CS 343 Operating Systems, Winter 2025
Queueing/Scheduling Lab

Contents

1 Introduction

The purpose of this lab is for you to gain some experience with scheduling and understanding the differences between different schedulers, and how consequential these differences can be under conditions of high load. In addition, this lab will introduce you to discrete event simulation, which is a very widely used model for simulating systems in which it is possible to identify events and cause/effect relationships among them.

The framework of the lab creates the environment of a single CPU (i.e., single hardware thread) scheduling problem. In multiple CPU systems (which are by far the most common today), a common approach is to have a separate single CPU scheduler for each CPU, and then have these schedulers steal work from each other (or foist work on each other). This lab only implements non-real-time schedulers, which schedule jobs without known deadlines.

You may work in a group of up to three people in this lab. Each group will have its own Github repo for developing code. Clarifications and revisions will be posted to Piazza.

2 Setup

You can work on this lab on any modern Linux system, although we will test your work on the class server (Moore). First you should accept the assignment and clone your project repo into a private folder in your home directory. How to do this will be posted in Piazza. You probably want to mark the directory as private (`chmod 700 repo-directory-name`).

Entering the directory, you can build the entire project with `make`. That should result in one executable file: `queuesim`.

```
$ make
```

A default FIFO scheduler is provided in `fifo_sched.c`. Several scheduler workloads are provided in `workloads/`. To run the event simulator on the FIFO scheduler, you can run the following command:

```
$ ./queuesim fifo_sched workloads/example.txt
```

The command will run the simulation and capture the results. After it is done, it will print the statistics of what the jobs experienced given the scheduler in use. Different schedulers will produce different behaviors here, especially at high load.

You can also run the simulation in single step mode, where it will pause after each event and display the contents of various queues. This is particularly useful when debugging your schedulers.

```
$ ./queuesim fifo_sched workloads/example.txt singlestep
```

Looking in your repo directory, you'll see at least the following files:

- `Makefile`: The build setup. You will have to edit this as you add more schedulers.
- `debug.h`: The debug configuration. You may edit this disable some debugging.
- `queuesim.c`: The simulator's event loop and `main`. You should not edit this file.
- `fifo_sched.c`: A simple first-in-first-out scheduler. **Your schedulers will have a similar structure.**

- `support/`: C and H files to build the scheduler and simulation. You should not be editing these files, but will have to use functionality they provide.
- `support/context[ch]`: The context of the current simulation.
- `support/event.[ch]`: The simulator's event implementation.
- `support/eventqueue.[ch]`: The simulator's event queue implementation.
- `support/job.[ch]`: The implementation of a job within the simulator
- `support/jobqeueue.[ch]`: The implementation of a job queue (e.g. ready queue) within the simulator.
- `support/scheduler.[ch]`: The scheduler abstraction and code to keep track of installed schedulers. Essentially, this is the base class of the schedulers you will write.
- `support/list.h`: An intrusive doubly-linked list implementation used in various places. You are welcome to use it too. This is the cache-optimized, high-performance linked list implementation used throughout the Linux kernel and numerous other kernels, including the kernel you will use later in the class.

- `tools/`: Perl scripts for creating test input and visualizing results.
- `logs/`: Logged results from running `queuesim`.
- `workloads/`: Various workloads to test your schedulers on.

3 Scheduler Terminology

`queuesim` uses the following terminology with regard to the scheduling system it is simulating:

- Job: a job is self-contained chunk of work for the CPU that has a given Arrival Time and initial Size.[1] Jobs can be periodic (repeating with deadlines), sporadic (single run with a deadline), or aperiodic (single run without a deadline). **All jobs in this lab will be aperiodic.**[2]
- Job Arrival Time: the point in time when a job becomes known to the scheduler. For example, if a job has an arrival time of 1024 seconds, this means that it becomes available to the scheduler 1024 seconds after the scheduler is booted.
- Job Size: the amount of time on the CPU required to complete the job. For example, a job that has size 12 seconds means that it needs to execute on the CPU for 12 seconds before it is complete. Another term for this concept is the job service time.
- Job Completion Time: the point in time when a job has run on the CPU for its entire size, and thus is finished.

