CS423 : MP3 Documentation

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

April 5, 2015

1 Introduction

We describe the important data structures and the functionalities of the functions we created to build up *Virtual Memory Page Fault Profiler*. 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* and *de-registration*. In Section 1.5 we demonstrate the memory buffer allocation procedure. In Section 1.7 we describe the functions that are needed to handle linked-lists in the Kernel. Section 1.6 creation of Character Device and `mmap` callback. Section 1.8 demonstrates the working of the work-queue and timer. Section 2 demonstrates how to run the kernel module. Section 4 concludes the documents along with our learnings and the steps we took to debug the code.

1.1 Files included in the MP3 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 += mp3_final.o
mp3_final-objs := linklist.o mem.o char_dev.o workqueue.o mp3.o
```

The name of the object module has been changed to `mp3_final.o` as mentioned in the `obj-m`. That `mp3_final.o` contains three different object modules as mentioned in the `mp3_final-objs`. The *ordering* of the object module is important here because mp3.o contains a call to functions in linklist, mem, char_dev and workqueue. Hence, it should compiled before else gcc will give a compiler error message ‘‘`implicit function declaration found`". Further, mp3.c contains the `MODULE_LICENSE` declaration and hence it needs to be compiled at the very end after all other C

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
`mp3.c`	Includes the kernel init and exit modules, proc file creation, proc file read and write function
`linklist.h`	Declares all interface functions to read and write into the linkedlist
`linklist.c`	Definitions of all the interface functions to read and write in the linked list
`workqueue.c`	Implements the work queue functionalities along with timer implementation
`workqueue.h`	Contains the function declarations for workqueue and timer implementation
`mem.c`	Implementation of memory buffer allocation using `vmalloc`
`mem.h`	Definitions of all the interface functions to implement memory buffer allocation using `vmalloc`

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 `mp3_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.

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

The kernel module `mp3_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 **mp3** under the **/proc** filesystem using bf proc_mkdir() and then it uses **proc_create()** to create the status file **status** at the location **/proc/mp3** with a **0666** permission so the user application can read and write at the status file. The **/proc/mp3/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 kernel module. The **procfile_write** first copies the data from user space to kernel space using`copy_from_user()`. As suggested in the MP3

