MP 2: Introduction to Linux Kernel Programming

Due Date: Completed and turned in via git before March 22, 2021 at 11:59pm
Points: MP 2 is worth 100 points

Goals and 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 the performance of object memory allocation in the kernel.
- You will implement a simple application to test your Real-Time Scheduler.

Extra Credit Checkpoints

You can earn extra credit by starting on this MP early:

- EC Checkpoint 1: Complete Steps 1-3 by Monday, Mar. 8 at 11:59pm for +5 points.
- EC Checkpoint 2: Complete Steps 1-9 by Monday, Mar. 15 at 11:59pm for +5 points.

TA Walkthrough

- Slides
- Recording

Introduction

Several systems that we use every day have requirements in terms of response time (e.g. delay and jitter) and *predictability* for the safety or enjoyment of their users. For example, if 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, 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 guarantee as close to mathematical certainty as needed. One of the most common models used is the *Periodic Task Model*.

A periodic task as defined by the Liu and Layland model [10] is a task in which a job is released 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.

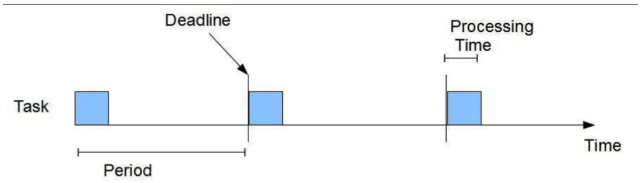

Figure 1: Liu and Layland Periodic Task Model

In this MP, we will develop a CPU scheduler for the Liu and Layland Periodic Task Model. The scheduler will be a *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*[1] *with a higher priority will always preempt a task with a lower priority until its processing time has been used*.

Developmental Setup

In this assignment, you will again work on the provided virtual machine and you will develop kernel modules for the Linux Kernel 4.4.0-*NetId* that you installed on your machine as part of MP0. Again, you will have full access and control of your VM; you will be able to turn it on, and off using the VMWare vSphere Console. Inside your VM you are free to install any editor or software like Gedit, Emacs and Vim.

During the completion of this and other MPs, errors in your kernel modules could potentially lead to "bricking" your VM, rendering it unusable. If this happens, please post to Piazza asking for help and a TA will work with Engr-IT to have your VM restored. However, this will cost you precious hours of development time! It is recommended that you backup your work often by using version control and snapshotting your VM.

Finally, you are encouraged to discuss design ideas and bugs on Campuswire. Campuswire is a great tool for collective learning. *However, please refrain from posting large amounts of code*. Two or three lines of code are fine. High-level pseudo-code is also fine.

Starter Code

We have provided some starter code to get started on this MP. Much like in MP1, you will receive this code in your CS 423 github.

1. Create your CS 423 github by following our "Setting up git" guide here, if you have not already done so. If you did MP1, you don't need to do this again.
2. Run the following commands in your `cs423/netid/` git repo:

```
git fetch release
git merge release/mp2 -m "Merging initial files"
```

Problem Description

In this MP, you will implement a Real-Time CPU scheduler based on the Rate-Monotonic Scheduler (RMS) for a single-core processor. You will implement your scheduler as a Linux Kernel module and use `procfs` to communicate between the scheduler and userspace applications. Use a *single `procfs` entry* named `/proc/mp2/status`, readable and writable by any user, for all communication. Your module must implement three operations available via writes to `/proc/mp2/status`:

- **Registration**: Notify the kernel module that an application will use the RMS. These messages are strings with the following format:

  ```
  R, <pid>, <period>, <processing time>
  ```

  where `<pid>` is the integer PID of the process, `<period>` is the task period, and `<processing time>` is the task processing time. All times should be in milliseconds and encoded as integers. Notice that the "R" in the string is a literal 'R' that denotes this is a registration message.

- **Yield**: Notify the RMS that the application has finished its period. After sending a yield message, the application will block until the next period. Yield messages are strings with the following format:

  ```
  Y, <pid>
  ```

  where `<pid>` is the integer PID of the process.

- **De-registration**: Notify the kernel module that the application has finished using the RMS. This message has the following format:

  ```
  D, <pid>
  ```

  where `<pid>` is the integer PID of the process.

Your RMS will only register a new periodic application if the application's scheduling parameters pass the *admission control*. The admission control must decide if the new application can be scheduled along with other

already admitted periodic application without causing *any* deadlines to be missed for *all* registered tasks in the system. The admission control should be implemented as a function in your kernel module. Your scheduler does *not* need to handle *overrun cases*; i.e., when the application uses more than the reserved processing time.

This scheduler will rely on the Linux Scheduler to perform context switches; therefore, you should use the Linux Scheduler API. You do not need to handle any low-level functions. We will discuss this more in Step 6. of the implementation overview.