[1]The code also uses the term Task. A Task refers to a sequence of jobs and is only relevant to the real-time parts of the system, which are included in the extra credit. Specifically, a Periodic Task produces a sequence of Periodic Jobs that have Arrival Times with a fixed interval between them. Real-time jobs also have deadlines.

[2]Except in parts of the extra credit that involve real-time scheduling!

- Job Turnaround Time: the difference between a job's arrival time and its completion time.[3]
- Job Queuing Time: the difference between a job's turnaround time and its size.
- Job Slowdown: the ratio of a job's turnaround time to its size.

The event file given to the simulator typically includes a set of job arrivals (each having the time it occurs, the job size, and the job's priority). Using your scheduler, the simulator computes the completion time of each job, and thus its turnaround and queuing times, as well as its slowdowns.[4]

4 Discrete Event Simulation

`queuesim` is a discrete event simulator. What this means is that instead of simulating the passage of time directly, it jumps from event to event. For example, suppose the current time is 100 and a job of size 50 arrives. This is represented as a "job arrival" event (for a size 50 job) that occurs at time 100. If there is no job currently running, the scheduler might decide to immediately run the new job. To do so, it tells the simulator to post a "job completion" event for time 150. If there is no other event between now (time 100) and time 150, the simulator can immediately jump to time 150. When the simulator gets that "job completion" event, it records the turnaround time as being 50, and its slowdown as being 1.

Discrete event simulators are very powerful tools that are widely used in science and engineering. `queuesim` is implemented in the usual manner for a discrete event simulator. There is a priority queue, called the eventqueue, which stores the events in time order. The simulator main loop simply repeatedly pulls the earliest event from the eventqueue and passes it to a handler until there are no more events in the eventqueue. The handler for an event may insert one or more new events into the eventqueue. A handler can also update or delete events in the eventqueue. As in the example, the handler for a job arrival might post a new completion event for itself. As another example, the handler for a timer interrupt event (needed for preemptive schedulers) might delete the current job's completion event, put that job to sleep, switch to a different job, schedule that job's completion event, and then also schedule a new timer interrupt event.

The eventqueue will usually have multiple job arrivals within it at the start, loaded from a workload file (more below). Your scheduler will add to it, up to one job completion event at a time, which may be removed before it occurs if the job is preempted. Some schedulers may also add up to one timer interrupt event at a time which signifies timeslice expiration. *While your implementation is up to you, an implementation with multiple enqueued job completion or timer interrupt events (more than one of each) is likely incorrect.*

Relevant source files The files `support/event.h` and `support/eventqueue.h` are most relevant here. They show the types of events in the simulator and the functions that can be applied to the simulator's eventqueue. Within a scheduler, the `context` argument provided to each function allows accessing the eventqueue for the simulation.

Getting started with `queuesim` may be a bit confusing as your scheduler will itself manage queues of jobs, while the simulator core will manage a queue of events. Play around with the provided scheduler example and look through the files in `support/` to make sure you understand what's going on.

[3]The term for this in the performance analysis world (and the queueing handout) is the Job Response Time. We are using the common OS term (and the one the book uses) here to try to reduce confusion. Note, however, that the OS world also sometimes uses the term response time to mean the time from when a job is unblocked to when it begins to run again. We want you to see scheduling from both worlds' perspectives, hence we cannot totally avoid these differences in terminology.

[4]When real-time tasks and jobs are considered (such as in some of the extra credit), the simulator will also report on whether these jobs meet their deadlines or not.

Workload files Events are loaded into the simulation through workload files. `queuesim` loads the initial list of events from a workload input file. Several example workloads are provided in `workloads/`. The example workload in `workloads/example.txt` is a good example when creating your own workloads.

The following are the kinds of events that can be placed directly into the workload input file. You can also place comments into the file—blank lines or those starting with # are ignored. The first set of events involves the arrivals of different types of jobs and tasks:

```
arrival_time APERIODIC_JOB_ARRIVAL size static_priority

arrival_time PERIODIC_TASK_ARRIVAL period slice numiters

arrival_time SPORADIC_JOB_ARRIVAL size absolute_deadline
```

`APERIODIC_JOB_ARRIVAL` is simply the "job arrival" concept from before. Some schedulers observe job priorities in making their decisions. The `static_priority` is the baseline for an aperiodic job's priority. The other two kinds of arrivals are for real-time tasks and jobs. You can ignore them—they are only relevant to some of the extra credit.

