What You Need

**Required Items**
- Completed MP0.
- Able to read, write, and debug program codes written in C language.

**Recommended Items**
- Have a handy code editor.
  - If you need some recommendations: VSCode, Neovim, or GNU Emacs
- Use Linux Kernel Documentation to help you understand concepts.
- Use Elixir Cross Referencer to help you go through codes.
  - [https://elixir.bootlin.com/linux/v5.15.127/source](https://elixir.bootlin.com/linux/v5.15.127/source)
Get Your Starter Code

Accept the Assignment on GitHub Classroom First.

- Go to this link: https://classroom.github.com/a/P4KJTn7f
- Login your GitHub account and find your Email.
- Accept the assignment.
- The starter code will be available in the repo created.
About Kernel Programming

• Lack of Isolation
  • Unhandled exception in a user program: The program dead
  • Unhandled exception in the kernel: The system dead

• Preemption is not Always Available
  • Infinite loop and dead locks are fatal
  • Make sure you use loops and locks carefully

• Lack of User Library
  • You will deal with a new set of functions (e.g. kmalloc, printk, snprintf)

• No Floating Points
  • You will destroy user program's calculation results
The Task

- A kernel module that measures the User Space CPU Time (User Time) of processes.
  - It allows multiple processes to register themselves and monitor their CPU usage concurrently.

- A user program that does some work and then checks its User Time.
  - It communicates with your kernel module to register itself and read User Time info.

- The kernel module and user program communicates via a Proc Filesystem Entry.

- A README file to briefly introduce the tasks you have done.
Component Overview

User App
- Write self PID to /proc/mp1/status
- Then
- Do some calculations (10-15 seconds)
- Register itself to
- Then
- Read /proc/mp1/status and print to screen
- Read results from

Kernel Module
- Proc Filesystem Entry
- Triggers on Read
- Proc File Read Callback
- Register Process
- Check and Report Run Time
- List of Registered Processes
- Update Run Time and Remove Dead Process

- Triggers on Write
- Proc File Write Callback
- Schedule Work
- Work Function (Bottom-Half)

- Timer Callback (Top-Half)
- Module Initializer
- Module Finalizer
The Kernel Module

The kernel module should be your main focus. It contains three parts:

- A Basic Runtime to track user program lists and do init/uninit jobs.
- A Proc Filesystem Entry to handle read and write requests for user programs.
- A Periodic Work to update User Time for programs.
The Kernel Module – Basic Runtime

• An initializer that allocates memory, lock, list, etc. when loading the module.
• An finalizer that deallocates the resources you allocated when unloading the module.
• A Linked List to store the User Times of registered processes.
The Kernel Module – Basic Runtime

- The **initializer** will be automatically called when inserting your module into a Linux kernel.
  - The entry point is provided in the **starter code**.

- The **finalizer** will be automatically called when unloading your module.
  - The entry point is provided in the **starter code** as well.

- To store the User Times of registered processes, you should use the **Linked List**.
  - The length of list is unknown during compile time.
  - Items may be removed from the middle of the list. (You may want to remove dead processes from the list)
  - You can check `include/linux/list.h` for Linux APIs on Linked List operations.
  - You can check references of the APIs in Elixir Cross Referencer to see their real-world use cases for better understanding.
The Kernel Module – Periodic Work

- Set a **Timer** in kernel to update the User Time of processes periodically (once per 5 seconds).
- The Timer will invoke a **Callback** when it is due.
- The Callback should use **Workqueue** to enqueue a Worker to do the real job.
- The **Worker** will be automatically called on a kernel thread when it is leaving the queue.
The Kernel Module – Periodic Work

• Why so complex?
  • The registered process list may be very long.
  • It may also need to wait on locks.
  • It is better not blocking the Timers for too long as this may affect other Timers in the system.

• Where to look at:
  • Timer API is defined in `include/linux/timer.h`
  • Workqueue API is defined in `include/linux/workqueue.h`
  • A good use case is `samples/ftrace/sample-trace-array.c#L24-L44,L79-L80`
  • Challenge: Linux Timer only fires once, how to make it fire multiple times in a fixed interval?
  • Answer: In the Timer Callback, modify the timer itself to fire again after another 5 seconds.
The Kernel Module – Periodic Work

• What to do in the Timer Callback?
  • Reset the Timer so that it can fire again after another 5 seconds
  • Enqueue a Worker onto the Workqueue

• What to do in the Workqueue Worker?
  • Lock the process list using Mutex to prevent race conditions with the Proc File handlers
  • Iterate through the registered process list
  • Check if each process is still alive and their up-to-date User Timer
  • Update the process entry to record the newest User Time if the process is alive
  • Remove the process from the list if it is dead
The Kernel Module – Proc Filesystem Entry

- Allow the user program to communicate with your module and get results. (File perm: 0666)
- Locates at /proc/mp1/status. Create the folder /proc/mp1 first, then the file /proc/mp1/status.
- Read: Report the User Time of all registered processes.
- Write: Register a new process using the PID of the process.
The functions to create Proc Filesystem folders and files are in `include/linux/proc_fs.h`.

- See `fs/lockd/procfs.c#L70-L92` for a simple real-world use case on creating/destroying Proc Filesystem Entries
- See `fs/jfs/jfs_debug.c#L20-L52` for a simple real-world use case on handling read/write for Proc Filesystem Entries

You will need to deal with “user pointers”, i.e. pointers that are unsafe to deference in kernel space.

- Kernel marks this type of pointers in this format: `void __user *ptr`
- You need to copy them to/from kernel space to access them safely.
- Use functions such as `copy_from_user()` or `copy_to_user()` before accessing them to eliminate security warnings.

You will need to parse and format strings to/from integers

- Use functions such as `snprintf()` (print to a buffer) or `kstrtoint()` (parse string to int)
The Kernel Module – Proc Filesystem Entry

- Example for Write

```bash
echo "1" > /proc/mp1/status # register PID 1
```

- Example for Read

```bash
# read all registered PIDs and User Times
cat /proc/mp1/status
1: 82902
1728: 3317982
1743: 3421024
```
The Kernel Module – Others

- Use Mutex lock to prevent race conditions between Proc File requests and periodic updates
  - Defined in `include/linux/mutex.h`

- Use Slab allocator to allocate memories
  - Defined in `include/linux/slab.h`

- Don’t worry on checking the liveness and User Time of processes
  - A function will be given to you as a part of the starter code
The User Program

- Get its own PID using getpid()
- Register itself to your kernel module via writing the PID to /proc/mp1/status
- Do 10-15 sec calculation (provided as a part of starter code)
- Read the User Time output from /proc/mp1/status, print to console, and exit
Write a README file

- Edit the README.md in your GitHub starter code repo
- Briefly describe how you design and implement each parts of the kernel module
  - E.g. which system API used in what part, how parts interact with each other, anything special with your implementation
- If your code failed to run correctly on the test machine, this will help you get partial grade
- Don’t need to be very detailed
- No word limit
Submission

• Push all your works into your GitHub repo (the repo containing your starter code)
• Grading will be based on your last commit pushed before the deadline
• TAs will compile and run your code on a MP0 VM to see if it works

• Deadline: Sep 26th at 11:59 PM CT
Recap and Q&A