Project 1

- Due Oct 13 by 11:59pm
- Points 20

CMPSC 473 Project 1: Thread Scheduler

(due Oct 13 before 11:59PM via Github - NO EXTENSIONS WILL BE GIVEN!)

This page will continue to be updated. Please check back periodically for updates and announcements.

Please direct all your project-related questions/clarifications to the TAs, during their office hours or via Canvas Discussions/Email.

Academic Integrity Statement:

The University defines academic integrity as the pursuit of scholarly activity in an open, honest and responsible manner. All students should act with personal integrity, respect other students' dignity, rights and property, and help create and maintain an environment in which all can succeed through the fruits of their efforts (refer to Senate Policy 49-20. Dishonesty of any kind will not be tolerated in this course. Dishonesty includes, but is not limited to, cheating, plagiarizing, fabricating information or citations, facilitating acts of academic dishonesty by others, having unauthorized possession of examinations, submitting work of another person or work previously used, or seeking help from others, without informing the instructor, or tampering with the academic work of other students. Students who are found to be dishonest will receive academic sanctions and will be reported to the University's Office of Student Conduct for possible further disciplinary sanctions (refer to Senate Policy G-9). The Academic Integrity policy for the Department of Computer Science and Engineering for programming projects can be explicitly found at ***EECS Academic Integrity Policy (https://www.eecs.psu.edu/students/resources/EECS-CSE-Academic-Integrity.aspx)*** *. Please go through it carefully and do ask the instructor for any clarifications if something is not clear.*

There should be no exchange/dissemination of code, or collaborative working on the projects across teams! Any sharing of code (both the giver and the taker), or using code from sites on the Internet, use of AI Tools, and Tutoring services/sites, or consulting others (except the instructor and TAs) to do your projects, will be considered as academic dishonesty, leading to serious consequences. Ensure that your code is NOT visible to anyone outside your team. For any violation, you will be given a score of 0 for that project apart from being reported to the concerned authorities! Any second violation will have more serious consequences including course grade penalties.

Note that all team members (when working in a team) will be held equally accountable and responsible for the integrity of the entire project, even if you have worked on only a part.

Summary

In this project, you will implement a thread scheduler with three CPU thread scheduling policies and semaphore support.

Deadline

October 13, 2025, 11:59 PM

Goals

- Learn to use pthreads library to implement multithreaded programs
- Know the importance of synchronization (locks, condition variables, semaphores), and how to properly use them
- Understand and implement different CPU scheduling policies

Overview

You will be given a test program that creates many threads. Each created thread will do one or more of the following (i.e., call these functions):

- CPU burst `[cpu_me()]`
- IO operation `[IO_me()]`
- P or wait on a semaphore `[P()]`
- V or signal a semaphore `[V()]`
- End thread `[end_me()]`

The resulting thread's execution profile will be saved as a Gantt chart in the output directory. You need to implement a scheduler with the operations (APIs) above and return from the functions at the correct time. If implemented correctly, the output will be the correct Gantt chart corresponding to the input file.

Hardware Model

Your scheduler should manage these threads to run on a machine with: Single CPU core system (preemption may be possible based on scheduling policy) + Single IO device (no preemption, i.e. once an I/O operation is initiated, it should complete before another can be initiated). The CPU and IO device operate independently (i.e, while a thread has the CPU, the IO device can be performing some *other* thread's operation). However, two threads cannot be running on the CPU at the same time.
See *Scheduling Policy Explanation* below for more details.

Input File Format

An input to the program is a text file which has the following format:

```
arrival_time tid [operation] ... [operation] E
arrival_time tid [operation] ... [operation] E
arrival_time tid [operation] ... [operation] E
...
```

where [operation]: C/I/P/V and E are explained below:

Operation	Explanation
`Cx`	CPU burst of x ticks
`Ix`	IO of x ticks
`Px`	P() on semaphore id x
`Vx`	V() on semaphore id x
`E`	End thread

The arrival time can be a float value. However, the scheduling decisions and P/V operations are done in integral values/boundaries. P/V operations take no time (i.e, P/V are instantaneous, but P may block - as per the semaphore semantics). Go through the following examples for clarity.

