MP1Walkthrough

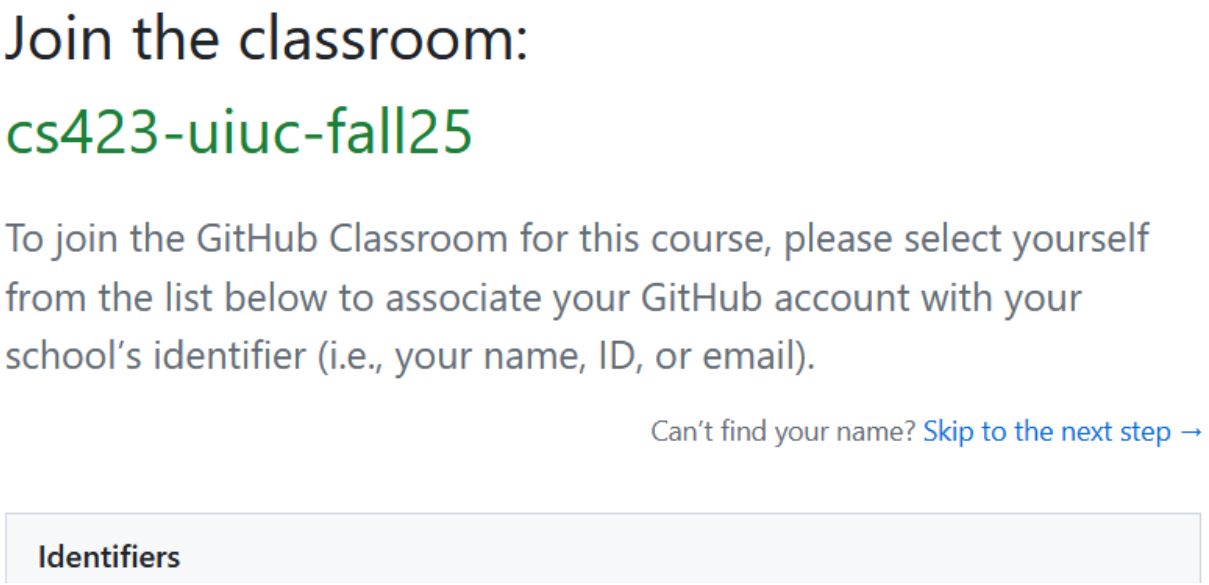
Cathy Cai
9/16

Credits: Siyuan Chai, Jinghao Jia

Get Starter Code

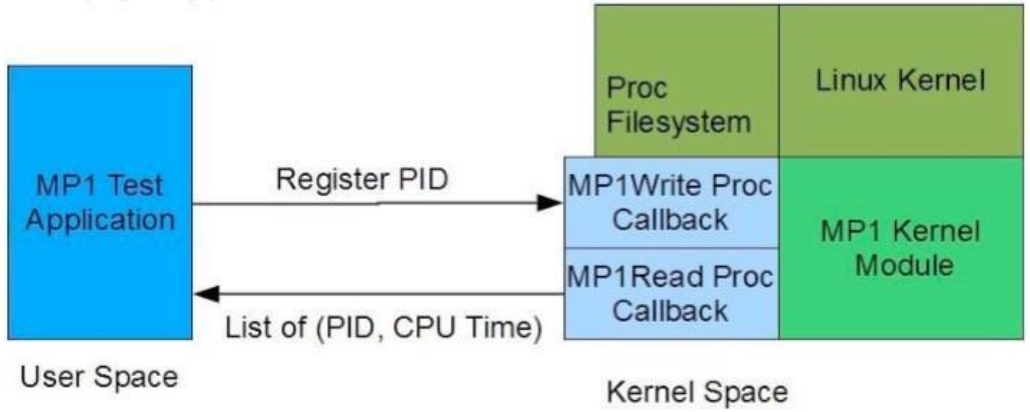
- https://classroom.github.com/a/HXioa_mx
- Find your name and click (don’t click on other’s name!)
- Due Oct. 2nd at 11:59 PM CT

- P.S. Don’t forget to submit your MP0!

