

# CS423 : MP1 Documentation

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

February 15, 2015

## 1 Introduction

We describe the important data structures and the functionalities of the functions we created to build up this kernel module. 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 needed to change the provided `Makefile` file. In Section 1.1, we demonstrate 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 implement the creation of `proc` file system, the APIs to `read` from and `write` into the `proc` filesystem. Section 1.4 describes the usage of the kernel linked-list and the necessary functions that we wrote to implement that. Section 1.5 details the lock type and the reason for choosing it. Section 1.6 describes the implementation of the `work queue` and `timer`. Finally, Section 1.7 enlists our learning from this machine problem, details how to run the program and concludes the document.

### 1.1 Files included in the MP1 Project

Table 1: File list included in the project

<code>Makefile</code>	To build up the Kernel module and user application called <i>my_factorial</i>
<code>mp1.c</code>	Includes the kernel init and exit modules, proc file creation, proc file read and write function
<code>linklist.c</code>	Implements the update function for the kernel linklist
<code>linklist.h</code>	Contains the function declarations for the linklist implementation
<code>Work_Queue.c</code>	Implements the workqueue, timer, interrupt using the two halves concept
<code>workqueue.h</code>	Contains the function declarations of for the workqueue and timer
<code>my_factorial.c</code>	Implementation of the user application
<code>my_factorial.h</code>	Header file for the user application

## 1.2 Important Changes in Makefile

In the given Makefile we had the following line where only a single object file (namely `mp.o`) created from the single source file (namely `mp1.c`) and in turn a kernel module is created (namely `mp1.ko`).

```
obj-m:= mp1.o
```

Since we took a multi-file approach to make sure each of the group member can work on different parts of the implementation, we changed the Makefile in the following way:

```
obj-m += mp1_final.o
mp1_final-objs := linklist.o Work_Queue.o mp1.o
```

The name of the object module has been changed to `mp1_final.o` as mentioned in the `obj-m`. That `mp1_final.o` contains three different object modules as mentioned in the `mp1_final-objs`. The *ordering* of the object module is important here because `mp1.o` contains a call to a linkedlist function that is defined in `linklist.o`. Similarly, `Work_Queue.o` contains function call that is defined in `linklist.o`. Hence, `linklist.c` needs to be compiled before else gcc will give a compiler error message "implicit function declaration found". Further, `mp1.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 `mp1_final.ko` which we load in the kernel using `insmod` command.

## 1.3 Proc File System Creation

The kernel module `mp1_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 **mp1** under the **/proc** filesystem using `proc_mkdir()` and then it uses `proc_create()` to create the status file **status** at the location **/proc/mp1** with a **0666** permission so the user application can read and write at the status file. The **/proc/mp1/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 `__init` function also initializes the kernel linkedlist and the workqueue.

The `procfile_writes` data from user space using the API `copy_from_user()` from the user buffer named `buffer` to the dynamically created proc file system buffer named `procfs_buffer`. If all the user space data cannot be copied into the proc file system buffer, a `-EFAULT` signal is sent. On success, the process ID namely `pid` is added to the kernel linked list and the bytes amount that are copied from user space to kernel space is returned.

The `procfile_read` needs a tricky implementation. The function is 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/mp1/status" command) would output the content of the **/proc/mp1/status** indefinitely. We use `copy_to_user()` to copy data from proc filesystem buffer to user space buffer. When `cat /proc/mp1/status` is executed in user space, `procfile_read` outputs the PID of the currently registered processes and the CPU time of the 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 the linkedlist from kernel memory, destroys the workqueue and the timer thread. 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

<code>static int __init mp1_init(void);</code>	Initializing the kernel module, initialize linkedlist and initialize timer.
<code>procfs_entry* proc_filesys_entries(char *procname, char *parent);</code>	Creates the <b>mp1</b> directory in <b>/proc</b> and the status file in <b>/proc/mp1</b>
<code>static ssize_t procfile_read (struct file *file, char __user *buffer, size_t count, loff_t *data);</code>	Used to read data from kernel space to user space
<code>static ssize_t procfile_write(struct file *file, const char __user *buffer, size_t count, loff_t *data);</code>	Used to read data from user space to kernel space
<code>static void remove_entry(char *procname, char *parent);</code>	Removes the status file and the <b>mp1</b> directory from <b>proc</b> filesystem

## 1.4 Functionalities of Linklist

We use a linklist data structure to store the PIDs and their CPU usage time. The linklist representing the process structure is called `process_info` and its declaration is shown below.

```
struct process_info {
    int pid;
    unsigned long cpu_time;
    struct list_head list;
};
```

This data structure is encapsulated through a set of interfaces available in the `linklist.h` header file. These functions are used by the proc read/write module and the work queue module to interface with the linklist. The list of interfaces and their functionalities are explained in the Table 3.