Sample Input and Output

① Basic Case

```
1.3 0 C5 I7 C3 I8 E
```

can be interpreted as:
Thread0 arrives at time 1.3. It then does CPU burst of 5 ticks, IO for 7 ticks, CPU burst of 3 ticks, IO for 8 ticks, and then terminates.
Following is the Gantt chart (output of the program) of the example above:

```
  2~  3: T0, CPU
  3~  4: T0, CPU
  4~  5: T0, CPU
  5~  6: T0, CPU
  6~  7: T0, CPU
    ~ 14: T0, Return from IO
 14~ 15: T0, CPU
 15~ 16: T0, CPU
 16~ 17: T0, CPU
    ~ 25: T0, Return from IO
```

Since the scheduling decision is done at integer granularity, thread 0 arrives at time 1.3, but gets scheduled at time 2. After finishing CPU burst of 5 ticks at time 7, IO gets scheduled for 7 ticks at time 7. Therefore, IO finishes at time 14. C3 gets called next, and so forth. For this input, the result is the same for all scheduling options (FCFS, SRTF, MLFQ).

② P/V Example

```
0.0 0 C5 P1 C2 E
1.1 1 C2 V1 C3 E
```

can be interpreted as:
Thread0 arrives at time 0.0. Initial CPU burst of 5 ticks, does P() on semaphore-id 1, then has subsequent CPU burst of 2 ticks, and then finishes its task.
Thread1 arrives at time 1.1. Initial CPU burst of 2 ticks, does V() on semaphore-id 1, then has subsequent CPU burst of 3 ticks, and then finishes its task.
Following is the output Gantt chart of the example above:

FCFS	SRTF

```
  0~  1: T0, CPU
  1~  2: T0, CPU
  2~  3: T0, CPU
  3~  4: T0, CPU
  4~  5: T0, CPU
  5~  6: T1, CPU
  6~  7: T1, CPU
   ~  7: T1, Return from V1
   ~  7: T0, Return from P1
  7~  8: T0, CPU
  8~  9: T0, CPU
  9~ 10: T1, CPU
 10~ 11: T1, CPU
 11~ 12: T1, CPU
```
	```
	  0~  1: T0, CPU
	  1~  2: T0, CPU
	  2~  3: T1, CPU
	  3~  4: T1, CPU
	   ~  4: T1, Return from V1
	  4~  5: T0, CPU
	  5~  6: T0, CPU
	  6~  7: T0, CPU
	   ~  7: T0, Return from P1
	  7~  8: T0, CPU
	  8~  9: T0, CPU
	  9~ 10: T1, CPU
	 10~ 11: T1, CPU
	 11~ 12: T1, CPU
	```

- FCFS: since Thread0 arrives at 0, it is immediately scheduled. Thread1 will call C2 with time 1.1, but since FCFS, you need to let it wait until thread0 finishes its first CPU burst. At time 5, Thread0 calls next operation P1, which will block (check semaphore definition). When thread1 finishes its C2 at time 7, it will immediately call V1, which will immediately unblock thread0's P1. Therefore, thread0 will call C2 and thread1 will call C3 simultaneously at time 7 (since P/V operations take no time - instantaneous). So, C2 and C3 both arrive at time 7... which is first-come? In this case, the *tie-breaker* is to choose the least tid. (See *Additional Implementation Requirements* below.)
- SRTF: thread1 arrives at time 1.1. Therefore, the scheduling decision at time 2 (integer granularity) is thread1's C2 vs. the remaining thread0's CPU burst of 3 ticks. Which one has the shortest remaining time at time 2? Thread1's C2! Therefore, thread1 gets the CPU for 1 tick. Which one has the shortest remaining time at time 3? It is thread0's 3 ticks vs. thread1's 1 tick. Thread1 gets the CPU again. At time 4, thread1 calls the next operation immediately, which is V1. V1 will increment the semaphore 1 and return immediately at time 4. (Again, P/V operations take up no time.) After this, thread1 will call the next operation C3 at time 4. Which one has the shortest remaining time now? There is thread0's initial CPU burst leftover of 3 ticks, and thread1's new C3 operation. The shortest remaining time is the same! Therefore, as a tie-breaker, we choose the least tid, which is thread0. At time 5, it is 2 ticks vs. 3 ticks (new C3), so we choose thread0. At time 6, the remaining time is 1 tick vs. 3 ticks, so we choose thread0