```
typedef struct process_info {
     int pid;
     ulong start_jiff;
     ulong last_sample_jiff;
     ulong min_flt;
     ulong maj_flt;
     ulong cpu_util;
     struct list_head list;
     struct task_struct *task;
} my_process_entry;
```

Figure 1: Augmented Process Control Block Structure

documentation guideline, we will check the first character of the data and will know whether the process is calling for `registration` or `de-registration`. For de-registration, we find the object of type `my_process_info` 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.

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/mp3/status"command) would output the content of the **/proc/mp3/status** indefinitely. We use **copy_to_user()** to copy data from proc filesystem buffer to user space buffer.

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 work queue, character device 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 and de-registration

`static int __init mp3_init(void);`	Initializing the kernel module, initialize linkedlist and initialize timer.
`procfs_entry* proc_filesys_entries(char *procname, char *parent);`	Creates the **mp3** directory in **/proc** and the status file in **/proc/mp3**
`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 **mp3** directory from **proc** filesystem, removes the character device and the linkedlist
`int remove_task(pid_t pid)`	Removes the task specified by the PID from the linked list

1.5 Allocation of Memory Buffer

Memory buffer is allocated in memory using `vmalloc()` when kernel module is initialized and freed when module is unitialized. A set of interface methods provide initialization, writing and freeing of the memory buffer. These function are declared in `mem.h` header file. The list of interfaces and their functionalities are explained in the Table 3.

Table 3: List of functions to allocate memory for mmap buffer

`int mm_initialize_list(void)`	Allocates memory for mmap buffer using vmalloc()Ideally, this should be called from the kernel module init function.
`int mm_add_data(sampling_data *data)`	Adds sampling data in the memory buffer allocated.
`void *mm_get_buffer(void)`	returns a pointer to mmap_buf.
`int mm_cleanup(void)`	Frees all memory created during initialize. Should be called from module_exit function
`void mm_set_mem_index(void)`	sets the memory index to 0

1.6 Implementing character device

Table 4: List of functions to initialize and cleanup Character Device

`static int mmap_callback(struct file *fp, struct vm_area_struct *vma)`	mmap callback function
`int cd_initialize(void)`	used to create and register character device
`int cd_cleanup(void)`	used to unregister and delete the character device

A summary of the functions used to create character device file are given in the Table 4.

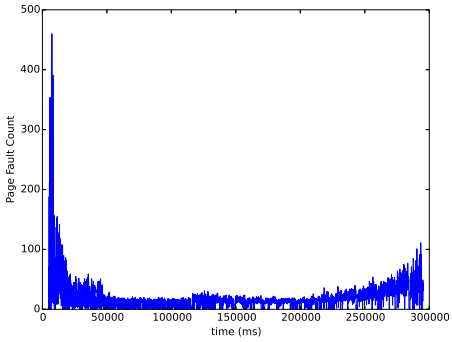
1.7 Functionalities of Linklist

We use a linklist data structure to store the PIDs and the sampled data. The linklist representing the process structure is called `process_info` and its declaration is shown in Table 5.

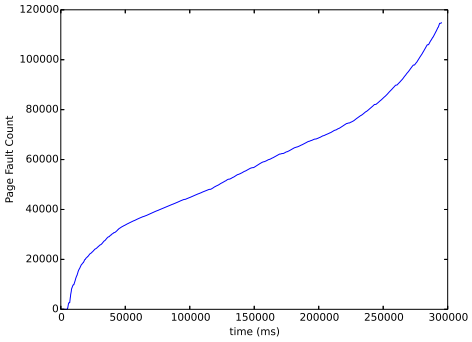
1.8 Implementation and working principles of Work Queue and Timer: Two Halves Approach

The necessary functions that we implemented are enlisted in Table 6 along with their functionalities. The Kernel timer(Top Half) is being used to schedule the timer every 50 mili seconds (20 samples per second) and once the timer is triggered, the bottom half gets the sampled data (major faults, minor faults and CPU utilization) and updates it in the linked list.

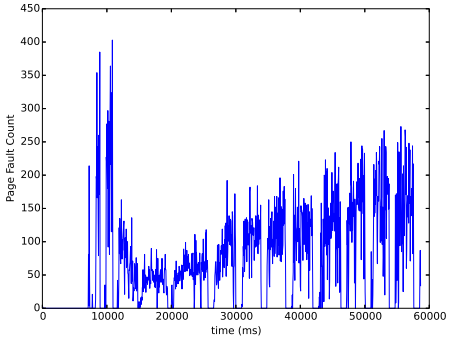
Initialization of timer has been done in the `_init` function using `initialize_timer()`. The timer structure parameters are initialized in this function. The timer expiry parameter is set to 50 mili seconds using jiffies. Data can be passed to the timer handler using data parameter of



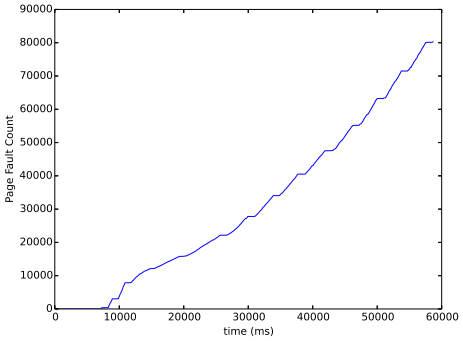
(a) Both processes use random access



(b) Cumulative fault count for Figure 2a



(c) One process uses random access and the other process uses local access



(d) Cumulative fault count for Figure 2c

Figure 2: Case Study I: Thrashing and Locality

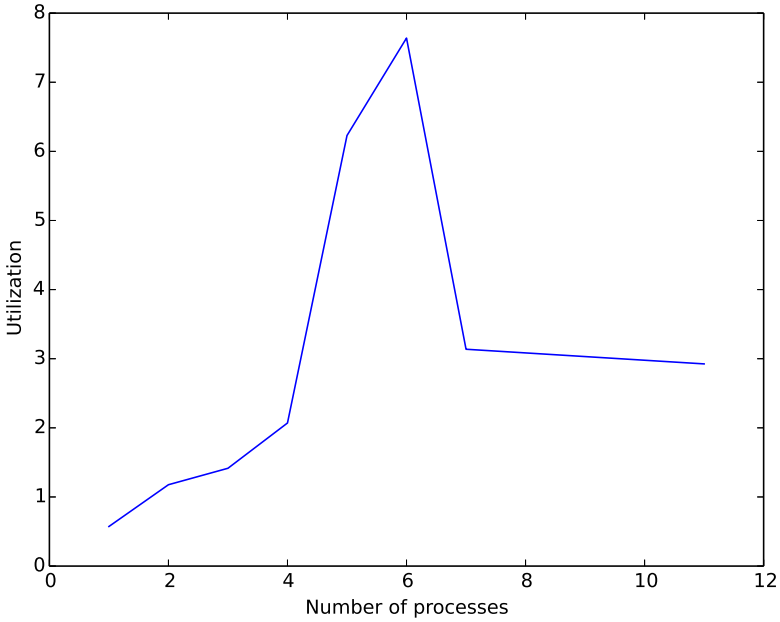


Figure 3: Case Study II: Multi-programming

the timer. The timer callback function has to be registered with the timer structure. The timer 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 sampled data is updated and stored in the linked list.

2 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 `mp3_final.ko` kernel object module.

2. To insert the kernel module:

   ```
   # sudo insmod mp3_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 along with creating a character device file **node**.