The remaining events are for generating output:

```
arrival_time PRINT_STATS

arrival_time PRINT_JOB_QUEUES

arrival_time PRINT_EVENT_QUEUE

arrival_time PRINT_ALL

arrival_time DISPLAY_QUEUE_DEPTHS
```

The last of these pops up a gnuplot window with a graph showing the history of the depths of your jobqueues over time. The simulation will stall until you hit enter. Any of these output events can be scheduled to arrive at any time to display partial results.

In addition to the events that can appear in the file, your scheduler may add `JOB_DONE` and `TIMER` events to the eventqueue to represent the job completing or the job quantum expiring[5] respectively.

5 Scheduler Example

Within the discrete event simulation, schedulers handle job events by adding or removing events from the simulation as necessary. They also manage their own queue of outstanding jobs to be completed. Some schedulers run each job to completion while others may preempt the running job and switch to another.[6] In the latter case, fields within the job may need to be updated to track the remaining size of the job or its dynamic priority.

An example scheduler implementation is provided in `fifo_sched.c`. It implements a non-preemptive, first-in first-out scheduler. Jobs are completed in the order they arrive and are not preempted while running. This kind of scheduling is sometimes called "first come, first served."

[5]I.e., a timer interrupt.
[6]For example, to make an interactive job more responsive.

Relevant files Schedulers make frequent use of the fields and functions in `support/job.h` and the functions in `support/jobqueue.h`. They also use the context defined in `support/context.h` to access simulation parameters and structures.

Scheduler State Each scheduler function receives a handle to the scheduler's state as its first argument. The state is passed in generically as a `void*` pointer but can be cast back into the state struct for this particular scheduler.

The purpose of the scheduler state is to hold whatever state the scheduler needs about currently running operations. For example, the FIFO scheduler keeps track of whether it is currently running a job with the `busy` field. Your own schedulers may add whatever fields you want to the scheduler state for your own use. **Your scheduler implementations MUST NOT use global variables. Any information you want to save must instead be added to the scheduler state.**

Constructor Each scheduler must also create a constructor function that registers the scheduler with the simulation. In the FIFO scheduler example, this is called `fifo_sched_init()` and is located at the bottom of the file. Note that the name of this function must be different for each scheduler.

The marking `__attribute__((constructor))` is a signal to the compiler that this function constructs a key part of the program and should be run before `main()` is started.

Required Functions Each scheduler must implement four operations to support the simulation.

1. `initialize()` handles initialization of the scheduler (if any).
2. `aperiodic_job_arrival()` handles job arrival events.
3. `job_done()` handles job done events.
4. `timer_interrupt()` handles timer expiration events (if any).

These functions are supplied to the simulation through the `sim_sched_ops_t` struct as function pointers. The combination of such a structure of function pointers (the interface) and the state passed to each function is a common idiom for doing object-oriented programming within C. Interface + State = Object. Linux (and NK) use this idiom extensively.

Each of these functions will possibly modify some of: the scheduler state, the jobqueue, the eventqueue, or fields of the job itself. Note that within `support/job.h` the fields `remaining_size` and `dynamic_priority` may be modified within the job struct. There are helper functions that can modify them. No other fields in the job struct or any other simulation structs should be modified directly by the scheduler.

Function Pointers Throughout the codebase, we use *function pointers*. A function pointer is a pointer to code instead of to data. For example, consider this declaration:

```
int (*func_ptr)(double x, double y);
```

This declares and defines a variable named `func_ptr` that points to a function that has two arguments (`x` and `y`) and returns an `int`. We can assign to this pointer:

```
func_ptr = some_function_we_have;
```

And then we can call the function it points to:

```
func_ptr(42.0, 23.0);
```

Note that unlike a regular function call, the actual function that is called is determined at run-time. `func_ptr` is a *variable*, not a constant or a name.

5.1 Creating a new Scheduler

A recommendation is to start by copying `fifo_sched.c`, or another scheduler you have written, to use as a starting point. Then you can make modifications as necessary.

Make sure to rename some parts of the file. For example, if copying `fifo_sched.c` each use of the word "fifo" or "FIFO" in the source code should be modified to match the name of your new scheduler. Particularly, the constructor function must have a unique name for each of your schedulers. Also make sure that the string name passed into `sim_sched_register()` is unique for each scheduler and matches the project specification.

