# CS423 Spring 2015
# MP1: Introduction to Linux Kernel Programming
# Due Feb. 16th at 9:00 am

**Note: The Engineering IT hasn't set up the VMs yet. We'll assign VMs to the groups once they are available from the Engineering IT.**

## 1. Goals and Overview

• In this Machine problem you will learn the basics of Linux Kernel Programming
• You will learn to create a Linux Kernel Module (LKM).
• You will use Timers in the Linux Kernel to schedule work
• You will use Workqueues to defer work through the use of the Two-Halves concept
• You will use the basics of the Linked Lists interface in the Linux Kernel to temporarily store data in the kernel
• You will learn the basic of Concurrency and Locking in the Linux Kernel
• You will interface applications in the user space with your Kernel Module through the Proc Filesystem

## 2. Development Setup

You will work on the provided Virtual Machine (Lubuntu 14.04.1 LTS 64 bits) and you will develop kernel modules for the Linux Kernel 3.13.0-44-generic provided in this Virtual Machine. You will have full access and control of your Virtual Machine, you will be able to turn it on, and off using the VMWare Server Console. While the Virtual Machine is designed for development, you will be responsible of keeping it fairly secure by choosing **strong password** for each of the accounts in the VM.

Your Virtual Machine has all the development tools required to compile Linux Kernel Modules. There is no need to download or install any other kernel source. Installing any other kernel source is discouraged because there has been significant changes in recent versions of the Linux Kernel that might affect your implementation. Also, you do not need to recompile the kernel, patch it or modify the main source in any way. The MP should be implemented **only through Linux kernel modules**.

**All the grading will be done using kernel 3.13.0-44-generic.** Inside your Virtual Machine you will find various editors including gedit, Emacs and Vim. Also two users have been created for you: A regular user CS423 with sudo access (password: cs423), and a TA-only user CS423ta.

The first account is for you to develop and test your code. Please remember to change the password of this account as soon as possible. The second account will be used by the TA's for troubleshooting purposes. **Do not change the password of the TA account**.

## 3. Introduction

Kernel Programming has some particularities that can make it more difficult to debug and learn. In this section we will discuss a few of them.

The most important difference between kernel programming in Linux and Application programming in the user space is the **lack of Memory Protection**. That is driver,

modules, and kernel threads all share the same memory address space. De-referencing a pointer that contains the wrong memory location and writing to it can cause the whole system to crash or corrupt important subsystems including filesystem and networking.

Another important difference is that in kernel programming, **preemption is not always available**, that means that we can indefinitely hog the CPU or cause system-wide deadlocks. This makes concurrency much more difficult to handle in the kernel, than in user space. For example in kernel space we are responsible for ensuring that interrupt handlers are as efficient as possible using the CPU for very little time.

Also another important issue is the **lack of user space libraries**. Glib, C++ Standard Library and other libraries reside in the user space and cannot be accessed in the kernel. This limits what we can do and how do we implement it. Another important difference is that Linux Kernel lacks from Floating-Point support. That is all the math must be implemented using integers. Also files, signals or security descriptors are not available.

Through the rest of the document and your implementation you will learn some of the basic mechanisms, structures and designs common to many areas of Linux Kernel Development. **Please consult the recommended links and tutorials in the References Section of this document as those documents detail everything that you need to implement this MP**.

## 4. Problem Description

In this MP you will build a kernel module that measures the User Space CPU Time of processes registered within the kernel module and a simple test case application that requests this service. In a real scenario many applications might be using this functionality implemented by our new kernel module and therefore our module is designed to support multiple applications/processes to register simultaneously.

The kernel module will allow processes to register themselves through the Proc Filesystem. For each registered process, the kernel module should write to an entry in the Proc Filesystem, the application's User Space CPU Time (known also as user time). The kernel module must keep these values in memory for each registered process and update them every 5 seconds. Figure 1 shows the application interface with the kernel module using the Proc filesystem.

The registration process must be implemented as follows: At the initialization of your kernel module, it must create a directory entry within the Proc filesystem (e.g. */proc/mp1*). Inside this directory your kernel module must create a file entry (e.g. */proc/mp1/status*), readable and writable by anyone (mask 0666). Upon start of a process (e.g. our test application), it will register itself by writing its PID to this entry that you created. When a process reads from this entry, the kernel module must print a list of all the registered PIDs in the system and its corresponding User Space CPU Times. An example of the format your proc filesystem entry can use to print this list is as follows:
PID1: CPU Time of PID1