Table 5: 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(struct process_info *new_proc);`	Adds a PID to list. Returns DUPLICATE if the entry is already present
`int ll_generate_proc_info_string(char **buf,int *count);`	Generates a string with all the PIDs and their CPU times
`int ll_update_item(int pid,ulong min_flt_,ulong maj_flt_,ulong cpu_util_,ulong last_jiff);`	Updates the sampled data of a process
`int ll_cleanup(void);`	Frees all memory created during initialize. Should be called from module_exit function
`int ll_is_pid_in_list(int pid);`	Checks if a PID is in the list. Return PASS/FAIL
`int ll_get_pids(int **pids, int *count);`	Gets an array of PIDs and their count. Caller should free the array after use.
`int ll_delete_item(int pid);`	Deletes a PID from the list
`int ll_list_size(void);`	Returns the size of the list
`void ll_print_list(void);`	Prints the list to kernel logs
`int ll_get_last_sample_jiff(int pid, ulong *prev_jiff);`	Gets the previous jiffies count

3. To run the user application:

   ```
   # nice ./work 1024 R 50000 & nice ./work 1024 L 10000 &
   ```

4. To run the monitor application:

   ```
   # ./monitor > profile2.data
   ```

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

   ```
   # cat /proc/mp3/status
   ```

6. To unload the kernel module:

   ```
   # sudo rmmod mp3_final
   ```

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

Table 6: List of functions for WorkQueue and Timer

`void initialize_timer(void);`	Used to initialize timer and set timer structure parameters.
`void timer_callback(unsigned long data);`	Used to submit work to the workqueue and modify timer if user process is running.
`void modify_timer(void);`	Used to activate an active timer.
`void work_handler(struct work_struct *work);`	Used to update cpu time in linklist
`void create_work_queue(void);`	Used to create workqueue.
`int init_workqueue(void);`	Used to call `initialize_timer()` and `create_work_queue()`.
`void cleanup_workqueue();`	Used to flush workqueue, flush any exisiting work using `cancel_delay_work()` API, delete timer and workqueue

3 Analysis

1. **Case study I:** It is evident from Figure 2a and Figure 2c that the total number of page faults in profile 1 is more than page faults in profile 2. It will clearer when you look at Figrue 2b and Figure 2d, which gives the cumulative count of faults. There are close to 120k faults in profile 1 and 80K faults in profile 2. These results are consistent with our understanding of case study 1. Profile 1 has both processes accessing memory randomly whereas in profile 2 one of the process in accessing locally. Local access will have less page faults due to spacial locality of memory access. The page that got swapped in is expected to contain the future references to memory and hence there will be less page fault.

2. **Case Study II:** The general understanding of multi-programming is that the CPU utilization will increase as more process will try to get hold of the CPU. But problem arises when there is less memory to hold all these processes. In such a scenario, each process will page fault, resulting in the kernel spending time to bring pages in and out of the RAM. This will result in CPU being utilized in the kernel context. This will result in lower CPU utilization by the user process. So, the expected plot should be similar to a bell curve where the utilization will increase till the point when page faults become an overhead. This is consistent with the plot shown in Figure 3.

4 Conclusion

In this MP we implement a kernel module performing profiling of page faults and CPU utilization.