CS423 : MP2 Documentation

Group 2: Debjit Pal (dpal2), Anirudh Jayakumar (ajayaku2), Neha Chaube (nchaub2) and Divya Balakrishna (dbalakr2)

March 2, 2015

1 Introduction

We describe the important data structures and the functionalities of the functions we created to build up *Rate Monotonic Scheduler* (RMS). We used a multi-file approach to keep the modularity of the design and to make sure that all the group members can work in parallel without harming others' code. Hence we changed the `Makefile` provided in MP1. In Section 1.1, we enlist the different files relevant to this work, in Section 1.2, we describe the changes that we made in the `Makefile`. In Section 1.3 we demonstrate the augmented process control block (PCB) that we created for this project. Section 1.4, we demonstrate the creation of `proc` file system, the APIs to `read` from and `write` into the `proc` filesystem. In this section we also demonstrate the usage of the `procfile_write` call back for process *registration, de-registration* and *yield*. In Section 1.5 we demonstrate the `admission_control` procedure which in turn used by the `procfile_write` function to check if a new process can be admitted without violating deadlines of any already registered process. In this section we also describe `remove_task` which removes the task from the linked list when it is done with its computation. In Section 1.6 we describe the functions that are needed to handle linklists in the Kernel. Section 1.7 demonstrates the working of the dispatching thread and enlists the necessary functions we wrote. Section 3 demonstrates step-by-step running procedure of the RMS module and the application. Section 4 concludes the documents along with our learnings and the steps we took to debug the code.

1.1 Files included in the MP2 Project

The list of the included files for this MP is given in Table 1.

1.2 Important Changes in Makefile

We took a multi-file approach to make sure each of the group member can work on different parts of the implementation, we modified the given `Makefile` from MP1 in the following way:

```
obj-m += mp2_final.o
mp2_final-objs := linklist.o thread.o mp2.o
```

The name of the object module has been changed to `mp2_final.o` as mentioned in the `obj-m`. That `mp2_final.o` contains three different object modules as mentioned in the `mp2_final-objs`. The *ordering* of the object module is important here because mp2.o contains a call to a dispatch

Table 1: File list included in the project

`Makefile`	To build up the Kernel module and user application called *my_factorial*
`structure.h`	Defining the augmented process control block, the functions and the other data types and data structures relevant to this project
`mp2.c`	Includes the kernel init and exit modules, proc file creation, proc file read and write function, admission control
`linklist.h`	Declares all interface functions to read and write into the linklist
`linklist.c`	Definitions of all the interface functions to read and write in the linked list
`thread.c`	Implements the dispatcher thread
`thread.h`	Contains the function declarations for the dispatcher thread implementation
`process.c`	Implementation of the parametrized factorial user application

thread function that is defined in thread.o. Hence, thread.c needs to be compiled before else gcc will give a compiler error message `"implicit function declaration found"`. Further, mp2.c contains the `MODULE_LICENSE` declaration and hence it needs to be compiled at the very end after all other C files are compiled else gcc will pop an error message `MODULE_LICENSE declaration not found`. If we load such a kernel module without `MODULE_LICENSE` definition, the kernel will get *tainted* and may crash. In our project, the name of the kernel module is `mp2_final.ko` which we load in the kernel using *insmod* command.

1.3 Process Control Block

As per the MP documentation, we did not modify the process control block (PCB) of Linux directly. Rather we augment it by creating our own PCB data structure that points to the corresponding PCB of each task. We added a pointer of type `struct task_struct` and indexed the augmented PCB by the process ID or PID. The structure that we defined is shown in Figure 1.

The augmented process control block stores the `period` and the `computation` time in miliseconds (ms). On a new process being admitted through the `admission_control`, the parameter values supplied by the process are copied to this structure member variables. We also annotate the current state of operation of the process using a `enum` of type `state` defined in `structure.h`. We also include a `struct sched_param` strcuture which will store the `real-time priority value` of the corresponding process.The `mytimer` of type `struct timer_list` is the associated `wakeup timer` for the process. The pointer `task` of type `struct task_struct` points to the Linux PCB of the corresponding task. `list_node` keeps track of the process in the linked list.

```
typedef struct process_entry {
     pid_t pid;
     ulong period;
     ulong computation;
     ulong c;
     enum state states;
     struct sched_param sparam;
     struct timer_list mytimer;
     struct task_struct *task;
     list_node mynode;
} my_process_entry;
```