again. Thread0's next operation is P1, which gets called at time 7 and returns immediately at time 7 because thread1 already incremented the semaphore. At time 7, the remaining time is thread0's new 2 ticks vs. 3 ticks, so we choose thread0 again.

③ MLFQ Example

```
50.3 0 C8 E
52.1 1 C8 E
```

Thread0 arrives at time 50.3. It does C8 (CPU burst of 8 ticks).
Thread1 arrives at time 52.1. It does C8 too.

FCFS	MLFQ
`51~ 52: T0, CPU`	`51~ 52: T0, CPU`
`52~ 53: T0, CPU`	`52~ 53: T0, CPU`
`53~ 54: T0, CPU`	`53~ 54: T0, CPU`
`54~ 55: T0, CPU`	`54~ 55: T0, CPU`
`55~ 56: T0, CPU`	`55~ 56: T0, CPU`
`56~ 57: T0, CPU`	`56~ 57: T1, CPU`
`57~ 58: T0, CPU`	`57~ 58: T1, CPU`
`58~ 59: T0, CPU`	`58~ 59: T1, CPU`
`59~ 60: T1, CPU`	`59~ 60: T1, CPU`
`60~ 61: T1, CPU`	`60~ 61: T1, CPU`
`61~ 62: T1, CPU`	`61~ 62: T0, CPU`
`62~ 63: T1, CPU`	`62~ 63: T0, CPU`
`63~ 64: T1, CPU`	`63~ 64: T0, CPU`
`64~ 65: T1, CPU`	`64~ 65: T1, CPU`
`65~ 66: T1, CPU`	`65~ 66: T1, CPU`
`66~ 67: T1, CPU`	`66~ 67: T1, CPU`

- FCFS: Schedule starts at time 51 (integer granularity). Rest is straightforward.
- MLFQ: At arrival of thread0, it enters MLFQ's queue 0 (which has the highest priority). The time quantum is 5 ticks for queue 0. Meanwhile, thread1 enters the MLFQ at time 52.1. Note that within a MLFQ queue, you use FCFS policy to choose a thread. Therefore, when making a scheduling decision at time 53 (integer granularity), per FCFS, you schedule thread0. At time 56, the thread0's time quantum is used up. Therefore, thread1 gets scheduled.

④ More Complex Example

Input	Output of FCFS
30.9 0 C2 V5 C3 E	`31~ 32: T1, CPU`
30.2 1 C2 V5 C3 E	`32~ 33: T1, CPU`
34.9 2 C3 E	`  ~ 33: T1, Return from V5`
	`33~ 34: T0, CPU`
	`34~ 35: T0, CPU`
	`  ~ 35: T0, Return from V5`
	`35~ 36: T1, CPU`
	`36~ 37: T1, CPU`
	`37~ 38: T1, CPU`
	`38~ 39: T2, CPU`
	`39~ 40: T2, CPU`
	`40~ 41: T2, CPU`

	```
	 41~ 42: T0, CPU
	 42~ 43: T0, CPU
	 43~ 44: T0, CPU
	```

Note that [arrival_time] may or may not be in order. We make a scheduling decision at time 31. In this case, since thread1 arrived at 30.2, it gets scheduled first. There is a difference between *arrival time* and schedule decision time. Schedule decisions are made at integer granularity, but FCFS accounts the actual first-come time. At time 33, thread1's initial C2 gets finished, and it immediately calls V1, which increases the semaphore 5 and immediately returns. Then it will call the subsequent C3 also at time 33. Comparing the thread0's C2 arrival time 30.9 and thread1's C3 arrival time 33, FCFS chooses thread0. At time 35, thread1's C3 arrival time is 33, thread0's C3 arrival time is 35, and thread2's arrival time is 34.9. Therefore, FCFS chooses thread1.