After you create your new scheduler file and rename necessary parts, you will need to add it to your `Makefile` for it to compile. Add new scheduler filenames to the `Makefile` as a part of the `SCHED_SOURCES` list. After you do this, running `make` should compile your new scheduler.

Different schedulers will have different requirements. Some, but not all, will require implementing the `timer_interrupt()` handler function. All will need to implement the other three handler functions though.

For some schedulers, it is helpful to sort the jobqueue by some parameter of the job. The function `sim_job_queue_enqueue_in_order()` in `support/jobqueue.h` will be useful for this. It takes in a function pointer to a helper function that you write which compares two jobs at a time and decides their sorting order. Be aware that the comparison function should return an `int` value: -1, 0 or 1, while some job parameters are `doubles`. Subtraction alone won't work right here!

Some schedulers will need to modify the `remaining_size` field of jobs to keep track of how much of the job has already completed. The `remaining_size` field is only for scheduler tracking purposes and may be safely modified by the scheduler. Likewise, the `dynamic_priority` field of jobs may be modified. Be careful not to modify any other fields of the job or else statistics on job completion may become inaccurate.

6 Testing and Debugging

We **STRONGLY** recommend you create your own example workloads to test for common cases and edge conditions for each scheduler. It is very important to work through some simple examples, where you only have a few jobs, by hand. Basically determine the right answer yourself, and then see if your simulator gives the same result. This is a form of "simulator validation", which builds trust in the simulator.

You should also calculate expected turnaround for the example provided in `workloads/example.txt` for each scheduler to help know if the scheduler is working or not. The expected value for turnaround is different on the example workload for each scheduler.

Remember that while turnaround values change with scheduler, all schedulers should complete a given schedule in the same amount of time. The completion time for all three jobs in the example workload should always be 360.0 regardless of scheduler. If the completion time of a workload changes for a scheduler, that is a bug in your scheduler.

If your simulation results do not match your math, one of the two is incorrect. One extremely valuable tool for checking your code is the ability to single step through events. At each point in the simulation you can check that the jobs in the jobqueue and events in the eventqueue match your expectations. To run the simulation in single step mode, add the argument “singlestep” to the end of the argument list. For example:

```
$ ./queuesim fifo_sched workloads/example.txt singlestep
```

For schedulers that use timers, you may want to change the timer quantum while single-stepping so things move along faster. You may change the value of the quantum during testing by setting the environment variable `QUEUESIM_QUANTUM` while running `queuesim`. For example you can set the quantum to ten seconds while also single-stepping with:

```
$ QUEUESIM_QUANTUM=10 ./queuesim rr_sched workloads/example.txt singlestep
```

7 Tasks

7.1 Task 0: First-in First-out Scheduler

Get it, build it, run it on the example input file. Run it again in single-stepping mode, like this:

```
  $ ./queuesim fifo_sched workloads/example.txt singlestep
```

In this mode, before each event is handled, you’ll see the state of the simulator and the simulated system. The simulator will pause on each event, waiting for you to hit return.

Out of the box, everything will be run through the `fifo_sched` scheduler. Make sure you understand how the event simulator is working, and how this simple default scheduler is implemented.

This is also a great time to run your code under gdb. While there is no bug, you can use gdb to observe what is happening dynamically, which is usually much more useful for understanding a codebase than trying to eyeball all the source code statically.

The `fifo_scheduler` scheduler is a non-preemptive scheduler applied to all tasks (real-time or not). When a job arrives, it is placed at the back of the job queue. When a job is complete, the scheduler selects the next job from the front of the job queue. The job will run to completion—this a non-preemptive scheduler. “FIFO” stands for “first-in, first-out”. Intuitively, this scheduler operates just like standing in line at a grocery store.

Submission There is no submission for this task.

7.2 Task 1: Non-preemptive Shortest Job First Scheduler

In this task, you will write a scheduler that is actually optimal under certain conditions. The idea of `sjf_sched` is that when you select the next job, you will choose the one which has the smallest initial size (shortest service time). If ties in size occur, the job with the earliest arrival time should run first.