Additionally, any application running in the system should be able to query which applications are registered and their scheduling parameters. When `/proc/mp2/status` is read, the kernel module must output a list with the PID, period and processing time of each registered application. This list should have the following format:

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

You will also develop a simple test application for your scheduler. This application will be a *single-threaded* periodic application with individual jobs doing some computation. This application must do the following in order:

1. Register itself via `/proc/mp2/status` with the scheduler and pass admission control.
2. Read from `/proc/mp2/status` to ensure its PID is listed, indicating it was accepted.
3. Signal the RMS it is ready to start by sending a yield message via `/proc/mp2/status`.
4. Initiate a real-time loop and execute periodic jobs to do some computation. Each job is equivalent to one iteration of the real-time loop; after each job, the process should yield via `/proc/mp2/status`.

   Each iteration of the loop (i.e., each job) must also print the following to stdout: how long the job took to wake up after the previous job, and how long the job took to complete. The output line per job should have the following format:

   ```
   wakeup: <wakeup_time>, process: <process_time>
   ```

   Your test application must be able to adjust the number of jobs run and the period via its arguments. The first argument is the job period in milliseconds as a string encoding an integer; the second argument is the number of jobs to run as a string encoding an integer.
5. Once all jobs are done, de-register itself via `/proc/mp2/status`.

Here is some C-like pseudo-code for this test application. Note that this is pseudo-code - for example, `clock_gettime()`'s output is actually given via a pointer parameter, not its return value.

```
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

  //this is the real-time loop
  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;
}
```

Note that we did not specify what the actual computation done by the job must be. We recommend a simple task, e.g., computing the factorial of a fixed number.

Implementation Challenges

Scheduling usually involves 3 challenges that must work all together or you might find yourself with zombie processes or processes that do not obey the RMS policy:

1. Waking your application when it is ready to run. RMS has a very strict release policy and does not allow the job of any application to run before its period. This means that your application *must sleep until the beginning of the period without any busy waiting* or you will waste valuable resources that could have been used for other applications. This challenge clearly states that our applications will have various states in the kernel:
    - READY, in which an application has reached the beginning of the period and a new job is ready to be scheduled.
    - RUNNING, in which an application is currently executing a job. This application is currently using the CPU.
    - SLEEPING, in which an application has finished executing the job for the current period and is waiting for the next period.
2. Preempting the *current application* when another application with a *higher priority is ready to run*. This involves triggering a context switch. You will use the Linux Scheduler API for this.
3. Preempting an application that has *finished its current job*. To achieve this, assume that the application always behaves correctly and notifies the scheduler that it has finished its job for the current period. Upon receiving a yield message from `/proc/mp2/status`, the RMS must put the application to sleep until the next period. This involves setting up a timer and preempting the CPU to the next ready application with the highest priority.

To address these three challenges, you will need to augment the Process Control Block of each task. You will need to include the application state (READY, SLEEPING, RUNNING), a wakeup timer for each task, and the scheduling parameters of the task, including the period of the application (which denotes the priority in RMS). Also, the scheduler will need to keep a list or run queue of all registered tasks to pick the correct task to schedule during any preemption point. An *additional challenge* is performance – CPU schedulers must minimize their overhead. Floating point arithmetic is very expensive and therefore it must be avoided (although you shouldn't use floating point arithmetic in the kernel at all, if possible).

Implementation Overview

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

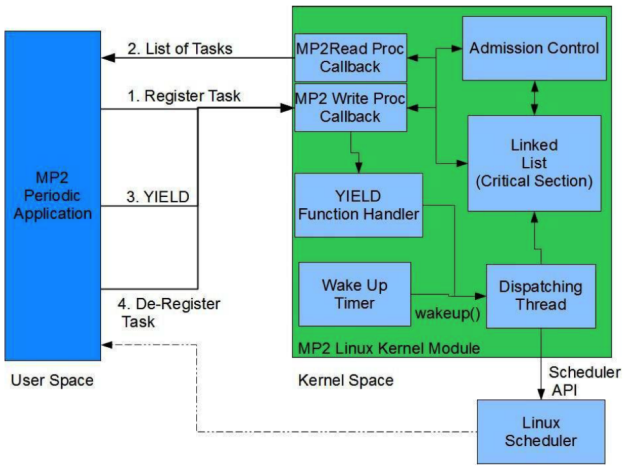

Figure 2: MP2 Architecture Overview

1. Start from an empty (e.g., "Hello World!") Linux Kernel Module.
2. Begin setting up the `procfs` entry `/proc/mp2/status`. For now, just use a dummy `read` implementation (e.g.,

output nothing). We recommend setting up your framework to distinguish the three types of messages for your `write` implementation.