⑤ More Corner Cases

Input	Output of FCFS

50 0 C3 E
5.5 1 C3 E
	```
	  6~  7: T1, CPU
	  7~  8: T1, CPU
	  8~  9: T1, CPU
	 50~ 51: T0, CPU
	 51~ 52: T0, CPU
	 52~ 53: T0, CPU
	```

7.1 0 V1 V2 V3 V4 V5 P2 C2 E	
	```
	   ~  8: T0, Return from V1
	   ~  8: T0, Return from V2
	   ~  8: T0, Return from V3
	   ~  8: T0, Return from V4
	   ~  8: T0, Return from V5
	   ~  8: T0, Return from P2
	  8~  9: T0, CPU
	  9~ 10: T0, CPU
	```

50.3 0 V5 V5 V5 P5 P5 E	
	```
	   ~ 51: T0, Return from V5
	   ~ 51: T0, Return from V5
	   ~ 51: T0, Return from V5
	   ~ 51: T0, Return from P5
	   ~ 51: T0, Return from P5
	```

50.0 0 C3 P1 I2 E
0.0 1 C3 P1 I5 E
70 2 V1 C1 V1 E
	```
	  0~  1: T1, CPU
	  1~  2: T1, CPU
	  2~  3: T1, CPU
	 50~ 51: T0, CPU
	 51~ 52: T0, CPU
	 52~ 53: T0, CPU
	   ~ 70: T2, Return from V1
	   ~ 70: T0, Return from P1
	 70~ 71: T2, CPU
	   ~ 71: T2, Return from V1
	   ~ 71: T1, Return from P1
	   ~ 72: T0, Return from IO
	   ~ 77: T1, Return from IO
	```

2000 0 I3 E
1500 1 I3 E
	`~1503: T1, Return from IO`
	`~2003: T0, Return from IO`

543.1 0 E
123.5 1 C3 E
	`124~125: T1, CPU`
	`125~126: T1, CPU`
	`126~127: T1, CPU`

If there are no operations, you should not return (print) anything (notice the gap between time 9~50 in the first case). Also, check again how semaphores work. Note that there may be no 'C' operation at all. Additional test cases are provided in the GitHub repo. You should also make your own to test your program.

APIs to Implement

In `api.h`, the following functions are declared. You should implement them according to the descriptions below.

// Scheduler type

enum sch_type {

SCH_FCFS = 0, // first come first served

SCH_SRTF = 1, // shortest remaining time first

SCH_MLFQ = 2, // multi-level feedback queue

};

// This is invoked once in the beginning. You can initialize anything you may want here.

`void init_scheduler(enum sch_type scheduler_type, int thread_count);`

// This is invoked once at the end. Free all memory and data structures.

`void finish_scheduler();`

The below functions may be called multiple times, depending on the input.

Common variables

float current_time: the time when the call is made by some thread (note, this can have float values)

int tid: the thread id

`int cpu_me(float current_time, int tid, int remaining_time);`

A thread calls this function for CPU burst, with the remaining_time in this burst. You must return from this function if the calling thread should run on the CPU for at least the next tick. The thread will then subsequently come back and call this function for the remaining time in the burst (i.e. 1 tick less than the previous call).

Return value: the time when the thread has completed 1 tick of execution on the CPU. Note that this may not necessarily be 1 more than the time that it was called with - depends on whether some other thread will get to execute on the CPU in-between.

/* For example, when only one thread0 arrived at time 1.1 with 'C3 E',

/* the following calls to your scheduler will be made:

cpu_me(1.1, 0, 3); --> would be scheduled (by you) at time 2, return value should be 3 (since it used 1 CPU burst for time 2~3)

cpu_me(3, 0, 2); --> calls with current_time 3, return value should be 4

cpu_me(4, 0, 1); --> calls with current_time 4, return value should be 5

cpu_me(5, 0, 0); --> calls with current_time 5, return value ignored

*/

`int io_me(float current_time, int tid, int duration);`

A thread calls this function at the start of the IO operation, with the duration specified in the input file.