This policy can be proven to minimize the average turnaround time seen by jobs. It does require that the size of the jobs be known when they arrive, which is a serious challenge, particularly in a commodity operating system. The policy can also lead to starvation. Consider what happens if a short job arrives, followed by a long job, and then a long chain of short jobs.

Be sure the modify your `Makefile` to compile your new scheduler.

Submission Properly implement the scheduler in a file "`sjf_sched.c`" in your repository that registers the scheduler "`sjf_sched`".

7.3 Task 2: Preemptive Shortest Remaining Processing Time Scheduler

Now you will implement your first preemptive scheduler, `srpt_sched`. In a preemptive scheduler, an event can happen that requires us to pause the currently running job (preempt it), and replace it with some other job. Eventually, we will resume running the preempted job. `srpt_sched` is the preemptive variant of `sjf_sched`. It also has a range of desirable properties and is used in a number of contexts.

The idea is that each job will have a remaining time associated with it—this is how much time it has left to run. When a new job arrives, its remaining time is set to the job size (service time). If the new job's remaining time is less than the remaining time of the current job, then the current job is preempted by the new job.

If ties in remaining size occur when scheduling, the job with the earliest arrival time should run first.

The simulator code for a preemptive scheduler is considerably more complex than for a nonpreemptive scheduler. In particular, you will need to be able to update or delete events, not just add them.

Submission Properly implement the scheduler in a file "`srpt_sched.c`" in your repository that registers the scheduler "`srpt_sched`".

7.4 Task 3: Preemptive Static Priority Scheduler

The next scheduler to implement is also preemptive but focuses on a different metric. `priority_sched` uses the static priority of jobs to determine the order they should be completed in. If a new job arrives with a higher `static_priority`, the current job should be preempted and the new job should immediately run.

This kind of scheduler is also known as a fixed priority scheduler.

Static priority values are specified in the workload file and are values in the range of 0–99 inclusive. Higher values have a higher priority, so a job with a priority of zero is the lowest possible priority.

In the case of a tie in priority, the first job to arrive should be scheduled first. If jobs match in priority and arrival time, run the job with the shortest remaining size. Jobs matching in priority, arrival time, and remaining size can be scheduled in any order.

You will find that this scheduler is similar in many ways to `srpt_sched`.

Submission Properly implement the scheduler in a file "`priority_sched.c`" in your repository that registers the scheduler "`priority_sched`".

7.5 Task 4: Round Robin Scheduler

Next, you will build a scheduler that preempts by time. The idea is that you have a hardware device, a timer, that can invoke the scheduler periodically, rather than just having the scheduler be invoked on job arrivals and completions.

In your round-robin scheduler, `rr_sched`, whenever the timer goes off, you will pause the current job, put it at the back of the job queue, and then resume the job at the front of the job queue. This means that the scheduler will quickly cycle through existing jobs. Otherwise, it works like `fifo_sched`

The basic benefits of such a scheduler are that it is simple, assures that all jobs make progress, and spreads the pain of high load. Suppose you have 10 jobs in your queue and you are working at 100 Hz. Each one runs for 0.01 seconds over every 0.1 second interval. Or, you can think that each one runs 10 times slower than if it was running alone.

An important property of this kind of scheduler is that it is useful for interactive jobs (those that have a user interface). An interactive job in a typical round-robin environment will make a little bit of progress over every human-perceptible interval of time. Additionally, round-robin can create the illusion that multiple interactive jobs are all running "at the same time". They really are not—it's just that the kernel juggles jobs so fast that it looks like a blur to a human user.

To implement the round-robin scheduler, every time a job is run, a timer should be started with a duration equal to `context->quantum` (10 ms by default). To simulate a hardware timer, you can create a timer event for the current time plus the value of `context->quantum`. Note that you must use the quantum value provided by the `context` and we may change the value of the quantum during testing.

You may change the value of the quantum during testing by setting the `QUEUESIM_QUANTUM` environment variable while running `queuesim`. For example you can set the quantum to one second with:

```
$ QUEUESIM_QUANTUM=1 ./queuesim rr_sched workloads/example.txt
```

A common mistake when implementing the round-robin scheduler is to continue with a partially elapsed timer when scheduling a new job after a previous job has completed. Timeslices are associated with a running job, so resuming a different job means starting a new timer as well. Note that if a job is paused/stopped for any reason other than a timer, you may need to cancel the outstanding timer event. The scheduler is in charge of determining when the timer should, or should not, be running.

