thread must handle the following scenarios:

- If there is a newly chosen task, the dispatching thread context switches to it, sets its RMS state to RUNNING, and sets the current task to it.
- If there are no tasks in the READY state, it does nothing.

Note that the current task can never be in READY RMS state because that implies it is waiting to be scheduled.

As discussed above, the dispatcher thread handles the context switch of the chosen and current running task, to wake up the chosen task or preempt the running task. To wake up the chosen task, the dispatching thread should execute the following code:

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

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

We know that 99 is the highest priority in the Linux scheduler, and any task running on the `SCHED_FIFO` will hold the CPU for as long as it wishes. This will ensure that the scheduler wakes up the task as soon as possible and keeps it running as long as possible.

Similarly, for the old running task (preempted task), the dispatching thread should execute the following code:

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

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

This will clear previous attributes and demote the task, ensuring that the scheduler puts the preempted task to sleep.

Finally, you also need to put the dispatcher thread to sleep after each run. As mentioned before, you are in the context of the dispatcher thread, so you can directly use the functions `set_current_state(TASK_INTERRUPTIBLE)` (Please think: why interruptible in this case?) and `schedule()` to get the dispatching thread to sleep immediately.

After it wakes up, it will make another scheduling decision and go to sleep. It works like a loop, so make sure that you terminate the dispatching thread when your kernel module is unloaded: after it wakes up, it should check if it should terminate itself.

7. Implementing the Wake-up Timer Handler

Now you should implement the wake-up timer handler. The handler should change the state of the corresponding task to READY and wake up the dispatching thread. You should use the two-halves mechanism, where the top half is the wake-up timer handler and the bottom half is the dispatching thread.

Independent of the wake-up timer, based on your implementation needs, consider whether to wake up the dispatching thread during application de-registration.

8. Implementing the YIELD Handler

In this step, you will implement the YIELD handler function from the Proc filesystem callback that we left blank in Step 2. In this function, you should change the RMS state of the calling task to SLEEPING, calculate the next release time (that is the beginning of the next period), set the timer in the PCB, and wake up the dispatching thread.

As discussed above, when yielding, you are already in the context of the running task, so you can use `set_current_state(TASK_UNINTERRUPTIBLE)` (Please think: why uninterruptible in this case?) and `schedule()` to get the current task to sleep immediately.

9. Implementing Admission Control

You should now implement the admission control. This checks if the set of currently registered tasks and the soon-to-be-registered task can all be scheduled without missing any deadlines according to the utilization-bound-based method. If not, admission control must fail the registration of the new task with a meaningful error message.

The utilization bound-based admission method establishes that a task set is schedulable if the following equation is true:

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

Where *T* is the set of all registered tasks in the system including the soon-to-be-registered task, C_i is the Processing Time used per period P_i for the i-th task in the system.

To implement admission control or any time computation, do not use Floating-Point. Floating-Point support is very expensive in the kernel and should be avoided at all costs. Instead, use Fixed-Point arithmetic implemented through integers.

10. Memory De-allocation

You should ensure that all data structures are regularly cleaned up and their corresponding memory is deallocated. This is especially true after the kernel module is removed. You can assume that each application is well-behaved and de-registers at its completion.

11. Testing

Now implement the test application and make sure that your scheduler behaves as expected. It is recommended that you test with multiple instances of the test application and different periods and computation loads. Your test application should print the start time and finish time of every job (iteration of its loop) and run the application with various periods and number of jobs. We also recommend that you design your test application such that the period and number of jobs of the application can be specified as command line arguments (argc/argv); this will be useful for convenient testing of your scheduler.

Note on Code Quality

Please note that the code quality of each MP will also affect your grade. In MP2, code quality accounts for 10% of the total score.

You can read about the Linux Kernel's requirements for code quality here: https://www.kernel.org/doc/html/v5.15/process/4.Coding.html

For MP, we use a relaxed version of the Kernel Code Quality Guideline for grading. For example, we require:

- Your code should not trigger compiler warnings.
- Properly protect all multithreaded resources with locks.
- Abstract the code appropriately and use functions to split the code.
- Use meaningful variable and function names.
- Write comments for non-trivial code segments.