PID2: CPU Time of PID2

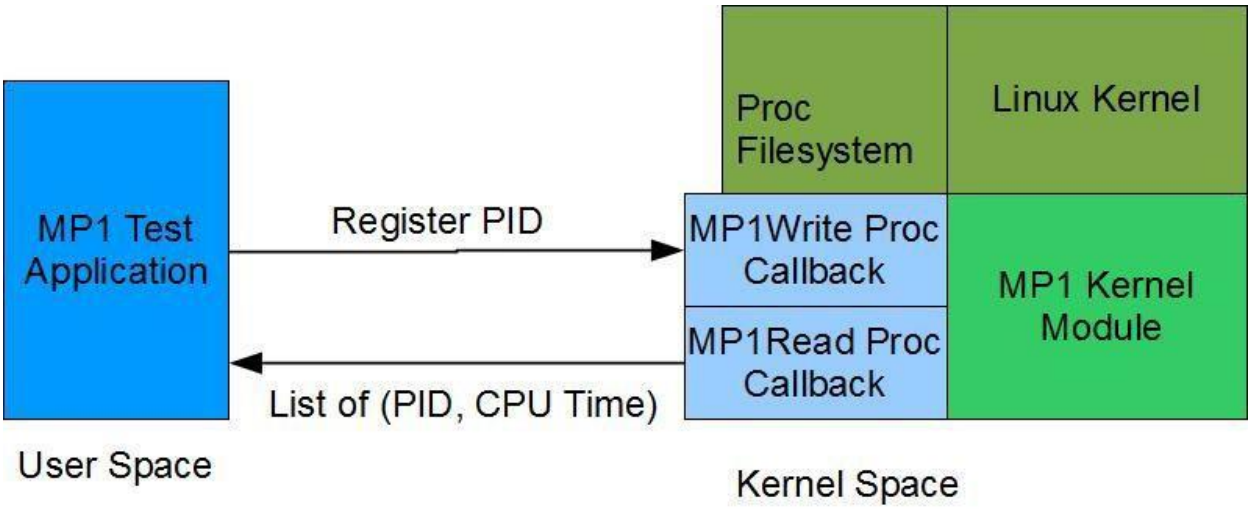


**Figure 1: Proc Filesystem Interface between Test Application and MP1 Kernel Module**

Your kernel module implementation must store the PIDs and the CPU Time values of each process in a Linked List using the implementation provided by the Linux kernel. Part of the goals of this MP is that you learn to use this facility provided by the kernel. Additionally, the CPU Time values of each process must be periodically updated by using a Kernel Timer. However, you must use a technique called **Two-Halves approach**. In this approach an interrupt is divided in two parts: Interrupt Handler (Top-Half) and Thread performing the work (Bottom-Half). The slides on Linux Kernel Programming, posted on compass explain this concept more in detail.

In our case the Top-Half will be the Timer Interrupt Handler, its sole purpose will be to wake up the Bottom-Half. For the Bottom-Half we will use a *work function* in a workqueue. A workqueue is a kernel mechanism that allows you to schedule the execution of a function (work function) at a later time. A worker thread managed by the kernel is responsible of the execution of each of the work functions scheduled in the workqueue. In our MP, the work function will traverse the link list and update the CPU Time values of each registered process.

It is acceptable for your work function to be scheduled even if there are no registered processes, however you might consider an implementation where the timer is not scheduled if there are no registered processes. Figure 2 on Section 6 shows the architecture of the kernel module you should implement, including the timer interrupt and the workqueue.

Finally, the kernel module will be responsible of freeing any resource that it allocated during its execution. This includes stopping any work function pending, destroying any timer, workqueue or Proc filesystem entry created and freeing any allocated memory.

As a test case you must implement a simple program that registers itself in the kernel module using the proc filesystem and then calculates a series of factorial computations.

This computation can repeat or can be different. However, this program should run for sufficient time to test your kernel module, sometime between 10 and 15 seconds should be sufficient. At the end of the computation the application must read the proc file system entry containing the list of all the registered applications and its corresponding CPU Times.

**5. Implementation Challenges**

In this MP you will find many challenges commonly found in Kernel Programming. Some of these challenges are discussed below:

a) During the registration process you will need to **access data from the User Space**. Kernel and Applications both run in two **separate memory spaces**, so de-referencing pointers containing data from the User Space is not possible. Instead you must use the function *copy_from_user()* to copy the data into a buffer located in kernel memory. Similarly, when returning data through a pointer we must copy the data from the kernel space into the user space using the function *copy_to_user()*. Common cases where this might appear are in Proc filesystem callbacks and system calls.

b) Another important challenge is the **lack of libraries**, instead the kernel provides similar versions of this commonly used functions found in libraries. For example *malloc()* is replaced with *kmalloc()*, *printf()* is replaced by *printk()*. Some other handy functions implemented in the kernel are *sprintf()*, and *sscanf()*. This functions are introduced in [2].

c) Throughout your implementation, you will need to face **different running contexts**. A context is the entity whom the kernel is running code on behalf of. In the Linux kernel you will find 3 different contexts:
   1. Kernel Context: Runs on behalf of the kernel itself. Example: Kernel Threads and workqueues
   2. Process Context: Runs on behalf of some process. Example: System Calls
   3. Interrupt Context: Runs on behalf of an interrupt. Example: Timer Interrupt

d) The Linux kernel is a **preemptible kernel**. This means that all the contexts run concurrently and can be interrupted from its execution at any time. You will need to protect your data structures through the use of appropriate **locks** and prevent **race conditions** wherever they appear. Please note that architectural reasons limit which type of locks can be used for each context. For example, **interrupt context code cannot sleep and therefore semaphores will create a deadlock when used in this case**.