Unlike previous preemptive schedulers, a round-robin scheduler does not preempt upon job arrival. Newly arriving jobs should be added to the end of the jobqueue.

You can see an example of a real round-robin scheduler in NK, if you are curious. Note that the code will not help you here, and your simulator-based implementation will be *much* simpler.

Submission Properly implement the scheduler in a file "`rr_sched.c`" in your repository that registers the scheduler "`rr_sched`".

7.6 Task 5: Stride Scheduler

The final scheduler to implement is a stride scheduler.[7] This is similar to a round-robin scheduler, except rather than all jobs being given equal time, jobs are given timeslices proportionally to their static priority.

In your stride scheduler, `stride_sched`, you will track dynamic priority values for each job. When the timer for a running job expires, its dynamic priority should be increased by its *stride* value. Then the job with the lowest dynamic priority will be scheduled to run (which may be the same job).

Stride values can be calculated for each job based on its static priority. A higher priority results in a smaller stride and the job running more often. You can calculate stride with the following equation:

$$stride = 1000000/static\_priority$$

The one million value is included to make the stride value a whole number when truncated and stored as a `uint64_t`. Just like in `priority_sched`, static priority values are in the range of 0–99 inclusive, with higher numerical values representing a higher priority job.

[7]The stride scheduler was first described by Waldspurger et al. in 1995. You can find the academic paper at http://publications.csail.mit.edu/lcs/pubs/pdf/MIT-LCS-TM-528.pdf.

When the first job arrives, its `dynamic_priority` should be set to zero. If there are one or more existing jobs, the newly arriving job should have its `dynamic_priority` set to the currently running job's `dynamic_priority`.[8] Newly arriving jobs do not preempt the currently running job.

Whenever a job's timer expires, update its dynamic priority based on its stride. Then run the job with the lowest `dynamic_priority` value. Similarly, when a job completes, the remaining job with the lowest `dynamic_priority` should be scheduled for an entire timer quantum. Ties in `dynamic_priority` should be broken by arrival time, and then remaining size.

The calculation of stride has a special case for jobs with a static priority of zero. Any job with a static priority of zero should only be scheduled after *all* jobs with non-zero priority are completed. Since they all have equal priority, the remaining jobs should be scheduled in a round-robin fashion. You may want to initially ignore this special case and implement it last.

A tip for implementing the zero-static-priority case: you should just `enqueue` such arriving jobs at the end of the job queue, rather than attempting to place them in a certain order in the queue. However, make sure that doing so doesn't break non-zero-static-priority jobs, which should still be sorted by `dynamic_priority` and come first in the queue.

Like the round-robin `rr_sched` implementation, you must use `context->quantum` as the duration for timeslices. You can use the `QUEUESIM_QUANTUM` environment variable to test your scheduler with different timer quantum values.

Submission Properly implement the scheduler in a file "`stride_sched.c`" in your repository that registers the scheduler "`stride_sched`".

7.7 Task 6: Scheduler Evaluation

In your final task, you will apply each of your schedulers to several different workloads and report on the results.

We have created a number of workloads with jobs that have an exponential random interarrival time with a mean of 1.0 seconds. These jobs vary in size from 0.1 to 1.5 seconds in duration. As new jobs are coming in each second, this means the load on the system is set from 0.1 to 1.5 (overload). The workloads can be found in `workloads/` and are labeled with their mean job size.

For each workload, we want you to see how the mean and standard deviation of the turnaround time and slowdown are effected by the choice of scheduler. These metrics are in the last few lines of the output from `queuesim`. You will make a table of the results and then comment on the table. Put your valuation in a file named `RESULTS`. (Your table could go in the `RESULTS` file or be a separate spreadsheet if preferred.)

Submission A `RESULTS` file with a table of results for all schedulers and all provided exponential workloads including commentary on the results.

[8] This is a simplification for this lab. See section 2.2 of the paper if you want to understand a more precise method for handling new arrivals.

8 Grading

Your group should regularly push commits to Github. You also should create a file named STATUS in which you regularly document (and push) what is going on, TODOs, what is working, etc. Your commits are visible to us, but not to anyone else outside of your group. The commits that we see up to deadline will constitute your hand-in of the code. Commits after the deadline will be considered a late submission. The STATUS file should, upon submission, clearly document that state of your lab (what works, what doesn't, etc).

