CS423 Spring 2020
MP2: Rate-Monotonic CPU Scheduling

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

2 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 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*.

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

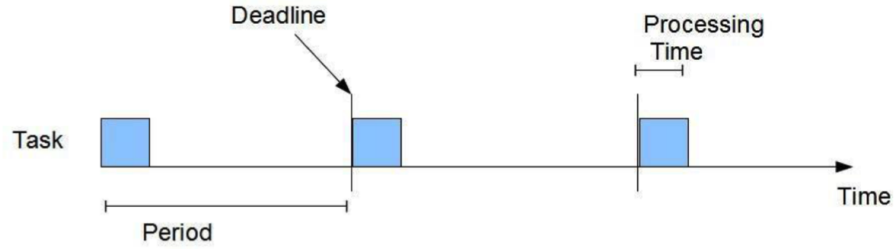

Figure 1: Liu and Layland Periodic Task Model

next period, referred to as deadline D. As part of the model, each job requires certain processing time C. Figure 1 illustrates this model.

In this MP we 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*[1] *with a higher priority will always preempt a task with a lower priority until its processing time has been used.*

3 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 Virtual Machine, you will be able to turn it on, and off using the VMWare vSphere Console. Inside your Virtual Machine 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 Virtual Machine.

Finally, you are encouraged to discuss design ideas and bugs in Piazza. Piazza 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.

[1]Please note that in this documentation we will use the term application and task interchangeably.

4 Problem Description

In this MP we will implement a Real-Time CPU scheduler based on the Rate-Monotonic Scheduler (RMS) for a Single-Core Processor. We will implement our scheduler as a Linux Kernel module and we will use the Proc filesystem to communicate between the scheduler and the user space applications. We will use a *single Proc filesystem entry* for all the communication (`/proc/mp2/status`), readable and writable by any user. Our scheduler should implement three operations available through the Proc filesystem:

- *Registration*: This allows the application to notify to Kernel module its intent to use the RMS scheduler. The application also communicates its registration parameters to the kernel module (PID, Period, Processing Time).
- *Yield*: This operation notifies the RMS scheduler that the application has finished its period. After a yield, the application will block until the next period.
- *De-Registration*: This allows the application to notify the RMS scheduler that the application has finished using the RMS scheduler.

Our Rate-Monotonic scheduler will only register a new periodic application if the application's scheduling parameters pass through the *admission control*. The admission control must decide if the new application can be scheduled along with other already admitted periodic application without missing any deadlines for any of the 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 cases where the application will use more than the reserved Processing Time (Overrun Cases).*

This scheduler will rely on the Linux Scheduler to perform context switches and therefore you should use the Linux Scheduler API. You do not need to handle any low-level functions. Please read Sections 5 and 6 for more information.

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 a list with the PID, Period and Processing Time of each application.

We will also develop a simple test application for our scheduler. This application will be a single threaded periodic application with individual jobs calculating factorials. This periodic application must register itself with the scheduler (through admission control). During the registration process it must specify its scheduling parameters:

The Period of the jobs expressed in milliseconds and Processing Time of each job also expressed in milliseconds.

After the registration the application must read the Proc filesystem entry to ensure that its PID is listed. This means the task is accepted. After this, the application must signal the scheduler that it is ready to start by sending a YIELD message through the Proc filesystem. 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. At the end of the Real-Time Loop, the application must de-register after finishing all its periodic jobs. The de-registration of the application from the scheduler is done using the Proc filesystem. Below you can find the pseudo code of the Periodic Application:

```
void main(void)
{
REGISTER(PID, Period, JobProcessTime); //Proc filesystem
list=READ STATUS(); //ProcFS: Verify the process was admitted
if (!process in the list) exit 1;
//setup everything needed for RT loop: t0=gettimeofday()
YIELD(PID); //Proc filesystem
//this is the real-time loop
while(exist jobs)
{
do_job(); //wakeup_time=gettimeofday()-t0 and factorial computation
YIELD(PID); //ProcFS. JobProcessTime=gettimeofday()-wakeup_time
}
UNREGISTER(PID); //ProcFS
}
```

To determine the processing time of a job you can run the application using the Linux scheduler first and measuring the average execution time of one iteration of the Real-Time Loop.

During the real-time loop our test application must print the wake-up time of the process and the time spent during the computation of each job. You can use the `gettimeofday()` function. This value will be useful to test your scheduler.