This function must return only when the whole IO duration is finished. Note that it is possible that the IO device is busy when this call is made, in which case the request should get serviced only after all prior IO requests are serviced.

Return value: the time IO completely finishes and returns

/* For example, when only one thread0 arrived at time 0.3 with 'I5 C2 E',

/* the following calls to your scheduler will be made:

io_me(0.3, 0, 5); --> would be scheduled (by you) at time 1, return value should be 6 (since it did IO for time 1~6)

cpu_me(6, 0, 2); --> at time 6, the CPU burst will start (called)

cpu_me(7, 0, 1); --> calls with current_time 7, return value should be 8

cpu_me(8, 0, 0); --> calls with current_time 8, return value ignored

*/

`int P(float current_time, int tid, int sem_id);`

P implements the WAIT operation for the semaphore identified by sem_id

This may be a blocking call, according to the semaphore definition

Return value: the time it returns

`int V(float current_time, int tid, int sem_id);`

V implements the SIGNAL operation for the semaphore identified by sem_id

If more than 1 thread is blocked on the same sem_id, the thread with the lowest tid will return from P()

Return value: the time it returns

`void end_me(int tid);`

Notify your scheduler that the calling thread tid is terminating.

You can implement additional functions (helper functions) and have additional variables and structures in your scheduler, but note that the main.c will call only the functions listed in *api.h*. You may NOT modify the *main.c* file.

Scheduling Policies

① For CPU, you will implement three scheduling policies, as stated above as 'scheduler_type'. Since our hardware is a uni-processor environment, only one thread can run on the CPU at a time.

I. FCFS (First Come First Serve): The first thread that arrives at the scheduler gets the CPU for the entire remaining_time. Note that you must still return from cpu_me() every tick (as stated above) even though the currently running thread will continue to run for the next tick. By definition FCFS is non-preemptive.
II. SRTF (Shortest Remaining Time First): The thread with the shortest remaining_time gets the CPU. The shortest remaining_time may differ each time tick, and this policy is preemptive.
III. MLFQ (Multi-level Feedback Queue): Preemption-based with 5 levels. The time quantum for the 5 levels is 5, 10, 15, 20, and 25 time ticks respectively, from highest priority to lowest priority. When a thread completes its time quantum on a particular level, it will be moved to the tail of the next lower level. Your scheduler will select threads from a lower level queue only when there are no threads in the higher level queues. A thread initially entering the queue (i.e., the start of a Cx burst in input file) will enter the highest priority queue, which will preempt threads at the lower levels. Within a level, use FCFS to select a thread.

② There is a single IO device in the system, which uses FCFS policy with no preemption. (If a thread is scheduled to do IO, it does IO for the full duration.) The device can service only 1 request at a time.

Semaphores

Semaphores are initialized to 0. (You can initialize this in init_scheduler() function). Semaphores are *counting* semaphores. The sem_id will be from 0 to MAX_NUM_SEM-1 (defined in *api.h*). You may implement semaphores using pthread locks and pthread condition variables. For more information, check the above API section.

Additional Implementation Requirements/Restrictions

- You CANNOT use any time/timer functionality provided by the operating system (e.g., C time(), sleep(), jiffies, pthread_cond_timedwait(), timespec, clock_gettime() etc.).
- No busy wait or spin locks. Instead, use pthread locks and condition variables. For example, you **CANNOT** do this:

```
// Thread called cpu_me() having arrival time=30.1,
// but your scheduler's current global time is 5
while(arrival_time > global_time)
```

```
    ;       // Busy Wait until global_time gets incremented...
// Now time is right...?!
```

