CS 343 Operating Systems, Spring 2022
Queuing Lab: Exploring Scheduling and Learning Discrete Event Simulation

1 Introduction

The purpose of this lab is for you to gain some experience with scheduling and to understand how consequential scheduling can be under conditions of high load. In addition, the 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).

The framework was originally created to support a lab in real-time systems, and adopts a common model from that domain.[1] You may see the terms "periodic", "sporadic", and "aperiodic" in the code and our discussion. "Aperiodic" refers to non-real-time work, and it is the only form of work you need to be concerned about in the lab. If you are curious about real-time, there is extra credit you can do.

You may work in a group of up to three people in this lab. Clarifications and revisions will be posted to Campuswire.

2 Setup

You can work on this lab on any modern Linux system, although we will test your work on the class server (Moore). We will describe the details of how to access the lab repo via Github classroom in on Campuswire. Use this information to clone the repo. At this point you should will have a subdirectory named `queuelab`. If this is on a shared machine, you probably want to mark the directory as private (`chmod 700 queuelab`).

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

- `README.md`: Details about the lab code.
- `Makefile`: The build setup. You will have to extend this as you add more schedulers.

[1] In a real-time system, the user process describes the work to be done and the deadline by which it must be done. If the kernel accepts the work, the kernel guarantees that the work will be done before its deadline. In case of a soft real-time system, the guarantee is weak (usually statistical), while in a hard real-time system, the guarantee is ironclad, and it is a failure for the kernel not to meet a deadline.

- `debug.h`: The debug configuration. You can change which parts of the system will print debug information from here. You will want to add to it to control debugging output from your schedulers.
- `queuesim.c`: The simulator's event loop and `main`.
- `fifo_all.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 may want to read some of them.
- `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.
- `tools/plot*.pl`: Tools to plot data as we run. You can plot to a window or to a PNG file.
- `tools/make_arrivals.pl` and `tools/Gen.pm`: These tools create a starting condition for simulation by creating an initial set of events, particularly job arrivals according to some stochastic process.

- `logs/`: Logged results from running `queuesim`.
- `workloads/`: Various workloads to test your schedulers on.

Please be sure to read the `README` file.

To compile the lab:

```
$ make
```

This will build the program `queuesim` (the simulator itself). The simulator needs an input. You can make these using the `tools/make_arrivals.pl` command, but we also include a simple one to start it, `workloads/example.txt`. To run the simulator using that input file:

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

Here `fifo_all` refers to the scheduler being used (the simplest conceivable model—`fifo_all` just runs jobs in the order received without any preemption), and `example.txt` is the set of initial events, such as job arrivals.

The command will run the simulation and capture the results. After it is done, if you have graphics functional, this will pop up a graph showing running information about jobs as a function of time. Next, 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.

3 Scheduling 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 Size.[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.
- 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, and the job size). 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 and queuesim

`queuesim` is a discrete event simulator. What this means is that instead of simulating the passage of time directly, it instead jumps from event to event. For example, suppose the current time is 100 and a job of

[2]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. 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 a deadline.

[3]The term for this in the performance analysis world is the Job Response Time. We are using the common OS term (and the one the book uses) here to avoid confusion. Sometimes the OS world calls the response time the time from when a job is unblocked to when it begins to run again.

[4]When real-time tasks and jobs are considered, the simulator will also report on whether these jobs meet their deadlines or not.

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 event queue, which stores the events in time order. The simulator main loop simply repeatedly pulls the earliest event from the queue and passes it to a handler until there are no more events in the queue. The handler for an event may insert one or more new events into the event queue. A handler can also update or delete events in the queue. 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.

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.

The following are the kinds of events that can be placed directly into the input file, out of the box. You can also place comments into the file—blank lines or those starting with # are ignored. The first set of events is self-explanatory:

```
arrival_time PERIODIC_TASK_ARRIVAL period slice numiters

arrival_time SPORADIC_JOB_ARRIVAL size absolute_deadline

arrival_time APERIODIC_JOB_ARRIVAL size static_priority
```

The first two of these are for real-time tasks and jobs. You can treat these the same as aperiodic jobs unless you are interested in extra credit. `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 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 queues over time. The simulation will stall until you hit enter.