Additionally your application can perform a simple computation, we recommend calculating the factorial of a fixed number. This MP is about Real-Time scheduling, so keep the application simple.

5 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) The first challenge involves waking your application when it is ready to run. Rate Monotonic has a very strict release policy and does not allow the job of any application to run before its period. This means that our application *must sleep until the beginning of the period without any busy waiting* or we will waste valuable resources that can be used to schedule other applications. This challenge clearly states that our applications will have various states in the kernel:

   – **READY** state in which an application has reached the beginning of the period and a new job is ready to be scheduled.

   – **RUNNING** state in which an application is currently executing a job. This application is currently using the CPU.

   – **SLEEPING** state in which an application has finished executing the job for the current period and it is waiting for the next period.

2) The second challenge involves *preempting an application that has higher priority than the current application as soon as this application becomes available to run*. This involves triggering a context switch. We will use the Linux Scheduler API for this.

3) The third challenge is to preempt an application that has finished its current job. To achieve this we will 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 the Proc filesystem, the RMS scheduler 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.

From these three challenges we know that we will need to augment the Process Control Block of each task: We will need to include the application state (READY, SLEEPING, RUNNING), a wake up 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 a run queue of all the registered tasks so we can pick the correct task to schedule during any preemption point. An *additional challenge* is performance – CPU scheduler must minimize its overhead. Floating Point arithmetic is very expensive and therefore it must be avoided.

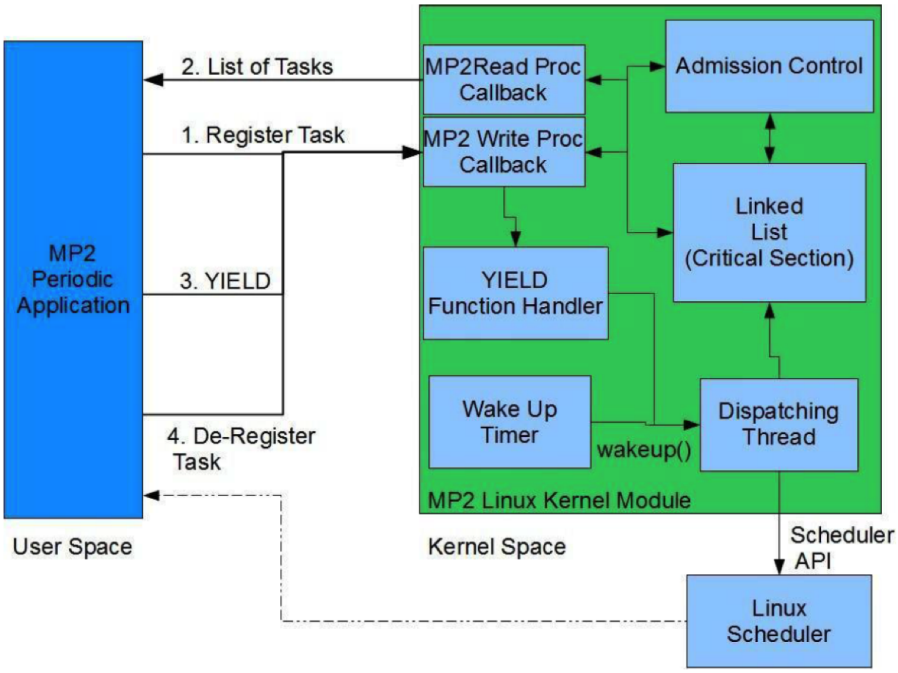

Figure 2: MP2 Architecture Overview

6 Implementation Overview

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

1. The best way to start is by implementing an empty ('Hello World!') Linux Kernel Module. You should also be able to reuse some of the most generic functions you implemented on MP1, like linked list helper functions and so.

2. After this you should 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 step 2 of implementation you may leave the functions empty or simply print a message using `printk()`, but you will implement them then in full functionality during the steps 7 and 8.

We recommend adding an operation character at the beginning and performing the switch operation over that character. This allows you to receive various types of messages with a single Proc filesystem entry and provide a single unified interface. As an example we show the string formats for each the Proc Filesystem messages:

- For REGISTRATION: “R, PID, PERIOD, COMPUTATION”
- For YIELD: “Y, PID”
- For DE-REGISTRATION: “D, PID”

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

3. You should 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 and add 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 state of the task, including the period, the processing time, a wake up 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. Now you should be able to implement *registration*. Do not worry about admission control at this point, we will implement admission control in Step 8. Allow any task for now. To implement registration go back to the empty registration function from Step 2.