We DO NOT require, but encourage you to:

- Follow the kernel's code styling guideline (https://www.kernel.org/doc/html/v5.15/process/coding-style.html)
- Compile your code with the `-W` / `-Wall` flag.

Here is some advice:

- Your code should include comments where appropriate. It is not a good idea to repeat what the function does using pseudo-code, but instead, provide a high-level overview of the function, including any preconditions and post-conditions of the algorithm. Some functions might need only 1-2 lines of comments, while others might need a paragraph.
- Also, your code must be split into functions, even if these functions contain no parameters. This is a common situation in kernel modules because most of the variables are declared as global, including but not limited to data structures, state variables, locks, timers, and threads.
- An important aspect of kernel code readability is to show whether a function holds the lock for a data structure or not. Different conventions are usually used. A common convention is to start the function with the character _ if the function does not hold the lock of a data structure.
- In kernel coding, performance is a very important issue; usually, the code uses macros and preprocessor commands extensively. Proper use of macros and identifying possible situations where they should be used is important in kernel programming.
- It's also worth noting that Floating Point Units can be sluggish due to excessive context switch stores and might not always be accessible. Hence, relying on Floating Point

arithmetic should be minimized. A majority of generic code can be crafted using Fixed Point arithmetic.
- Finally, in kernel programming, the use of the goto statement is not rare. A good example of this is the implementation of the Linux scheduler function schedule(). In this case, the use of the goto statement improves readability and/or performance. “Spaghetti code” is never a good practice.

Note on SWE Practices

- Use small and frequent commits: Instead of working for three days and committing 500 lines at once, commit every time you finish a small, logical piece of work. If you break something, it is also much easier to roll back. This also helps you to show your engineering effort.
- Use Conventional Commits: e.g., “feat: return registered processes.” https://www.conventionalcommits.org/en/v1.0.0/
- Maintain a clean repo with .gitignore: 1) Exclude system junks like .DS_store and Thumbs.db, 2) no environment files like clang metadata, and 3) skip any program-generated files like your kernel module.

Submit Your Result

Here are the steps to accept and submit your MP.

- Open the GitHub classroom link (posted on Piazza) and login using your GitHub account.
- Find your name in the student list and click it to accept the assignment. Please double-check your name and email address before accepting the assignment (If you choose another student’s name by mistake, please contact TA).
- A repo named `cs423-uiuc-spring26/mp2-<GitHubID>` will be automatically created for you with the starter code in it.
- Your kernel module must be compiled to `mp2.ko`, and your test application must be compiled to `userapp`. Push your code to your repo before the deadline. We will grade your last commit before the deadline.
- Please also edit the `README` file to briefly describe how you implement the functionalities. e.g., how the YIELD function works, how the admission control works, etc. You can also describe any design decisions you made. This will help us understand your code better and give you more points. Please upload the `README` to your GitHub repo.
- Please also read the MP submission guidelines and AI policies posted on Piazza and make sure your submission does not violate them.

Grading Criteria

Criterion	Points

Are read/write ProcFS callbacks correctly implemented? (parse scheduler msg's, print status)*	10
Is your Process Control Block correctly implemented/modified?	5
Is your registration function correctly implemented (Admission Control, using slab allocator)?	15
Is your deregistration function correctly implemented?	5
Is your Wake-Up Timer correctly implemented?	5
Is your YIELD function correctly implemented?	15
Is your Dispatching Thread correctly implemented?	15
Are your Data Structures correctly Initialized/De-Allocated?	5
Does your test application follow the Application Model work?	10
Document Describing the implementation details and design decisions	5
Your code compiles without warnings, runs correctly, and does not use any Floating Point arithmetic.	5
Your code is well commented, readable, and follows software engineering principles.	5
Total	**100**