In addition to the events that can appear in the file, your scheduler may add these three kinds of events to the queue:

```
JOB_DONE

JOB_BLOCKED

TIMER              Timer interrupt
```

Your scheduler must implement the interface shown in `scheduler.h`, specifically the `sim_sched_ops_t` type defines the set of functions that you must have. A `sim_sched_ops_t` combined with pointer to state defines an object. You can look at `fifo_all.c` to see how this is done in practice for this framework. What you are seeing is a common way of implementing interface inheritance in kernel code, or more generally, implementing object-oriented concepts in C.

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.

Now let's look at the functions you will provide in groups:

Admission control

```
sim_sched_acceptance_t
   (*periodic_job_arrival)(void *state,
                           sim_context_t *context,
                           sim_job_t *job);

sim_sched_acceptance_t
   (*sporadic_job_arrival)(void *state,
                           sim_context_t *context,
                           sim_job_t *job);

sim_sched_acceptance_t
   (*aperiodic_job_arrival)(void *state,
                            sim_context_t *context,
                            sim_job_t *job);
```

The idea here is that when a job arrives, your scheduler has the opportunity to veto it. Acceptance testing periodic tasks and sporadic jobs is an important part of real-time scheduler, but ignored in a non real-time scheduler. Aperiodic (i.e., non-real-time) jobs are always accepted. Since you are writing non-real-time schedulers, all of these functions can simply return true.

Here is what the arguments mean: First, `state` points back to the state you registered (see the use of `sim_sched_register()` in `fifo_all.c`) when your scheduler was created. The `context` argument gives you access to simulator context, which includes the event queue (for inserting/modifying/deleting events) and the job queues (which you must maintain).

Job processing We already described the notion of jobs being done, blocked, or interrupted by a timer. Your scheduler will implement these functions, which correspond to those events:

```
void (*job_done)(void *state,
                 sim_context_t *context,
                 sim_job_t *job);

void (*job_blocked)(void *state,
                    sim_context_t *context,
                    sim_job_t *job);

void (*timer_interrupt)(void *state,
                        sim_context_t *context);
```

Unless you are interested in extra credit, you will not need to worry about `job_blocked()` in this lab.

Others Two additional functions complete the interface:

```
int  (*init)(void * state,
             sim_context_t *context);
void (*print)(void *state,
              sim_context_t *context,
              FILE *f);
```

`init()` is called when the simulation is about to start. This allows you to reset your scheduler. `print()` is called when we want you to dump your scheduler's state.

5 Task 0: Run the code, including single-stepping

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

```
$ ./queuesim fifo_all workloads/example.txt single
```

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_all` scheduler. Make sure you understand how the event simulator is working, and how this simple default scheduler is implemented.

The `fifo_all` 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.

6 Task 1: Implement a non-preemptive random scheduler

Your next task is to write your own scheduler, `rand_all`. `rand_all` will act just like `fifo_all`, but instead of selecting the next job from the front of the job queue, it will select a job from a random position in the job queue. Every job in the queue is equally likely to be picked.

This may seem like quite a strange policy for a scheduler to have, but it is surprising how often "random" is a good or at least useful model in systems. It's also a good choice in general in the absence of information.

To implement `rand_all`, you will want to create a file for that will look much like `fifo_all.c`, except in the details of what the scheduler does. That is, you will be able to use `fifo_all.c` as a template. You will also need to modify the `Makefile` to include your new file.

7 Task 2: Implement a 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_all` is that when you select the next job, you will choose the one which has the smallest size (shortest service time).

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.

8 Task 3: Implement a preemptive shortest remaining processing time scheduler

Now you will implement your first preemptive scheduler, `srpt_all`. 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_all` is the preemptive variant of `sjf_all`. 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.

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.