Your submitted code should contain a file for each scheduler you were tasked with creating. In addition, you will add and push a file RESULTS which will contain your evaluation (Task 6).

You will also need to submit your files to Gradescope. To do so, in the root of the repository run:

```
$ make submit
```

That will upload each of your scheduler files to Gradescope along with your STATUS and RESULTS files. After you submit, you'll need to mark your group members on your submission. Unfortunately, you will need to do this each time you submit.

We will run your schedulers on our own workloads and compare their results to known correct implementations. In all cases, the total time to complete all events should match. An mismatch in completion time demonstrates that the scheduler is not properly scheduling jobs and will result in loss of most points for the scheduler.

Assuming completion time matches, turnaround time should also match our solutions. All schedulers are deterministic, and so there should not be variation in turnaround for correct implementations. However, we do provide a little wiggle room on the expected value to handle edge cases.

Finally, we also look at the number of timer events that have occurred, but this is mostly used as a demonstration that the scheduler is (or isn't) using timer preemption as expected. We provide a large window of acceptable number of timer interrupts for all workloads.

The breakdown in score will be as follows:

- 10% Task 1 `sjf_sched` functional and gives reasonable results.
- 20% Task 2 `srpt_sched` functional and gives reasonable results.
- 10% Task 3 `priority_sched` functional and gives reasonable results.
- 20% Task 4 `rr_sched` functional and gives reasonable results.
- 25% Task 5 `stride_sched` functional and gives reasonable results.
- 15% Task 6 Evaluation of schedulers. Note that for the comment part there is no right answer—we want your thoughts.

9 Extra Credit

We will allow up to 20% extra credit in this lab. If you would like to do extra credit, please complete the main part of the lab first, then reach out to the instructor and TAs with a plan. Some possible extra credit concepts are the following:

- Implement a Lottery Scheduler, `lottery_sched`. This is a form of preemptive scheduler in which priorities are interpreted probabilistically. The idea here is that when your timer fires (or there is a new job arrival), you will create a probability distribution over all the jobs in the job queue, and then select the next job based on that probability distribution. If there are n jobs, then the probability of picking job i is:
$$\frac{priority(job_i)}{\sum_{j=0}^{n-1} priority(job_j)}$$

- Study your schedulers' behaviors in more depth by creating your own workloads. The program `make_arrivals.pl` can generate an arrival process with both exponential and power-law job sizes. See what happens in both cases as the load increases. Queueing theory indicates that power-law job sizes will behave very differently than exponential job sizes.

- Implement hard real-time support. One part of this extra credit is to put together a preemptive earliest deadline first (EDF) scheduling core for use with the periodic and sporadic real-time tasks. This is also known as a fixed priority scheduler, and you already have one given your previous tasks! Think of the deadline as the priority. And now, you will be managing the real-time queue, not the aperiodic queue.

  The more interesting part of real-time support is admission control, deciding when to accept or reject a new task or job. You only accept a new job/task if it will not cause any deadlines for accepted jobs/tasks to be missed. Implement rate-monotonic or utilization-based admission control for periodic and sporadic real-time tasks that will be scheduled under EDF. The `make_arrivals.pl` tool can produce real-time tasks/jobs in addition to the aperiodic ones you've been using in the rest of the lab.

  We often want to mix real-time and non-real-time (aperiodic) jobs together, which usually means we will want to "fill in" the real-time schedule with (preemptable) aperiodic jobs (from the aperiodic queue), perhaps reserving some time for those jobs. That is another complexity with this task.

- Generalize to I/O. In this lab, jobs never block on I/O or wait on other events - they always want the CPU. Generalize the simulator to support jobs queueing for I/O devices (such as disks) as well as the CPU. I/O device are not typically preemptable, and so I/O device queueing itself is usually FIFO, although more complex schedulers are also possible.

- Generalize to multiple CPUs. In this lab, jobs compete for a single CPU. Generalize the simulator to support multiple CPUs, each of which has its own job queue. Now, when a job arrives, it is placed on some CPU to start ("initial placement") and perhaps may be mobile later ("ongoing placement"). Deciding where to place and when/where to move jobs is a part of the general scheduling problem.