TCES 420, Fall 2016 -- Project #3

Note: I can't get the figure to appear properly in this description, so there is a PDF version of this description that includes the figure in the Files section of this course on Canvas. Please refer to the figure in the PDF version.

The objective of this project is to gain experience constructing, running, and debugging parallel, multi-threaded programs using pthreads as well as gain some experience with simple, centralized schedulers. You will need to implement this project as a user space application in the C-language.

For this project, you are to implement a simple centralized scheduler for a 16-core SMP machine in which a subset of thread acts as local worker threads for executing different phases of each job, similar to a thread-pool model.

A job in the case of this project is simply a sequence of two types of phases: CPU phases and I/O phases. You will define a data structure (or data structures) to keep track of the jobs as they are injected into the system. Each phase will have a time associated with it. The time indicates how long the phase lasts and how long the corresponding threads must “execute” it. Note that you will not have to execute anything during these phases; rather the time simply indicates how long each thread should delay before proceeding to the next phase.

For example, the below may be insufficient depending on your implementation, but one way may be to use a structure named job capable of tracking phases, durations, and progress to completion that you will use to track the state of each job in the system:

```
struct job {

int job_id;

int nr_phases;

        int current_phase;

// Phase types: 1 = CPU phase; 2 = IO phase

int phasetype_and_duration[NR_PHASES][NR_PHASES];

int is_completed = 0;

};
```

You will need to instantiate multiple instances of jobs with different phase types and durations including different combinations of CPU and IO phases (IOW, you don’t have to use the above phase definitions; you may choose your own mechanisms and/or definitions). Each phase must have a time associated with it, such that the local CPU and I/O schedulers know how long each phase runs.

Operational Details

Figure 1 below provides a pictorial view of how your threaded job executions sequence should work.

You will implement three queues to handle scheduling job to either run on CPUs or perform I/O:

1) A Read/Run queue for jobs ready to run a CPU-bound phase,

2) An I/O Wait queue for jobs waiting to start an I/O phase

3) A Finished queue for jobs that have completed all job phases.

Jobs that are not running should be placed in one of the three queues depending on their state If a given job is runnable, meaning the job was just started OR the next part of the job is CPU-bound, the job will be put into the run queue. If a job is waiting on I/O, it should put into the I/O wait queue, so that an I/O thread can run it.

Out of the 16 threads in the system, you must have ***three*** distinct types of threads running as follows:

- ***Eight CPU threads*** – These are threads who have the exclusive role of "running" the CPU-bound phases of jobs as defined in the job profile. These threads must handle adding jobs to the run queue when they are runnable (At the completion of a CPU-phase, where the next phase is runnable). They must also handle adding jobs to the I/O queue when a CPU-phase completes and the next phase is I/O-bound. Finally, they must handle the case when a job completes and should be put on the finished queue. CPU threads that are not currently running a job should continually check the Run queue for newly injected and/or scheduled jobs. Moreover, before transitioning jobs back onto one of the three queues for the next phase (or completion), the CPU threads should update the job data structure with the completed phase.
- ***Four I/O handling thread***s – these are the I/O handling threads. These threads will be responsible for checking on the state of the I/O Wait queue for jobs that are waiting for I/O phases to be completed. When jobs are added to the I/O Wait queue, they will take them off the queue and "execute" the I/O phase by waiting for the appropriate phase time before either putting them back onto the Run, IO Waiting, or Finished queues depending on the next phase of the job. Just like the CPU threads, the I/O handling threads should also update the job data structure when a phase completes and putting the job onto the appropriate queue for the next phase.
- ***Four Job submission threads-*** These threads are solely responsible for creating jobs at a given frequency and submitting them to the ready queue. This should follow a similar approach to the process of creating new processes and threads we discussed in class. Each job submission thread should have a way to keep track of the processes each created so that it can monitor the *Finished* queue for completed jobs it created and free the memory associated with the job. The minimum job creation and injection rate for these threads should be every 2-3 seconds. In other words, each of the four job submission threads should inject new jobs into the system at least every 2-3 seconds. When they are not submitting jobs, they should be checking the *Finished* queue for completed jobs.

Atomicity requirement – Each phase of each job should be executed atomically and only once. In other words, because there are eight CPU threads and four I/O handling threads, you could have multiple threads attempting to en-queue or de-queue jobs onto the queues concurrently. Consequently, you must ensure consistency and atomicity of share data structures. For example, adding or removing elements from each queue may require locks, semaphores, or other constructs as we've discussed for mutual exclusion.

Output requirements - The threads in your application should print out a message when 1) jobs are started, 2) when jobs transition between queues, 3) when executing on CPU threads and I/O threads, 4) when completed (transitioned to the Finished queue), and 5) when each jobs is officially complete (taken off the Finished queue and freed by the creating thread). Please either redirect the output from the program to a file (if you're just using stdout) or write to a file.

Project Submission Details:

Please submit the following via the Canvas site for the class by the dead line :

- C-based, pthread source code for your program.
- Makefile to build your project
- Copy of the output from your program showing job creations, transitions, completions, etc.
- Single page, written summary of the challenges you encountered doing this project. For this write-up please describe how you approached handling mutual exclusion amongst the threads.