CPTS 360: Lab Assignment 4

Introduction to Linux Kernel Programming

Notes:

- This is an individual lab.
- You must complete the lab on a Linux environment (either a dedicated machine or VM). Using WSL (Windows Subsystem for Linux) on Windows is NOT recommended.
- Your submission will be tested on a Linux environment. You will NOT receive any points if your submitted program does not work on a Linux system.
- Start EARLY. The lab may take more time than you anticipated.
- Read the entire document carefully before you start working on the lab.
- The code and other answers you submit MUST be entirely your own work, and you are bound by the WSU Academic Integrity Policy (`https://www.communitystandards.wsu.edu/policies-and-reporting/academic-integrity-policy/`).
- You MAY consult with other students about the conceptualization of the tasks and the meaning of the questions, but you MUST NOT look at any part of someone else's solution or collaborate with anyone.
- You may consult published references or search online archives, provided that you appropriately cite them in your reports/programs.
- The use of artificial intelligence (AI)–generated texts/code-snippets must be CLEARLY DISCLOSED, including the prompt used to generate the results and the corresponding outputs the tool(s) provide.

Good Luck!

1 Introduction

In this lab, you will learn the basics of Linux Kernel Programming. In particular,

1. You will learn to create a Linux *Kernel Module*.
2. You will use *Timers* in the Linux Kernel to schedule work.
3. You will use *Workqueues* to defer work.
4. You will use the basics of the *Linked Lists* interface in the Linux Kernel to temporarily store data in the kernel.
5. You will learn the basic of *Locking* in the Linux Kernel.
6. You will interface applications in userspace with your Kernel Module through the *Proc Filesystem*.

2 Overview

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 userspace 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.

A key difference is that, in kernel programming, *preemption is not always available*, which 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 userspace. For example, in kernelspace we are responsible for ensuring that interrupt handlers are as efficient as possible using the CPU for very little time.

Another important issue is the *lack of userspace libraries*. C/C++ standard library and other libraries reside in userspace and cannot be accessed in the kernel. This limits what we can do and how we implement it.

Note that that Linux Kernel *lacks floating-point support*, i.e., 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. The links and tutorials in Section 10 may be useful to implement this lab.

Development Guides

We recommend you complete this lab on a Linux Virtual Machine (VM), instead of a standalone Linux computer. During the completion of this lab, errors in your kernel modules could potentially lead to "bricking" VM, rendering it unusable. If this happens, you will need to restore your VM from the scratch. However, this will cost you precious hours of development time! These problems can be avoided by taking the following precautions:

- **VM Snapshots:** Keep a copy (snapshot) of your working VM with all necessary development libraries installed.
- **Code Versioning:** If your VM does become unusable, you need to use a fresh VM (perhaps reload from your saved VM snapshot). It is strongly recommended that you prevent loss of work by committing to your course git often. Note: we will only use your last submission before the deadline for grading.
- **Monitor Logs:** During the course of this lab, we will be generating a lot of log messages, which eventually find their way to the `/var/log` directory on our disk. Keep an eye on the `/var` partition using the `df -h` command and remove large log files if it starts getting too full.

3 Lab Tasks

In this lab, you will build a kernel module that measures the *userspace CPU time* of processes registered within the kernel module and a simple test case application that requests this service. In a real scenario

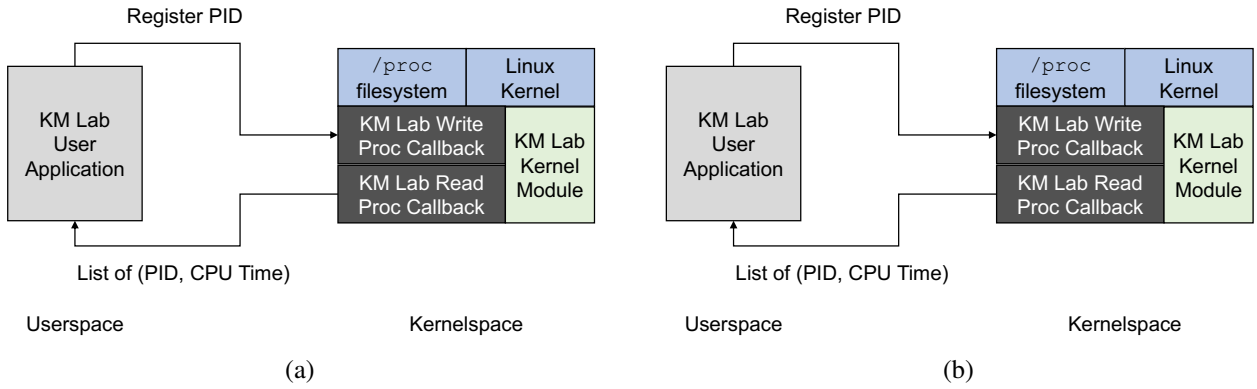

Figure 1: `/proc` filesystem interface between the test application (userspace) and the kernel module (called KM Lab module, left figure) and Architecture overview of the kernel programming lab (right figure).