Note that the first character of a message specifies its type ('R', 'Y', or 'D'); thus, you can switch on this character to know what kind of message must be processed. You can either leave the message handlers empty or use `printk` to ensure you recognize the correct types of messages; the latter will allow you to run some basic tests.

3. Augment the process control block (PCB). We are not going to directly modify the Linux PCB (`struct task_struct`), but instead declare a separate data structure that points to the corresponding PCB of each task.

   Create a new struct containing a pointer to a `struct task_struct`, which is defined in `<linux/sched.h>`. We recommend you index your list by PID. To obtain the `struct task_struct` associated with a given PID, we have provided you with a helper function in `mp2_given.h`.

   Add any other state you need per task to your new struct, including the period, the processing time, a wakeup timer for the task, etc. Your data structure should look something like this:

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

4. Implement *registration*. Do not worry about admission control at this point, you will implement admission control in Step 8. Allow any task for now. To implement registration go back to the empty registration function you set up in Step 2.

   In this function, you must allocate and initialize a new `struct mp2_task_struct`. We will use the *slab allocator* for memory allocation of the `struct mp2_task_struct`. 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()`). During your kernel module initialization, you must create a new cache of size `sizeof(struct mp2_task_struct)`. This new cache will be used by the registration function to allocate a new `struct mp2_task_struct` instance.

   The registration function must initialize the `struct mp2_task_struct`. Initialize the task to be in the SLEEPING state. However, let the task run until the application emits a yield message indicating the real-time loop. Until it reaches the real-time loop, do not enforce any scheduling. You will then need to insert this new struct into the list of tasks. This step is very similar to what you did in MP1.

   You should also implement `read` for `/proc/mp2/status`, as you now have the necessary information to do so.

5. Implement *de-registration*. 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 registering and de-registering some tasks.

6. In Tasks 6 and 7, you will implement the wakeup and preemption mechanisms. Before you implement anything, let's analyze how our scheduling will work:

   You will have a kernel thread (*dispatching thread*) that is responsible for triggering context switches as needed. The dispatching thread will sleep the rest of the time. There will be two cases in which a context switch will occur:

   1. after receiving a yield message from the application, and
   2. after the wakeup timer of the task expires.

   When `/proc/mp2/status` receives a yield message, it should put the associated application to sleep and setup the wakeup timer. It should also change the task state to SLEEPING. When the wakeup timer expires, the timer interrupt handler should change the state of the task to READY and should wake up the dispatching thread. **The timer interrupt handler must not wake up the application!**

   Step 6 will implement the dispatching thread and preemption mechanism. Step 7 will implement the yield handler.

   Start by implementing the dispatching thread. As soon as the dispatching thread wakes up, you will need to find the highest priority (i.e., shortest period) READY task in the list. Then, you need to preempt the currently

running task (if any) and context switch to the chosen task. If there are no tasks in READY state, simply preempt the currently running task. Set the new task's state to RUNNING. The state of the preempted task must be set to READY only if the state was RUNNING. This is because the yield handler will set the preempted task's state to SLEEPING.

To handle the context switches and preemption, you will use the scheduler API based on some known behavior of the Linux scheduler. Any task running on `SCHED_FIFO` will hold the CPU for as long as the application needs. So you can trigger a context switch by using the function `sched_setscheduler()`.

You can use the functions `set_current_state()` and `schedule()` to get the dispatching thread to sleep. For the new running task, the dispatching thread should execute the following code:

```
struct sched_param sparam;
wake_up_process(task);
sparam.sched_priority=99;
sched_setscheduler(task, SCHED_FIFO, &sparam);
```

Wake up the task at this point, not in the wakeup timer handler. Similarly, for the preempted task, the dispatching thread should execute the following code:

```
struct sched_param sparam;
sparam.sched_priority=0;
sched_setscheduler(task, SCHED_NORMAL, &sparam);
```

We recommend you keep a global variable of type `struct mp2_task_struct *` pointing to the current running task. This will simplify your implementation. This practice is common and it is even used by the Linux kernel (i.e., `current`). If there is no running task you can set this pointer to `NULL`.

Next, implement the wakeup timer handler. As mentioned in the overview to Step 6, the handler should change the state of the task to READY and should wake up the dispatching thread. You can think of this mechanism as a two-halves handler where the top half is the wakeup timer handler and the bottom half is the dispatching thread.

7. Implement *yield*. In this function, change the state of the calling task to SLEEPING. You need to calculate the next release time (i.e., the beginning of the next period), set the timer, and put the task to sleep as `TASK_UNINTERRUPTIBLE`. You can use the macro `set_task_state(struct task_struct *, TASK_UNINTERRUPTIBLE)` to change the state of any task.

   Note: you must only set the timer and put the task to sleep if the next period has not started yet. If you set a timer with a negative value, the two's complement representation of signed numbers will result in an extremely large unsigned number and the task will freeze.