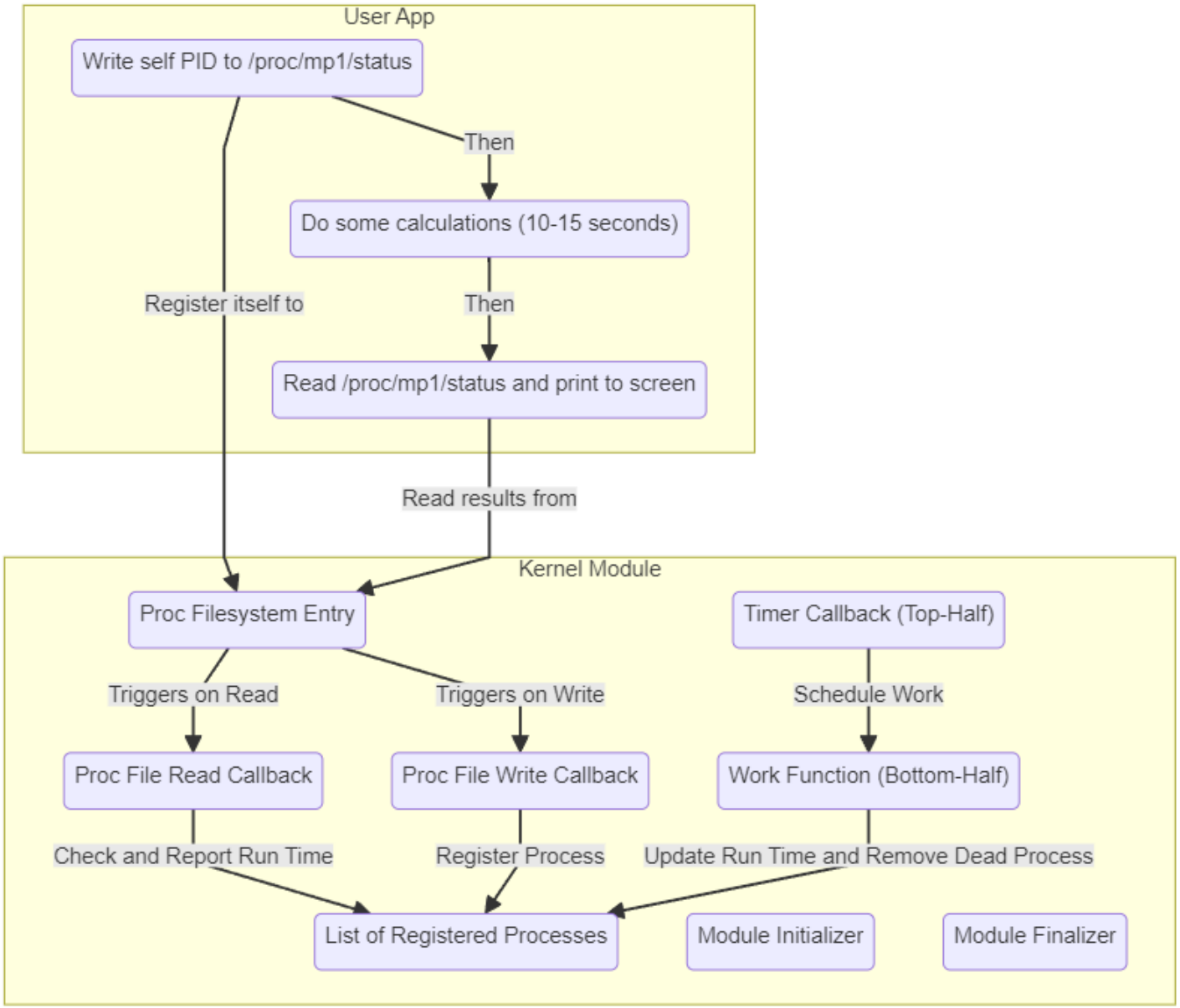


Problem Description

- Write a kernel module that measures the userspace CPU
- Time of processes registered within the kernel module
- Register processes using PID through the Proc Filesystem
- Kernel module updates the userspace CPU time of each registered process every 5s
- Print the userspace CPU time of each registered process