- Use pthreads library. `pthread_cond_init/destroy/signal/wait`, `pthread_mutex_init/destroy/lock/unlock`, `pthread_create/join`, etc. Go to **https://www.kernel.org/doc/man-pages/ (https://www.kernel.org/doc/man-pages/)** and type the functions in the search box to see the manual.
- You CANNOT use other outside libraries, other than pthreads.
- Multithread safe. Accesses to the same data (structure) may lead to deadlocks or errors. Make sure you are using locks/condition variables properly. If not, sometimes your scheduler may work, sometimes it may not.
- After you implement the proper scheduling policy, semaphore policy, and other requirements stated, and there is *still* a conflict of choosing a thread to schedule, always choose the thread with the lowest tid. (This means that your scheduler is deterministic. For a given input, the scheduler will output the same Gantt chart every time.)
  For example:

If your scheduling policy is FCFS, and thread0 and thread1 both arrive at time 3 and request CPU burst, choose thread0 to have the CPU burst for one tick.
If your scheduling policy is SRTF, and thread5 and thread7 both has shortest remaining_time of 2, choose thread5 to have the CPU burst for one tick.
If three threads call IO *at the same time*, IO starts for the one with the lowest tid. Other threads are blocked in the IO queue.
If thread5 and thread8 is waiting on the same semaphore (same sem_id), and one V() call wakes the semaphore sem_id, thread 5 should return from P(). (thread8 stays blocked)

- Your program must cleanly terminate without errors within a timeout. If your program executes longer than this timeout, it is assumed to be deadlocked (and therefore failed the given test case).

Teamwork and Submission Instructions

- You can work in teams (at most 2 members per team), or individually, for the project. You are free to choose your partner. But if either of you choose to drop out of your team for any reason at any time, each of you will be individually responsible for implementing and demonstrating the entire project and writing the project report by the designated deadline. (Please notify the TAs if this happens.)
- Even though you will work in pairs, each of you should be fully familiar with the entire code and be prepared to answer a range of questions related to the entire project. It will be a good idea to work together at least during the initial design of the code. You should also ensure that there is a fair division of work between the members. Remember that you both are owning up to the entire project that is being submitted, and you both are owning up to what works, what doesn't and the integrity of the entire project (not just your piece)!

- One team member should create a team using the following invitation. The other team member should join the team (if you would like to work in teams).
  Github invitation link: https://classroom.github.com/a/FjjZWRw3 (https://classroom.github.com/a/FjjZWRw3)
- All the code-bases will be tested on the W135 Linux Lab servers located in Westgate. Regardless of where/how you develop/debug your code (laptop linux, Linux VMs, etc.), you need to ensure you test and run them in W135 servers for being graded!
- Additional announcements would be sent out in the future to get your team information and the GitHub repository. Stay tuned!

Dos and Don'ts

- Fully (re)reading/understanding this document, the template code, understanding textbooks/slides, and reading the appropriate manuals are part of this project.
- Try to go through *main.c* and understand what it does. You can see that `./tester <scheduling-policy> <input-filename>` (e.g., `./tester 0 sample_input/input_0`)is how you run the program.
- Determine how you will manage time in your scheduler. For example, a global_time variable? How should you use/increment that variable? Note that you may NOT use the time functionality provided by C or operating system. Check *Additional Implementation Requirements* section.
- Think about "how to block a thread until it is its turn". Much of the scheduler is about figuring out whether the thread can run, and if not, blocking it until it can.
- Implement queue structures and the functions (e.g., enqueue(), delete(), find(), etc.) that may be used in your scheduler. You should determine what should be saved in the queue and when/how should the entries get removed from the queue.
- Think about common characteristics. the FCFS policy, within the same level of MLFQ, and IO: all use FCFS. Instead of implementing 3 times, is it possible to re-use and share some code?
- Think about concurrency. What will happen to your variables/structures/functions when multiple threads access them at the same time? Are they thread-safe? Where should you put locks?
- Implement cpu_me() and FCFS first.
- Create your own test cases (and corner cases) and see if your scheduler works properly. Test your program with inputs that have many threads and ensure there are no deadlocks.
- You may add additional `*.c` files (e.g., queue.c or linkedlist.c, etc.) inside the `./libscheduler` directory and add the `*.o` file in `./libscheduler/Makefile` (which is NOT the top-level Makefile) accordingly. Adding additional .c files inside ./libscheduler is optional.
- You CANNOT modify other files such as `main.c` and top-level `Makefile`.
- You CANNOT use other outside libraries, other than pthreads.