8. Implement *admission control*. The admission control should check if the current task set and the task to be admitted can be scheduled without missing any deadlines according to the utilization bound-based method. If the task cannot the accepted, then the scheduler must simply not allow the task in the system.

   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 tasks in the system including the task to be admitted, 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 arithmetic. Floating point support is very expensive in the kernel and should be avoided at all costs. Instead, use fixed-point arithmetic implemented via integers.

9. Implement the test application and ensure that your scheduler is behaving as expected. It is recommended that you test with multiple instances of the test application and with different periods and computation loads. When testing, you should run the application with various periods and number of jobs.

10. Ensure that you are properly freeing all memory. This is especially true for the module exit function. For this MP, you do not have to worry about tasks that do not perform de-registration before the module is terminated. We will assume all the tasks behave well.

Software Engineering

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 postconditions of the algorithm. Some functions might have as few as one line of comments, while others might have a longer paragraph.

Also, your code must be split into small functions, even if these functions contain no parameters. This is a common situation in kernel modules because many variables are declared as global, including (but not limited to) data structures, state variables, locks, timers and threads.

An important problem in kernel code readability is to know if 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. The kernel uses macros and preprocessor commands extensively. Proper use of macros and proper identification of possible situations where they should be used are important issues in kernel programming.

Finally, in kernel programming, the use of the `goto` statement is a common practice, especially when handling cleanup for memory allocation within a function. 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.

Submission

To submit your code, ensure you have done both of the following:

1. Turn your code in via git by committing it to your course repo. You can do this via the following commands:

```
git add -A
git commit -m "MP submission"
git push origin master
```

Adding a Tag

We will grade your latest pre-deadline commit for each checkpoint and the final MP. However, if you want us to grade a specific commit, you can tag it with a specific tag. Assuming that your tagged commit is before the deadline, we will grade your tagged commit instead of your latest pre-deadline commit. This will allow you to continue to work on your MP after ensuring you have made your submission for the checkpoint.

The tag names to use are as follows:

- For Checkpoint #1: `mp2-cp1`
- For Checkpoint #2: `mp2-cp2`
- For the final submission: `mp2-final`

(If you do not have a tag with the name, we will grade your latest pre-deadline commit+push. It's not necessary to add the tag, but it may give you extra assurance. :))

Grading Criteria

Criterion
Are read/write for `/proc/mp2/status` correctly implemented?
Is your process control block correctly implemented/modified?

Criterion
Is registration correctly implemented (admission control, using slab allocator)?
Is deregistration correctly implemented?
Is the wakeup timer correctly implemented?
Is yield correctly implemented?
Is the dispatching thread correctly implemented?
Are your data structures correctly initialized/freed?
Does your test application follow the specified application model?
Documentation describing implementation details and design decisions
Your code compiles and runs correctly, and does not use any floating point arithmetic
Your code is well commented, readable and follows software engineering principles

References

1. https://bootlin.com/doc/training/linux-kernel/linux-kernel-slides.pdf
2. The Linux Kernel Module Programming Guide. https://tldp.org/LDP/lkmpg/2.6/html/index.html
3. Linux Kernel Linked List Explained. https://rootfriend.tistory.com/entry/Linux-Kernel-Linked-List-Explained
4. Kernel API's Part 3: Timers and lists in the 2.6 kernel. https://developer.ibm.com/technologies/linux/tutorials/l-timers-list/
5. Access the Linux Kernel using the Proc Filesystem (a little outdated, but still useful). https://developer.ibm.com/technologies/linux/articles/l-proc/
6. Linux kernel threads (site now down; internet archive link here). https://web.archive.org/web/20161123122310/http://www.crashcourse.ca:80/wiki/index.php/Kernel_threads
7. Kernel Korner - Sleeping in the Kernel. https://www.linuxjournal.com/article/8144
8. Love Robert, Linux Kernel Development, Chapters 3, 4, 6, 9-12, 17-18, AddisonWesley Professional, Third Edition
9. Bovet Daniel, Understanding the Linux Kernel, Chapter 10, O'Reilly
10. Lui Sha, Rajkumar Ragunathan & Shrish Sathaye, Generalized Rate-Monotonic Scheduling Theory: A Framework for Developing Real-Time Systems. In Proceeding of the IEEE. Vol. 82. No. 1, Jan 1994,(pp. 68-82).
11. Liu C, Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment, Journal of ACM, Volume 20, Issue 1, Jan 1973.
12. Slab allocator. (Note: the API has changed, but this still explains how the slab allocator works.) https://www.kernel.org/doc/gorman/html/understand/understand011.html
13. Memory management APIs. (Look for `kmem_cache` documentation). https://www.kernel.org/doc/html/latest/core-api/mm-api.html

1. Please note that in this document, we use the term application and task interchangeably. ↩