Figure 1: Augmented Process Control Block Structure

1.4 Proc file system creation, process registration, de-registration and yield

The kernel module `mp2_final.ko` on being loaded into the kernel, the `__init` functions creates the **proc** file system using **proc_filesystem_entries** function. **proc_filesystem_entries** first creates a directory called **mp2** under the **/proc** filesystem using bf proc_mkdir() and then it uses **proc_create()** to create the status file **status** at the location **/proc/mp2** with a **0666** permission so the user application can read and write at the status file. The **/proc/mp2/status** has an associated `read` and `write` function namely **procfile_read** and **procfile_write** respectively. Both of them are defined using **file_operations** structure.

The **procfile_write** is one of the main function of this RMS kernel module. The **procfile_write** first copies the data from user space to kernel space using`copy_from_user())`. As suggested in the MP2 documentation guideline, we will check the first character of the data and will know whether the process is calling for `registration, de-registration` or to `yield`. For the registration, the function `admission_control()` is called (described in Section 1.5) to check whether the new process can be registered or its request will be denied. If the admissibility test passes, then an object of type `my_process_entry` is created to store the parameters of the new process, with the associated timers being set up using `init_timer()`. The object is then appended at the end of the linked list. For yield, some parameters are updated and the process is put into the `SLEEP` (with a `TASK_UNINRERRUPTIBLE` parameter)) by triggering a context switch using `sched_scheduler()`. For de-registration, we find the object of type `my_process_entry` associated with the process and remove it from the linked list and also free the data structures that was dynamically allocated during the time of registration. For this, we use a `remove_task()` function which uses `del_timer()` to delete the timer associated with the process and then uses `list_del()` to remove the process from the linked list.

The **procfile_read** needs a tricky implementation. The function should be aware about the fact that it has read the proc file system buffer completely and no more data is left to be transferred to user space application. Otherwise user space application (for example "cat /proc/mp2/status"command) would output the content of the **/proc/mp2/status** indefinitely. We use **copy_to_user()** to copy data from proc filesystem buffer to user space buffer. When **cat /proc/mp2/status** is executed in user space, **procfile_read** outputs the `PID, Period` and the `Computation Time` of the currently registered processes.

On kernel module being unloaded from the kernel, `__exit` function is called. This function calls

remove_entry to remove proc file system entries using **remove_proc_entry()**. The `__exit` also cleans up workerthread and the linked list. A summary of the implemented functions are given in the Table 2.

Table 2: List of functions to insert and remove kernel module, read and write proc file system, process registration, de-registration and process yield

`static int __init mp2_init(void);`	Initializing the kernel module, initialize linkedlist and initialize timer.
`procfs_entry* proc_filesys_entries(char *procname, char *parent);`	Creates the **mp2** directory in **/proc** and the status file in **/proc/mp2**
`static ssize_t procfile_read (struct file *file, char __user *buffer, size_t count, loff_t *data);`	Used to read data from kernel space to user space
`static ssize_t procfile_write(struct file *file, const char __user *buffer, size_t count, loff_t *data);`	Used to read data from user space to kernel space
`static void remove_entry(char *procname, char *parent);`	Removes the status file, the **mp2** directory from **proc** filesystem, removes the worker thread and the linkedlist
`int admisson_control(my_process_entry *new_process_entry);`	Decides whether the new process can be admitted or denied registration
`int remove_task(pid_t pid)`	Removes the task specified by the PID from the linked list

1.5 Admission Control Implementation

The admission control function `admission_control` traverses the linked list and evaluates the sum of the *utilization ratio* of the new process requesting registration and currently registered processes i.e. $\mathcal{U} = \sum_{i=1}^{n} u_i = \sum_{i=1}^{n} \frac{c_i}{p_i}$ where c_i is the computation time, p_i is the period of the process and n is the sum total of the number of processes that are already registered plus the new process. To avoid any *floating point calculation*, we calculate the *utilization ratio* by multiplying each c_i by 1000 and then dividing by p_i. The `admission_control` function checks whether $\mathcal{U} \leq 693$ and admits if the predicate is true else denies the registration of the new process.

1.6 Functionalities of Linklist

The process structure of the registered processes are stored in a linklist. A set of interface methods provide read and write access to the linklist. These function are declared in linklist.h header file. These methods are used by the proc read/write module and the dispatch thread. The list of interfaces and their functionalities are explained in the Table 3.