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

The registration function must initialize mp2_task_struct. We will initialize the task in SLEEPING state. However, we will let the task run until the application reaches

the YIELD message as indicated by the Real-Time Loop. Until we reach the Real-Time loop we do not enforce any scheduling. You will need then to insert this new structure into the list of tasks. This step is very similar to what you did in MP1.

As part of this step you should also implement the Read callback of the Proc Filesystem entry.

5. 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 registering and de-registering some tasks.

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

We will have a kernel thread (*dispatching thread*) that is responsible for triggering the 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 the Proc filesystem callback receives a YIELD message, it should put the associated application to sleep and setup the wakeup timer. Also it should 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!*

In Step 6 we will implement the dispatching thread and the kernel mechanism. In Step 7 we will implement the YIELD handler function.

6a. Let's start by implementing the dispatching thread. As soon as the context switch wakes up, you will need to find in the list, the task with READY state that has the highest priority (that is the shortest period). 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 we should simply preempt the currently running task. The task state of the old task must be set to READY only if the state is RUNNING. This is because we will previously set the state of the old task to SLEEPING in the YIELD handler function. Also you must set the state of the new running task to RUNNING.

To handle the context switches and the preemption we will use the scheduler API based on some known behavior of the Linux scheduler. We know that any task running on the SCHED_FIFO will hold the CPU for as long as the

application needs. So we 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);
```

At this point is where we wake up the task and not in the wake up timer handler. Similarly, for the old running task (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 with the current running task (struct mp2_task_struct*). This will simplify your implementation. This practice is common and it is even used by the Linux kernel. If there is no running task you can set it to NULL.

6b. Now we should implement the wake up timer handler. As mentioned before 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 where the top half is the wake-up timer handler and the bottom half is the dispatching thread.

7. In this step we will implement the YIELD handler function from the Proc filesystem callback that we left blank from Step 2. In this function we need to change the state of the calling task to SLEEPING. We need to calculate the next release time (that is the beginning of the next period), we must set the timer and we must 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 other task.

Please note that 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 notation of signed numbers will result in a too large unsigned number and the task will freeze.

8. You should now implement the admission control. The admission control should check if the current task set and the task to be admitted can be scheduled without missing any deadline 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. Floating-Point support is very expensive in the kernel and should be avoided at all cost. Instead use Fixed-Point arithmetic implemented through integers.

9. You should go back and make sure that you are properly destroying and de- allocating all the 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.

10. Now implement the test application and make sure that your scheduler is behaving as expected. It is recommended that you test with multiple instances of the test application and different periods and computation loads. For testing and demo purposes your test application should print the start time and finish time of every job 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 a command line parameter.

7 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 post-conditions of the algorithm. Some functions might have as few as one line comments, while some 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 most of the 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. Usually the code 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.

Another important aspect is that Floating Point Units are very slow and not always available. Therefore the use of Floating Point arithmetic must be avoided whenever possible. Most general purpose code can be implemented using Fixed Point arithmetic.

Finally, in kernel programming, the use of the `goto` statement is a common practice. 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.

8 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 following the Application Model work?	10
Document Describing the implementation details and design decisions	5
Your code compiles and 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**

9 References

1. http://free-electrons.com/doc/training/linux-kernel/linux-kernel-slides.pdf
2. The Linux Kernel Module Programming Guide http://tldp.org/LDP/lkmpg/2.6/html/index.html
3. Linux Kernel Linked List Explained http://isis.poly.edu/kulesh/stuff/src/klist/
4. Kernel API's Part 3: Timers and lists in the 2.6 kernel http://www.ibm.com/developerworks/linux/library/l-timers-list/
5. Access the Linux Kernel using the Proc Filesystem http://www.ibm.com/developerworks/linux/library/l-proc/index.html
6. Linux kernel threads http://www.crashcourse.ca/wiki/index.php/Kernel_threads
7. Love Robert, Linux Kernel Development, Chapters 3, 4, 6, 9-12, 17-18, Addison-Wesley Professional, Third Edition
8. Bovet Daniel, Understanding the Linux Kernel, Chapter 10, O'Reilly
9. 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).
10. Liu C, Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment, Journal of ACM, Volume 20, Issue 1, Jan 1973
11. Linux slab allocator http://www.makelinux.net/ldd3/chp-8-sect-2 (Note: The API has changed. See include/linux/slab.h)