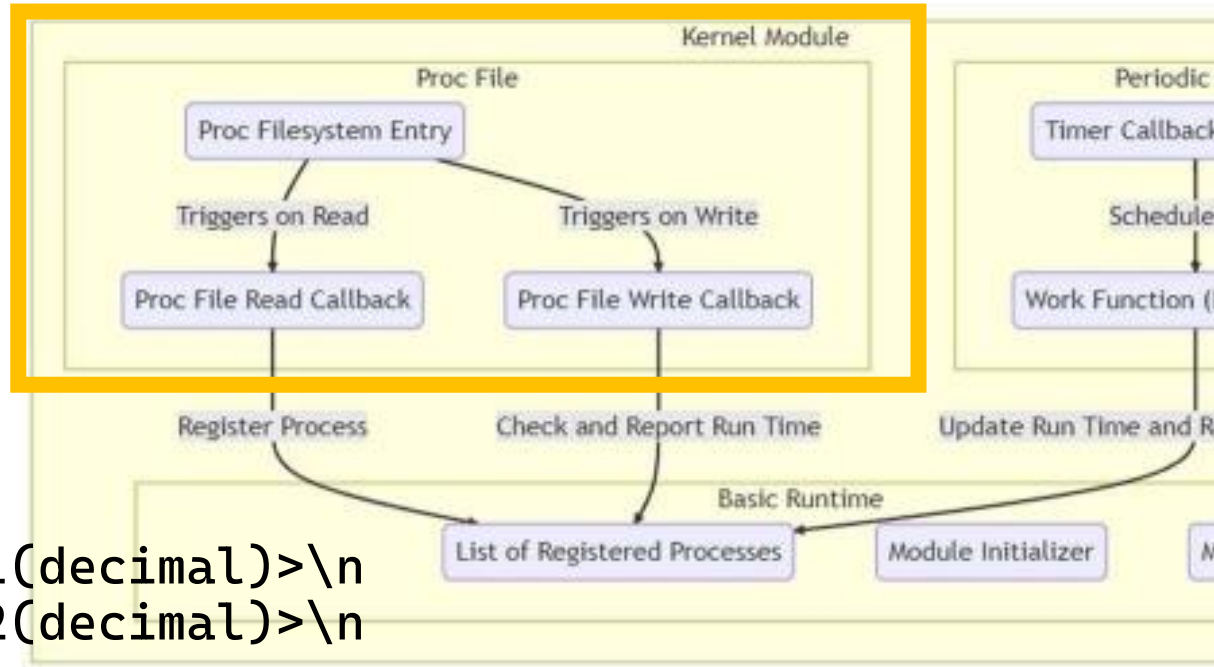


Overview


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
Timer Callback (Top-Half)
Triggers on Read
Triggers on Write
Schedule Work
Proc File Read Callback
Proc File Write Callback
Work Function (Bottom-Half)
Check and Report Run Time
Register Process
Update Run Time and Remove Dead Process
List of Registered Processes
Module Initializer
Module Finalizer

