***REAL TIME - RATE MONOTONIC SCHEDULING***

Muhammad Omer Haqqani 12-2069

Rafay Habibullah 12-2176

***DESCRIPTION:***

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 kernel module. 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

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.

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.

***Problems Faced:***

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

a) 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.

b) 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.

c) 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.

d) 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.

e) An additional challenge is performance. A CPU scheduler must minimize its overhead. Floating Point arithmetic is very expensive and therefore it must be avoided.

**PROBLEMS SOLUTIONS:**

**Step 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.

**Step 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.

**Step 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; … }

**Step 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.

**Step 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.

**Step 6:** Before we implement anything, let's analyze how our schedule works:

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 2. After the wakeup\_timer of 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.

**Step 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=MAX\_USER\_RT\_PRIO-1;

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 not uncommon and it is even used by the Linux kernel. If there is no running task you can set it to NULL.

Step 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.

**Step 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.

**Step 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.

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.

**Step 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.

**Step 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.

***The Actual Working Of The Project:***

Many real-time systems use preemptive multitasking, especially those with an underlying real-time operating system (RTOS). Priorities are assigned to tasks, and the RTOS always executes the ready task with highest priority.

In this case, the scheduling algorithm is the method in which priorities are assigned. Most algorithms are classified as fixed priority, dynamic priority, or mixed priority. A fixed-priority algorithm assigns all priorities at design time, and those priorities remain constant for the lifetime of the task. A dynamic-priority algorithm assigns priorities at runtime, based on execution parameters of tasks, such as upcoming deadlines. A mixed-priority algorithm has both static and dynamic components. Needless to say, fixed-priority algorithms tend to be simpler than algorithms that must compute priorities on the fly.

To demonstrate the importance of a scheduling algorithm, consider a system with only two tasks, which we'll call t1 and t2. Assume these are both periodic tasks with periods T1 and T2, and each has a deadline that is the beginning of its next cycle. Task t1 has T1 = 50ms, and a worst-case execution time of C1 = 25ms. Task t2 has T2 = 100ms and C2 = 40ms. Note that the utilization, Ui, of task ti is Ci/Ti. Thus U1 = 50% and U2 = 40%. This means total requested utilization U = U1 + U2 = 90%. It seems logical that if utilization is less than 100%, there should be enough available CPU time to execute both tasks.

Let's consider a static priority scheduling algorithm. With two tasks, there are only two possibilities:

Case 1: Priority(t1) > Priority(t2)  
Case 2: Priority(t1) <>2)

***REFRENCE:***

[1] Linux Kernel Programming Slides on webpage

[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-a simple introduction, http://kerneltrap.org/node/20903

[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] Anatomy of the Linux slab allocator http://www.ibm.com/developerworks/linux/library/l-linux-slab-allocator/