Debugging Hints using GDB

Deadlocks and other errors may happen. Compile your scheduler with *make clean*, then *make debug*, and try to debug using gdb. When deadlock happens inside the gdb environment, interrupt the process (Ctrl+C) and return back to gdb. In addition to gdb commands in project 0, in gdb command line:

See where the threads are currently executing (or stuck)	(gdb) info threads (or i threads for short)
Switch to a different thread x (id shown in info threads)	(gdb) thread x
Backtrace the selected thread (print the stack)	(gdb) backtrace (or bt for short)
Inspect frame number x (x is shown in backtrace)	(gdb) frame x
Execute current line of code (after e.g., breakpoint), and step into the function	(gdb) step (or s for short)
Execute current line of code (after e.g., breakpoint), but step over the function	(gdb) next (or n for short)
Execute until return of this function	(gdb) finish

This is not an exhaustive list. You should go over the gdb manual and find out about other useful commands.

Submission Guidelines

- Ensure you commit your code often using the `git commit` and the `git push` commands (similar to how you had done P0).
- You should write a brief report (`report.pdf` of around 2 pages) explaining the design of your scheduler, any implementation choices/restrictions, any issues not implemented, and distribution of the workload within the team.
- The project (your final scheduler implementation, including the `report.pdf`) must be pushed to the main branch by 11:59PM on the designated day. The last commit and push that you do prior to the deadline will be taken as your final submission.
- You may make multiple branches, but make sure you push your code into the main branch before the posted deadline. We will only clone from the main branch.
- Double-check and see if your submission is properly pushed by accessing your repository in GitHub.com
- NO EXTENSIONS WILL BE ENTERTAINED. ENSURE YOU PUSH YOUR CODE AND REPORT TO THE GITHUB REPOS BEFORE THE DEADLINE.

Project Grading

- We will clone the last push you made before the deadline, on the W135 server. We will type *make* to compile your program. We will NOT fix any compile errors.

- We have additional surprise test cases (they may be long and large, max: 2^12 threads, 2^12 operations per thread) and use them as input to your scheduler.
- After your scheduler produces all outputs (FCFS, SRTF, MLFQ), we will then sort* the output (using the *sort* command in command line). Then we will use the *diff* command to see any differences from our output. (i.e., Your output Gantt chart must match our results.)
- We will run the same test cases multiple times. This is to check if there are any deadlocks/race-conditions in the code. There is a timeout of 3 seconds for each execution for the sample test cases. If your scheduler runs longer than that, we will assume it has a deadlock, and the corresponding test case will fail. (The timeout will increase accordingly for longer/larger surprise test cases.)
- Read the *report.pdf* and see how your team attempted/designed/implemented the scheduler.
- Check if *main.c* was changed or not. Do not change *main.c* and top-level *Makefile* file. Also check if you have used any restricted functions such as time. (Check *Additional Implementation Requirements* section)
- Have a short meeting with your team to demonstrate and ask about the project, the code, the project report, and the functionality. For example, we may ask you to find, explain, and/or modify portions of code in the meeting.

Grading will depend on several factors: how much of the project was implemented? does it work for the pre-given inputs? does it work for the additional inputs? Under what situations does it not work? how was the design of the project requirements? Did team members understand all the intricacies of their implementation and how well are they able to demo and/or answer questions? How was the work partitioned amongst the team members?

* Why sort? The outputs below have equivalent meaning. If IO returning at time 33 is correct, both are OK. There is no fixed print order between functions returning *at the same time*.

```
31~ 32: T1, CPU
32~ 33: T1, CPU
  ~ 33: T2, Return from IO
33~ 34: T0, CPU
```

```
31~ 32: T1, CPU
  ~ 33: T2, Return from IO
32~ 33: T1, CPU
33~ 34: T0, CPU
```

Questions

For questions, please use the 'Discussions' tab of Canvas or use office hours. Please direct all project-related questions to the TAs. Please do NOT copy/paste code and post it in Discussions.