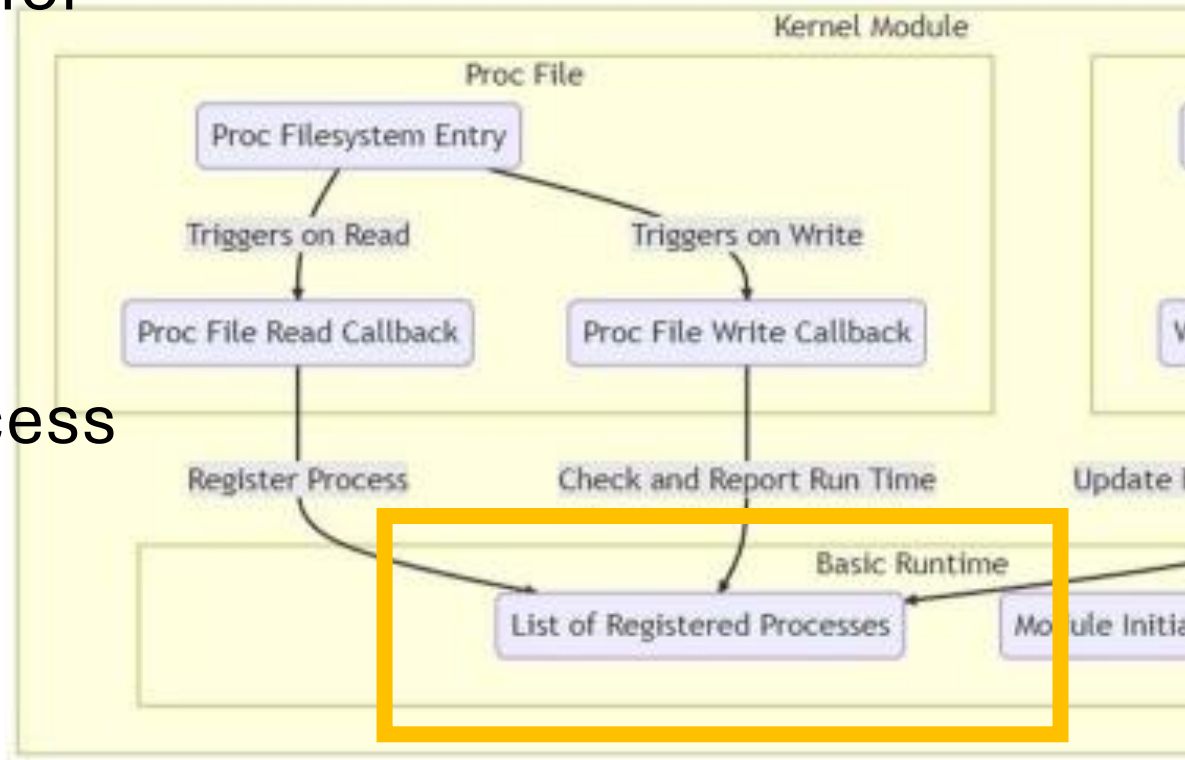
Proc Filesystem Entry

- Not regular files, does not store data in binary format
- Can be read/write as regular files
- Create an entry (e.g. `/proc/mp1/status`) in the proc filesystem
  - `proc_mkdir()`
  - `proc_create()`
- Register a process:
  - `echo “pid” > /proc/mp1/status`
  - Use `fprintf()`, etc.
- Get userspace CPU time:
  - `cat /proc/mp1/status`
  - Use `fgets()`, etc.
  - Should print in the following format:
    - `<PID1>:[space]<CPU time of PID1(decimal)>\n`
    - `<PID2>:[space]<CPU time of PID2(decimal)>\n`