This sleeping restriction in interrupt context also prevents you from using various functions that sleep during its execution. Many of these functions involve complicated operations that require access to devices like *printk()*, functions that schedule processes, copy data from and to the user space, and functions that allocate memory (e.g *kmalloc()*). Some exceptions to this rule of thumb are the function w*ake_up_process()* and the function *kmalloc()* when used with special flags.

Due to all these challenges, we recommend you that you test your code often and build in small increments. You can use the *BUG_ON()* macro to spot inconsistencies and trigger a stack dump.

## 6. Implementation Overview

In this section we will briefly guide you through the implementation. Figure 2 shows the architecture of MP1, showing the kernel module with its workqueue and timer and also the proc filesystem all in the kernel space. In the user space you can see the test application that you will also implement.

**Step 1:** The best way to start is by implementing an **empty ('Hello World!') Linux Kernel Module**.

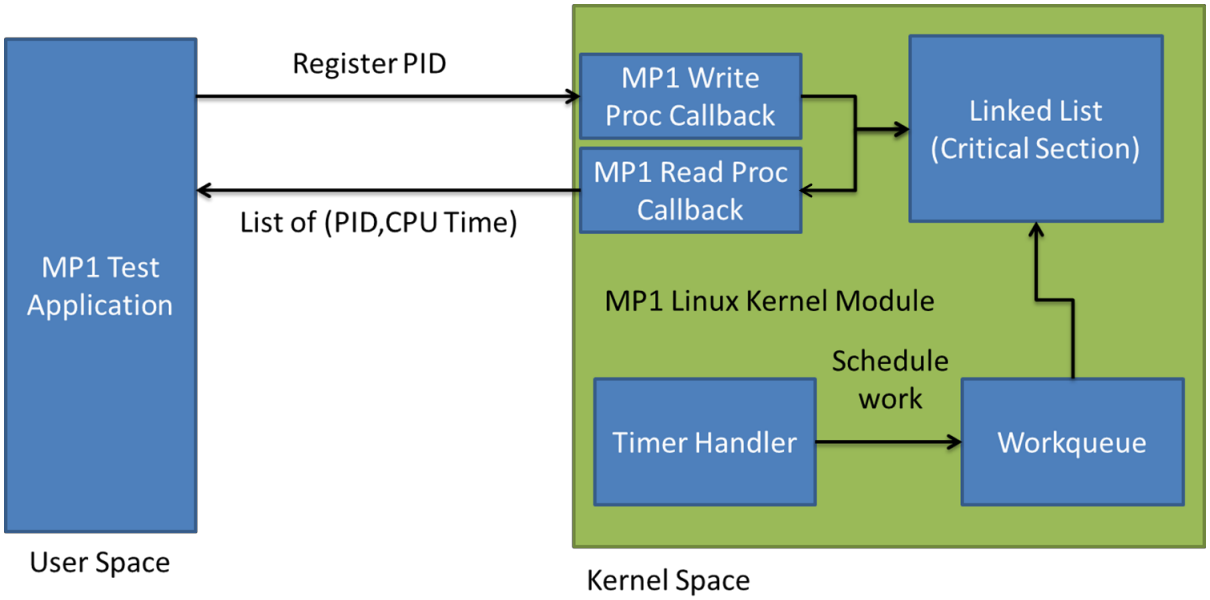


**Figure 2: MP1 Architecture Overview**

**Step2:** After this you should implement the **Proc Filesystem entries** *(i.e /proc/mp1/ and /proc/mp1/status)*. Make sure that you implement the creation of these entries in your module "init" function and the destruction in your module "exit" function.

At this point you should probably test your code. Compile the module and load it in memory using *insmod* or *modprobe*. You should be able to see the proc filesystem entries you created using ls. Now remove the module and check that the entries are properly removed.

**Step 3:** The next step should be to implement the **full registration**, you will need to declare and initialize a Linux kernel Linked List. The kernel provides macros and functions to traverse the list, and insert and delete elements.

**Step 4:** You will also need to implement the **callback functions for read and write** in the entry of the proc filesystem you created. Keep the format of the registration string simple. We suggest that a user space application should be able to register itself by simply writing the PID to the proc filesystem entry you created (e.g "*pid*">/mp1/status). The callback functions will read and write data from and to the user space so you need to use *copy_from_user()* and *copy_to_user()*. To keep things simple, do not worry about adding support for page breaks in the reading callback.