9 Task 4: Implement a preemptive 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.

Every time a job is run, a timer should be started with a 10 ms duration. To simulate a hardware timer, you can create a timer event for the current time plus 10 ms. In your round-robin scheduler, `rr_all`, 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 cycle through jobs at 100 Hz (100 times per second). Otherwise, it works like `fifo_all`

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.

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.

Another 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-perceptable 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.

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.

10 Task 5: Implement a preemptive lottery scheduler

In this task, you will implement a preemptive scheduler with priorities that are interpreted probabilistically, `lottery_all`. This scheduler combines the concepts of task 1 (randomness), task 3 (arrival preemption) and task 4 (timer preemption). 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:

$$\frac{priority(job_i)}{\sum_{j=0}^{n-1} priority(job_j)}$$

In your scheduler, you will use the `static_priority` that is given as part of a job arrival. A higher numerical value represents a higher priority. For jobs with a priority of zero, they should have no chance of being scheduled until all higher priority jobs already are (i.e., they should have zero odds of getting chosen). If all remaining jobs have zero priority, they can be scheduled in any order.

You can see an example of a real lottery 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.

11 Task 6: Evaluate schedulers on supplied workloads

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 variance 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.)

12 Testing

It is very important to work 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 can manually create new workloads to test on following the example of `workloads/example.txt`.

You can use the `tools/make_arrivals.pl` script to generate longer test sequences. We used it to create the other workloads in `workloads/`. Running lots of jobs like this can help you find bugs where you crash or have other issues. The below examples created the workloads already in `workloads/` but you run modified versions to create additional workloads to test with. Particularly, you could modify the `SEED` and/or number of jobs created. Be sure to not overwrite the default workloads as you'll need them for Task 6!

```
$ export PERL5LIB=./tools/
$ SEED=42 ./tools/make_arrivals.pl 1000 expexp 1.0 0.1 > workloads/expexp0.1.txt
$ SEED=42 ./tools/make_arrivals.pl 1000 expexp 1.0 0.5 > workloads/expexp0.5.txt
$ SEED=42 ./tools/make_arrivals.pl 1000 expexp 1.0 0.9 > workloads/expexp0.9.txt
$ SEED=42 ./tools/make_arrivals.pl 1000 expexp 1.0 0.95 > workloads/expexp0.95.txt
$ SEED=42 ./tools/make_arrivals.pl 1000 expexp 1.0 0.99 > workloads/expexp0.99.txt
$ SEED=42 ./tools/make_arrivals.pl 1000 expexp 1.0 1.5 > workloads/expexp1.5.txt
```

13 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. The `STATUS` file should, at that point, clearly document that state of your lab (what works, what doesn't, etc). In addition, you will add and push a file `RESULTS` which will contain your evaluation (Task 6).

The breakdown in score will be as follows:

- 10% Task 1 `rand_all` functional and gives reasonable results.
- 10% Task 2 `sjf_all` functional and gives reasonable results.
- 20% Task 3 `srpt_all` functional and gives reasonable results.
- 20% Task 4 `rr_all` functional and gives reasonable results.
- 20% Task 5 `lottery_all` functional and gives reasonable results.
- 20% Task 6 Evaluation of schedulers. Note that for the comment part there is no right answer—we want your thoughts.

14 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 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—it guarantees that the highest priority job is always the one currently running.
- Implement rate-monotonic or utilization-based admission control for periodic and sporadic real-time tasks that will be scheduled under EDF.
- Implement the classic dynamic priority scheduler from Unix. Some variant of the Unix scheduler is part of or influences many modern operating systems. A dynamic priority scheduler varies a job's priority as it executes to try to balance between being very responsive for interactive users (humans) by boosting the priority of jobs that look interactive, while simulaneously letting non-interactive jobs aggressively soak up CPU time.[5]
- Implement the silliest scheduler that does still schedule and complete jobs in a reasonable amount of time. This is worth very little extra credit, but will make the course staff chuckle.

[5] A Unix job's priority is typically the sum of its dynamic priority and its static priority (like in this lab). The static priority is called the "niceness" level. You can read about the "nice" and "renice" commands if you're curious.