## 1.5 Locking mechanisms

We use a read-write semaphore to synchronize the access to the linklist. The read-write semaphore works as follows.

- Allows multiple concurrent readers. In our case, we could have the interrupt(bottom half) and the proc read/write module access the linklist. These accesses can occur concurrently.
- Only one writer can access the critical section.

Table 3: List of functions to access the linklist

<code>int ll_initialize_list(void);</code>	Initializes the linklist and should be called before calling any other linklist function. Ideally, this should be called from the kernel module init function.
<code>int ll_add_to_list(int pid);</code>	Adds a PID to list. Returns DUPLICATE if the entry is already present
<code>int ll_generate_cpu_info_string(char **buf, int *count);</code>	Generates a string with all the PIDs and their CPU times
<code>int ll_update_time(int pid, unsigned long cpu_use);</code>	Updates the CPU time of a process
<code>int ll_cleanup(void);</code>	Frees all memory created during initialize. Should be called from module_exit function
<code>int ll_is_pid_in_list(int pid);</code>	Checks if a PID is in the list. Return PASS/FAIL
<code>int ll_get_pids(int **pids, int *count);</code>	Gets an array of PIDs and their count. Caller should free the array after use.
<code>int ll_delete_pid(int pid);</code>	Deletes a PID from the list
<code>int ll_list_size(void);</code>	Returns the size of the list
<code>void ll_print_list(void);</code>	Prints the list to kernel logs

- Writers get priority. This means that if a writer tries to enter the critical section then no reader will be allowed to access the critical section till all writers have completed their tasks. This could lead to reader starvation.

The read-write semaphore is ideal for situations where most of the access to the data are reads with few write access in between. In our kernel module, the linklist is only updated on two occasions a) every 5 seconds with the new CPU time and b) when a new process is registered by writing the PID to the proc entry.

## 1.6 Implementation and working principles of Work Queue, Timer and Interrupt Handler: Two Halves Approach

The necessary functions that we implemented are enlisted in Table 4 along with their functionalities. The interrupt handler needs to perform tasks quickly and also not keep interrupts blocked for a long period of time. Keeping this in mind, the *Two Halves approach* is used. The Top Half is used to schedule the interrupt and cannot sleep while the Bottom Half is used to execute the work once it receives an interrupt from the Top half. In this case, the Kernel timer(Top Half) is being used to schedule the timer every five seconds and once the timer is triggered, the bottom half is called which updates CPU time of the registered processes stored in the linked list.

The Top Half consists of Kernel Timer which triggers an interrupt to the bottom half every 5 seconds. Initialization of timer has been done in the `_init` function using `initialize_timer()`. The timer structure parameters are initialized in this function. The timer expiry parameter is set to 5 seconds using jiffies. Data can be passed to the timer handler using data parameter of the timer. The timer callback function has to be registered with the timer structure. The timer

Table 4: List of functions for WorkQueue and Timer

<code>void initialize_timer(void);</code>	Used to initialize timer and set timer structure parameters.
<code>void timer_callback(unsigned long data);</code>	Used to submit work to the workqueue and modify timer if user process is running.
<code>void modify_timer(void);</code>	Used to activate an active timer.
<code>int get_cpu_use(int pid, unsigned long *cpu_use);</code>	Used to fetch the CPU time of a particular process
<code>void work_handle(struct work_struct *work);</code>	Used to update cpu time in linklist
<code>void create_work_queue(void);</code>	Used to create workqueue.
<code>int init_workqueue(void);</code>	Used to call <code>initialize_timer()</code> and <code>create_work_queue()</code> .
<code>void cleanup_workqueue();</code>	Used to flush workqueue, flush any existing work using <code>cancel_delay_work()</code> API, delete timer and workqueue

becomes inactive after the `timer_callback` is executed. To activate it again, `mod_timer()` is used.

The timer interrupt handler of Top Half wakes up the Bottom Half which consists of a work function `work_handler( struct work_struct *work )` in a workqueue. Work is submitted to the workqueue when the interrupt is triggered. The `work_handler()` gets the pids of all the processes running in the user space and registered in this kernel module. For each running process, the CPU time is updated and stored in the linked list.

## 1.7 Learning and Conclusion

We have learnt the following from this MP:

1. Handling of proc file system, data exchanges between kernel space and user space
2. Usage of mutex locks to protect kernel link list data from being corrupted and to make sure read and write operations are done atomically.
3. Two halves approach to handle interrupt

We faced the following problem while integrating the different parts of the kernel module code and debugged

1. Earlier **procfile\_read** was reading the status file indefinitely. This has been debugged using a offset value which keeps track of the amount of the data that has been already sent to the user space.