**Step 5:** At this point you should be able to write a **simple user space application that registers itself** in the module. Your test application can use the function *getpid()* to obtain its PID. You can open and write to the proc filesystem entry using *fopen()* and *fprintf()*, or you can use *sprintf()* and the *system()* function to execute the string echo "*pid*">/mp1/status in the command line.

**Step 6:** The next step should be to create a **Linux Kernel Timer** that wakes up every 5 seconds. Timers in the kernel are single shot (i.e not periodic). Expiration times for Timers in Linux are expressed in **jiffies** and they refer to an absolute time since boot. Jiffy is a unit of time that expresses the number of clock ticks of the system timer in Linux. The conversion between seconds and jiffies is system dependent and can be done using the constant **Hz**. The global variable jiffies can be used to retrieve the current time elapsed since boot expressed in jiffies.

**Step 7:** Next you will need to implement the **work function**. At the timer expiration, the timer handler must use the workqueue API to schedule the work function to be executed as soon as possible. To test your code you can use *printk()* to print to the console every time the work function is executed by the workqueue worker thread. You can see these messages by using the command *dmesg* in the command line. **Also please note that the workqueue API was updated for kernel 2.6.20 and newer, therefore some documentation about workqueues on the internet might be outdated.**

**Step 8:** Now, you will need to implement the **updates to the CPU Times** for the processes in the Linked List. We have provided in the file *mp1_given.h* a helper function *int get_cpu_use(int pid, unsigned long* cpu_value)* to simplify this part. This function returns 0 if the value was successfully obtained and returned through the parameter *cpu_value*, otherwise it returns -1. As part of the update process, you will need to use locks to protect the Linked List and any other shared variables accessed by the three contexts (kernel, process, interrupt context). The advantage of using a two half approach is that in most cases the locking will be placed in the **work function** and not in the timer interrupt.

**Step 9:** Finally you should check for **memory leaks** and make sure that everything is **properly deallocated** before we exit the module. Please keep in mind that need to stop any asynchronous entity running (e.g timers, thread, workqueues) before deallocating memory structures. At this time, kernel module coding is finished. Now you should be able to **implement the factorial test application** and have some additional testing of your code.

## 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 identifying possible situations where they should be used is important in kernel programming.

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. Hand-In

All MPs will be collected using compass2g. You need to submit a single **group_ID_MP1.zip** (replace ID with your group number) file through compass2g. **Only one of the group members need to do the submission.** You need to provide all your source code (*.c and *.h) of your kernel module and the source of factorial application you implemented. Also, you should submit a *Makefile* that compiles both your kernel module and the user space application. Finally, you must write a document named Group_ID_MP1.pdf/Group_ID_MP1.doc (replace ID with your group number) briefly describing your implementation and design decisions. **Please clearly list all your group member names and netIDs in the beginning of the document.** Additionally, it needs to contain the following items in the end of the documents:
(1): Details of how to run your program
(2): Screenshot of the output when running the command “cat /proc/mp1/status”. At least two of your user program instances need to be running when capturing the screenshot. To run two user program instances concurrently, do “./userapp & ./userapp &” in your terminal (assume your user program is named userapp).

**Important**: You need to create a folder called CS423_TA in your VM home folder (To go to your VM home folder, do the command “cd ~” in your VM terminal, and then do “mkdir CS423_TA”.) Inside the CS423_TA folder, create another folder called “MP1”, and put all your source files, Makefile, and Group_ID_MP1.pdf/Group_ID_MP1.doc inside that folder.

MP1 will be graded based on the correctness of your code, and the clearness of the documents. **No late submissions are accepted!**

**9.** References

[1] The Kernel Newbie Corner: Kernel Debugging Using proc "Sequence" Files
https://www.linux.com/learn/linux-training/37985-the-kernel-newbie-corner-kernel-debugging-using-proc-qsequenceq-files-part-1
http://www.linux.com/learn/linux-career-center/39972-kernel-debugging-with-proc-qsequenceq-files-part-2-of-3
http://www.linux.com/learn/linux-career-center/44184-the-kernel-newbie-corner-kernel-debugging-with-proc-qsequenceq-files-part-3
[2] The Linux Kernel Module Programming Guide (a little bit outdated, but still useful), 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, (a little bit outdated, but still useful)
http://www.ibm.com/developerworks/linux/library/l-proc/index.html
[6] Kernel APIs Part2: Deferrable functions, kernel tasklets, and work queues
http://www.ibm.com/developerworks/linux/library/l-tasklets/index.html
[7] Love Robert, Linux Kernel Development, Chapters 6, 8-11, 17-18, Addison-Wesley Professional, Third Edition