Table 3: List of functions to access the linklist

`int ll_initialize_list(void)`	Initializes the linklist and should be called before calling any other linklist function. Ideally, this should be called from the kernel module init function.
`int ll_add_task(my_process_entry *proc)`	Adds a process_entry structure instance to the list.
`int ll_generate_proc_info_string(char **buf, unsigned int *size)`	Generates a string with all the currently registered processes and their period and computation time.
`int ll_cleanup(void);`	Frees all memory created during initialize. Should be called from module_exit function
`int ll_remove_task(pid_t pid)`	removes the process structure from the list and then delete it
`int ll_get_size(void);`	Returns the size of the list
`int ll_find_high_priority_task(my_pro **proc)`	return the task which is in READY state and having the least period
`int ll_get_task(pid_t pid, my_process_entry **proc)`	return the process structure with process id equal to pid

1.7 Implementing dispatching thread

kthreads of LINUX is used to implement the dispatching thread. This forms the bottom half for pre-emption and scheduling of user process. As soon as the timer wakes up the dispatching thread, the list containing all the processes is traversed to fetch the process with highest priority and which is in READY state. There are two possibilities of the current state: 1.If the current task is in RUNNING state, the state is changed to READY state and the priority is set to 0. The task is scheduled using sched_setscheduler()with SCHED_NORMAL parameter. 2. If there is no current task, the new task fetched from the list is set to RUNNING state, it's priority is set to 99 and the process is scheduled with SCHED_FIFO policy. A summary of the implemented functions are given in the Table 4.

Table 4: List of functions to initialize and cleanup Dispatching thread

`int thread_init(void);`	Initialize the thread using kthread_create()
`thread_callback(void* data);`	Implement worker thread to pre-empt and schedule the task
`ll_find_high_priority_task(&node);`	used to traverse the list to fetch process with highest priority
`int wake_thread(void)`	used to wake up the dispatching thread
`void thread_cleanup(void)`	used to stop the dispatching thread

2 User Application

We implemented an user application which is called `process` and built whenever the kernel module is built using the `Makefile`. The application takes three parameters namely

1. Period
2. Computation time
3. Number of jobs

A typical invocation of the process looks like:

$$\underbrace{./process}\ \underbrace{1000}_{\text{period}}\ \underbrace{100}_{\text{computation}}\ \underbrace{100}_{\text{number of jobs}}$$

The process implements the following functions as shown in Table 5.

Table 5: List of functions implemented in the user application named `process`

`void myregister(pid_t pid, unsigned long int period, unsigned long int computation, char *file);`	Registers the process in the proc file system with the user specified parameters.
`int read_status(char *file);`	Read the status of the registration of the process from proc file system. If successfully registered returns size of the proc size buffer read else returns `0`
`unsigned long factorial(unsigned long i);`	Calculates factorial of a pre-specified number
`void yield(pid_t pid,char *file);`	Sends request for yield
`void unregister(pid_t pid, char *file) ;`	When the process is complete with user specified number of jobs, this functions request for de-registration through the proc file system

3 How to run the program

Please use the following steps to compile, insert and remove the kernel module and to run our application *my_factorial* program.

1. To compile the kernel module and the user application:

   ```
   # make
   ```

   This will create `mp2_final.ko` kernel object module and the user application *my_factorial*.

2. To insert the kernel module:

   ```
   # sudo insmod mp2_final.ko
   ```

This should print a few confirmation messages in the **/var/log/kern.log** file confirming that the kernel module has been loaded successfully.

3. To run the user application:

```
# ./process 10 500 100 (./process <period> <computation> <number of jobs>)
```

4. To output the CPU time of the running processes in konsole:

```
# cat /proc/mp2/status
```

5. To unload the kernel module:

```
# sudo rmmod mp2_final
```

This will print a few confirmation messages in the **/var/log/kern.log** file confirming that the kernel module has been unloaded successfully.

4 Conclusion

In this MP we implement a kernel module performing real time scheduling following the principle of Rate Monotonic Scheduler. We also implement a test application to test the kernel module. In this MP we learnt

1. To use the real time priorities
2. Use of dispatching thread for context switch implementation, to send the process from the READY to RUNNING and SLEEPING states.
3. We learnt the usage of the timer.