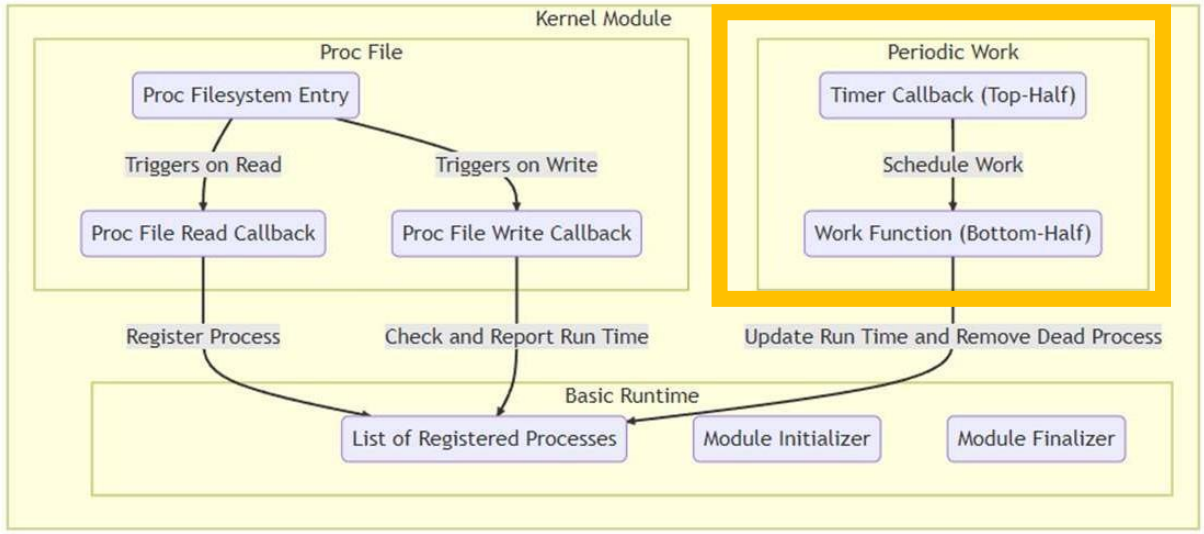


Store States

- Implement read and write callback for the proc entry
  - `proc_read()`
  - `proc_write()`
- Use kernel linked list to store the information of every registered process
  - APIs in `<linux/list.h>`
- Need to consider concurrency for linked list operations
  - E.g. using a lock



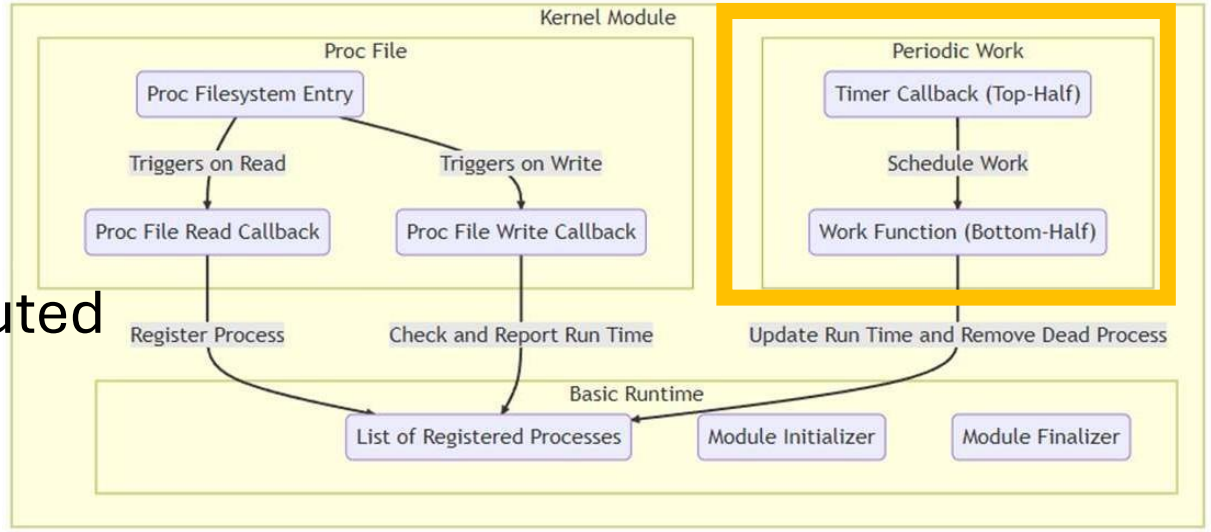
Periodic work: timer



- Use a kernel timer to perform a task after a preset timeout
  - APIs in `<linux/timer.h>`
- Setup timer
  - `timer_setup(timer, callback, flags)`
- Setup timeout
  - Timeout is represented in jiffy in kernel. Jiffy can be converted between regular time units (s, ms, etc.)
  - `mod_timer(timer, expires)`
- Challenge: timer only fire once