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 userspace 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/kmlab`). Inside this directory your kernel module must create a file entry (e.g., `/proc/kmlab/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 userspace 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
```

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*.

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 lab, 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 1 shows the architecture of the kernel module you should implement, including the timer interrupt and the workqueue.

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 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` filesystem entry containing the list of all the registered applications and its corresponding CPU times. We provide a skeleton userspace C implementation (`userapp.c`).

4 Implementation Challenges

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

- During the registration process you will need to access data from userspace. Kernel and applications both run in two separate memory spaces, so de-referencing pointers containing data from userspace 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 kernelspace into userspace using the function `copy_to_user()`. Common cases where this might appear are in `\proc` filesystem callbacks and system calls.
- Another important challenge is the lack of libraries, instead the kernel provides similar versions of commonly used functions found in libraries. For example, `malloc()` is replaced with `kmalloc()`, `printf()` is replaced by `printk()` and `pr_X()` family of functions. Some other handy functions implemented in the kernel are `sprintf()` and `sscanf()`.
- 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* (e.g., `spin_lock_irqsave()`) and prevent race conditions wherever they appear.

Due to all these challenges, we recommend that **you test your code often and build in small increments**. Besides, we recommend you read kernel programming lecture modules and test all source files discussed in the class.

5 Implementation Overview

This section will briefly guide you through the implementation. Figure 2 shows the workflow and components of the lab. In userspace, you can see the test application.

Step 1: The best way to start is by implementing an empty `(Hello, World!)` Linux Kernel Module. You can find the starter code in `kmlab.c` of your GitHub submission repository. Make sure to edit the `MODULE_AUTHOR` line first.

Step 2: After this you should implement the `/proc` filesystem entries (i.e., `/proc/kmlab/` and `/proc/kmlab/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`. You should be able to see the `/proc` filesystem entries you created using `lsmod`. Now remove the module using `rmmod` and check that the entries are properly removed.

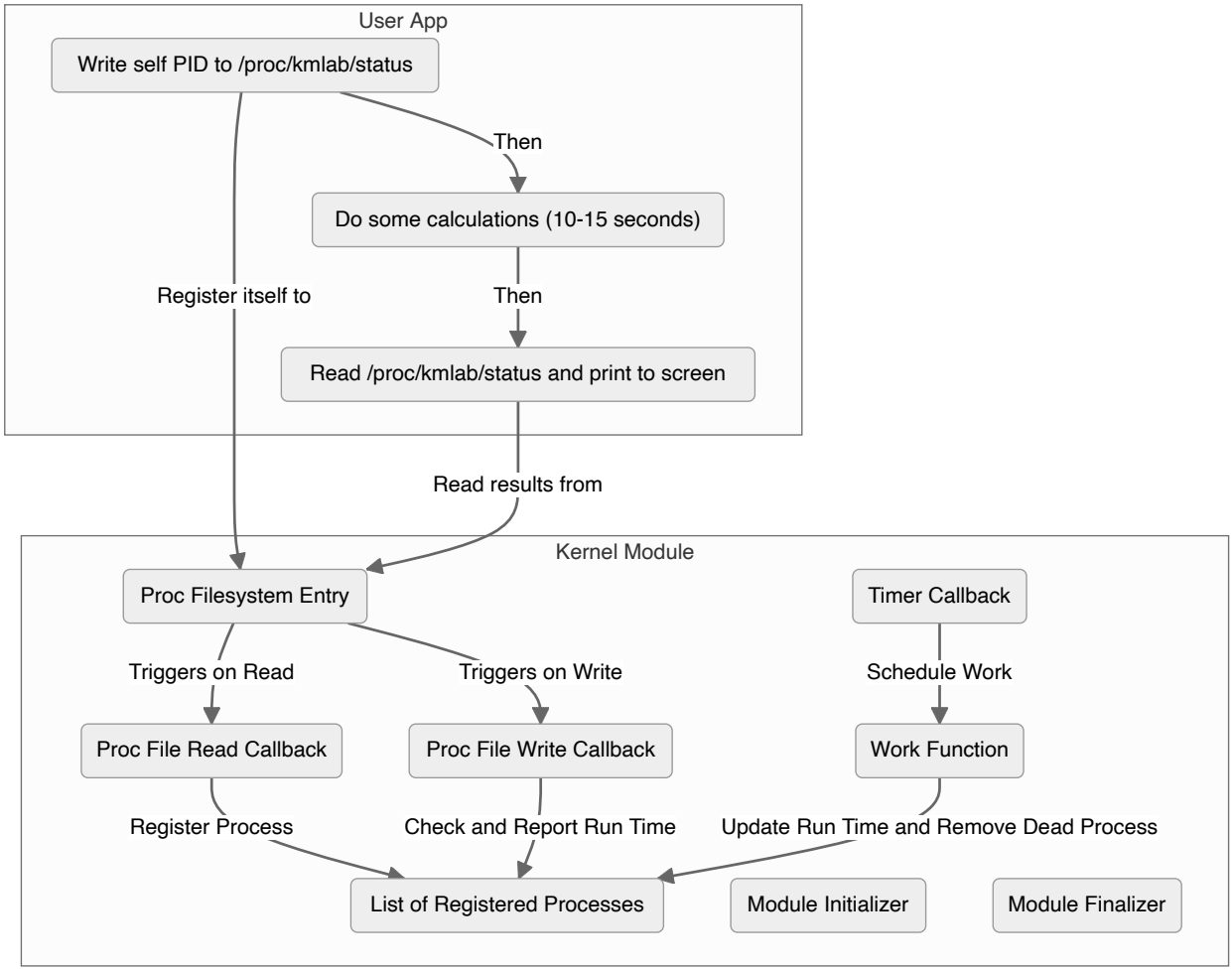

Figure 2: Workflow of the lab implementation.

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 userspace application should be able to register itself by simply writing the PID to the `/proc` filesystem entry you created (e.g.,`/proc/mp1/status`). The callback functions will read and write data from and to userspace 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 userspace 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> > /proc/kmlab/status` in a privileged shell.

Step 6: The next step should be to create a 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. **Note:** Workqueue APIs were updated in newer kernels, 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 a helper function (in `kmlab_given.h`):