### 1.7.1 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 `mp1_final.ko` kernel object module and the user application *my\_factorial*.

2. To insert the kernel module:

```
# sudo insmod mp1_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:

```
# ./my_factorial 10 (factorial of the argument is calculated)
```

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

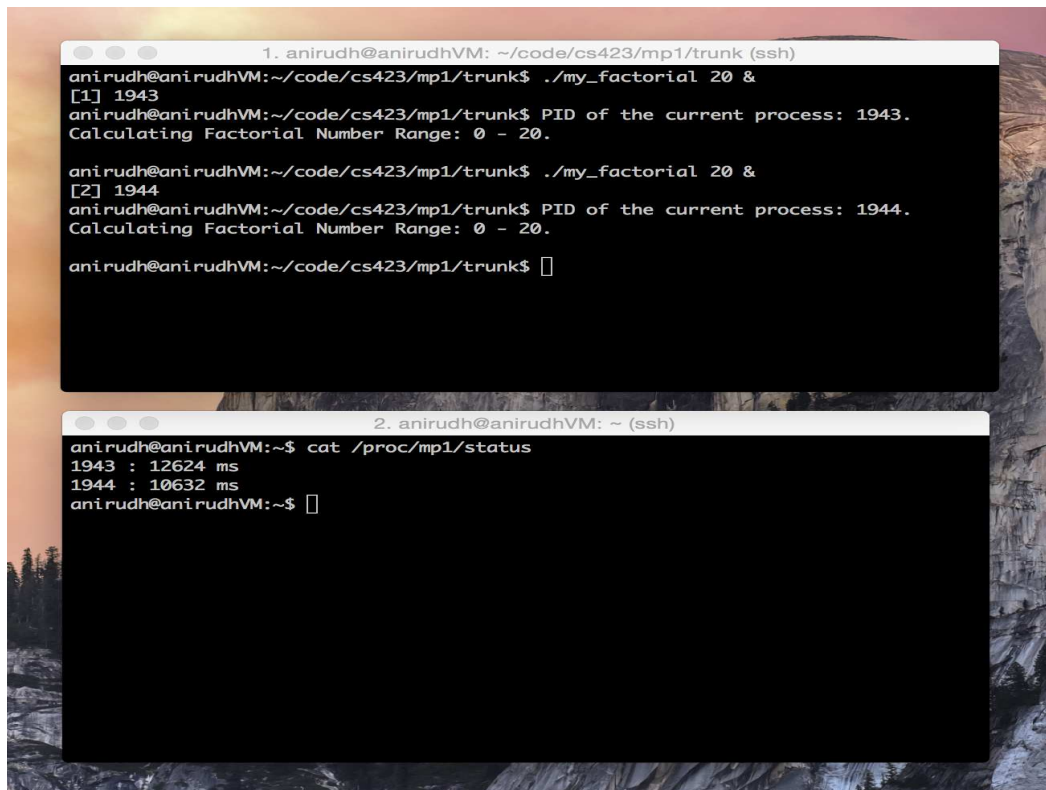
```
# cat /proc/mp1/status
```

5. To unload the kernel module:

```
# sudo rmmod mp1_final
```

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

A screenshot of the output when running the `cat /proc/mp1/status` is given in Figure 1:



The image displays two terminal windows from a desktop environment. The first window, titled '1. anirudh@anirudhVM: ~/code/cs423/mp1/trunk (ssh)', shows the execution of a program named `my_factorial` with the argument `20`. It runs twice, with the first instance (PID 1943) and the second instance (PID 1944) both calculating the factorial number range from 0 to 20. The second window, titled '2. anirudh@anirudhVM: ~ (ssh)', shows the command `cat /proc/mp1/status` being executed, which outputs the execution time for each process: 12624 ms for PID 1943 and 10632 ms for PID 1944.

```
1. anirudh@anirudhVM: ~/code/cs423/mp1/trunk (ssh)
anirudh@anirudhVM:~/code/cs423/mp1/trunk$ ./my_factorial 20 &
[1] 1943
anirudh@anirudhVM:~/code/cs423/mp1/trunk$ PID of the current process: 1943.
Calculating Factorial Number Range: 0 - 20.

anirudh@anirudhVM:~/code/cs423/mp1/trunk$ ./my_factorial 20 &
[2] 1944
anirudh@anirudhVM:~/code/cs423/mp1/trunk$ PID of the current process: 1944.
Calculating Factorial Number Range: 0 - 20.

anirudh@anirudhVM:~/code/cs423/mp1/trunk$ █

2. anirudh@anirudhVM: ~ (ssh)
anirudh@anirudhVM:~$ cat /proc/mp1/status
1943 : 12624 ms
1944 : 10632 ms
anirudh@anirudhVM:~$ █
```

Figure 1: Running `cat` command