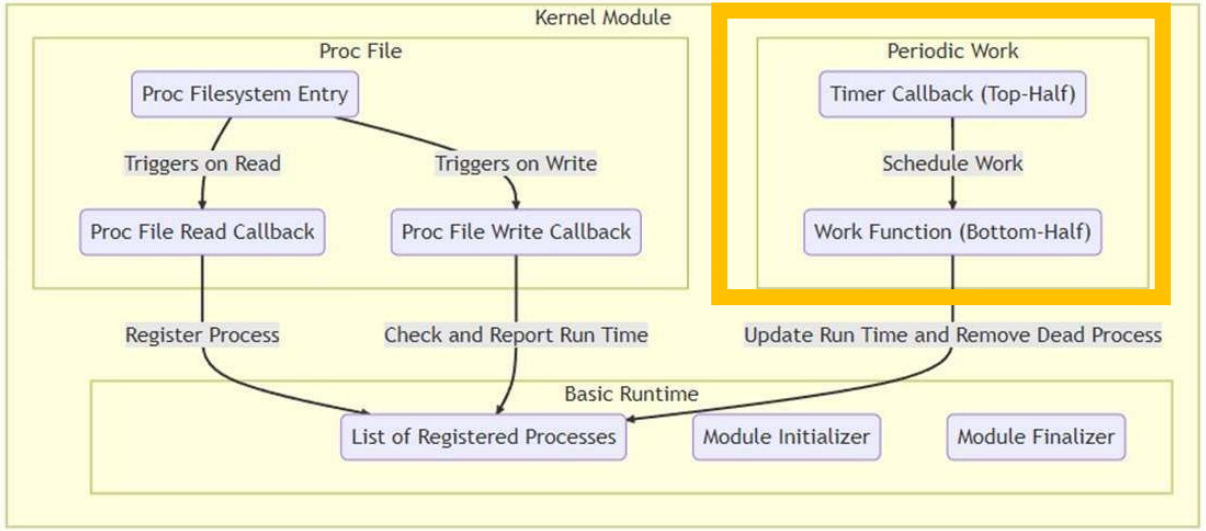
Periodic work: two-halves

- Not put all work in timer handler
  - Why?
  - Registered list can be long. Better not to not block other timers
- Use a two-halves approach
- Use kernel work queue
  - allow kernel functions to be executed by special kernel threads
  - APIs in `<linux/workqueue.h>`



Work Queue

- Schedule a function to be run in a work queue
  - `queue_work(work_queue, work)`
  - callback only calls `queue_work()` (Top-Half)
  - work is where we are going to do the actual updates (Bottom-Half)



Passing Data

- Passing data between kernelspace and userspace
  - E.g. `ssize_t proc_read(struct file *file, char user *buf, size_t size, loff_t *loff)`
  - buf here is a userspace address and can't be dereferenced directly in kernel space
  - Use `copy_from_user()/…` to copy to a kernel buffer
  - Same for `copy_to_user()/…`
- Free/deallocate any memory/objects before exiting the kernelmodule
  - Dynamic allocated memory using `kmalloc()` must be freed using kfree()
  - Objects such as timer/work_queue must be destroyed
  - Proc FS entry must be removed

Debug and Submission

- Debug
  - Use `printk()` to print to the kernel log
    - View the kernel log using dmesg (e.g. `dmesg | less`)
    - Works on any platform
  - Sufficient for MP1 (from my experience)
- Use gdb
  - A bit tricky to load the symbol table for kernel module. You can ask Cathy/Peizhe how to do that
- Submission
  - Push your code to your GitHub repo before ddl