```
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. If a registered process terminates, `get_cpu_use()` will return -1. In this case, the registered process should be removed from the linked list.

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

6 Compile and Test Your Code

We provide a `Makefile` to compile your kernel module.

To test your kernel module, you can try loading, unloading, and running it. The following commands may be helpful:

```
# inserting kernel module
insmod kmlab.ko

# removing kernel module
rmmod kmlab.ko

# registering PID 1 to the module
echo "1" > /proc/kmlab/status

# listing current processes and user times
cat /proc/kmlab/status

# print the kernel debug/printed messages
dmesg
```

We provide a sample shell script (`kmlab_test.sh`) that will install and test your module for two userspace processes.

7 Note on Code Quality

We recommend to read about the Linux Kernel's requirements for code quality here: `https://www.kernel.org/doc/html/v5.15/process/4.Coding.html`

For the lab, we use a *relaxed version* of the Kernel Code Quality Guideline for grading. For example, we require:

- Your code should not trigger compiler warnings.
- Properly protect critical resources with locks.
- Abstract the code appropriately and use functions to split the code.
- Use meaningful variable and function names.
- Write comments for non-trivial codes.

Here are some advice:

- 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 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.

8 Deliverable

- Complete the implementation of the user and kernel codes.

  [90 Points]

  - The point split for your implementation is as follows (see below for details):
    - **80 points** for correctness and
    - **10 points** for good code organization, indentation, and proper comments.

- A PDF document (no more than 2 pages, 11 points Times font, 1-inch margin) containing a summary of how you implemented each of the your kernel modules and **screenshots** of your output from `/proc/kmlab/status` (i.e., PID and corresponding runtime, **for both one and two process cases**)

  [10 Points]

  - Your PDF filename should be `lab4_lastname.pdf`, where `lastname` is your surname. Include your **full name** and **WSU ID** inside the report. We will **deduct 5 points** if the filename/contents does not follow this convention.

Correctness/Organization Criteria

- Can we insert your module?
- Does your user application function correctly?
- Does `/proc` read work correctly?
- Does `/proc` write work correctly?
- Does the interrupt handler work correctly (including removing finished processes)?
- Does your module correctly support multiple processes?
- Is your critical region lock correctly implemented?
- Does your module correctly free all memory?
- Can we remove your module?
- Your code compiles and runs correctly and does not use any floating point arithmetic.
- Properly written code (well commented, readable, and follows software engineering principles)

9 Submission Guidelines

Commit and push your work on GitHub Classroom Lab 4 repository. **Make sure you can successfully push commits on GitHub Classroom** ***last minute excuses will not be considered***). Get the Git commit id of your work by running the command: `git log -1 --format=oneline`. Paste the commit ID (the hexadecimal string) to the corresponding lab assignment section of Canvas. Check the course website for additional details.

Note: On Canvas, you can submit any commit ID (not necessarily the last one) from your GitHub Classroom Git history. You can also submit as many times as you want. However, we will grade the last commit ID submitted to Canvas before the deadline.

Your work on GitHub Classroom will not be considered for grading unless you confirm your commit ID on Canvas. Hence, **You will NOT receive any points if you fail to submit your commit ID on Canvas by the due date.**

10 Useful Links

- The Linux kernel module programming guide

  `https://sysprog21.github.io/lkmpg/`

- Linux Kernel Procfs guide (outdated, but still useful)

  `https://www.cs.cmu.edu/afs/grand.central.org/archive/twiki/pub/Main/SumitKumar/procfs-guide.pdf`

- Kernel linked lists (outdated, but still useful)

  `https://rootfriend.tistory.com/entry/Linux-Kernel-Linked-List-Explained`

- Timers and lists in the 2.6 kernel (outdated, but still useful)

  `https://developer.ibm.com/tutorials/l-timers-list/`

- Deferrable functions, kernel tasklets, and work queues (outdated, but still useful)

  `https://developer.ibm.com/tutorials/l-tasklets/`