This homework covers OSTEP chapters 7-8. It is due in class Friday, February 13. Hand in a hardcopy of your answers in class.

While you may discuss ideas and strategies for problems with other students, you should always make the first attempt on a problem yourself and **you must write up your own solutions in your own words**. *You may not collaboratively write solutions or copy a solution that one person in the group writes up. You also may not look for, copy, or use solutions from other sources, including from generative AI like ChatGPT, even if you make changes. There's no such thing as using someone else's solution for a problem "as an example" for writing your own.*

Preliminaries

- Copy the directories `cpu-sched` and `cpu-sched-mlfq` (and their contents) from `/classes/cs331` to your `~/cs331` directory. (You won't be writing any code, so they don't need to go into your workspace directory.)
- Each directory contains a python program (`scheduler.py` and `mlfq.py`, respectively) and a `README` with instructions for using the program. You can also find the text of the `README`s in the appendix sections A and B at the end of this document.

Exercises

Chapter 7 homework problems (at the very end of chapter 7) —

1. Do #1–3 for practice — make sure you understand the concepts (how the different schedulers work, how to compute response time and turnaround time) and how to run and use `scheduler.py`, but you don't need to hand in your solutions. (Remember that you can check your work by running `scheduler.py` with the `-c` option.)
2. For each of #4–7, try to answer the question(s) posed based just on your understanding of the different schedulers before using the simulator. Your writeup should answer the question(s) posed and provide support for your answer — give command line(s) for simulator runs producing the relevant data.

Chapter 8 homework problems (at the very end of chapter 8) —

3. Do #1 for practice — make sure you understand the concepts (how the MLFQ scheduler works, how to compute an execution trace) and how to run and use `mlfq.py`, but you don't need to hand in your solution. (Remember that you can check your work by running `mlfq.py` with the `-c` option.)

4. For #2–3, give the command line you'd run `mlfq.py` with to produce the desired behavior and explain how that command line achieves the desired result. "Each of the examples" refers to the cases shown in figures 8.2–8.6. If there are any you can't quite capture, get as close as you can and explain what isn't captured.

5. For #4, give the command line for running `mlfq.py` and explain your answer — what is it about the specific workload and the scheduler parameters chosen that results in that behavior?

6. For #5, answer the question and give a command line for running `mlfq.py` that demonstrates the behavior described.

7. For #6, describe the effect you see — how does the `-I` flag affect things? Also give command line(s) for running `mlfq.py` that demonstrate the effect.

Real systems —

8. Read the three articles below about scheduling in MacOS, Windows, and Linux. Briefly describe the priority levels in each of the three systems, then compare and contrast them — what are the similarities and differences?

    - MacOS: (only read the section titled "Overview of Scheduling") https://developer.apple.com/library/archive/documentation/Darwin/Conceptual/KernelProgramming/scheduler/scheduler.html
    - Windows: https://learn.microsoft.com/en-us/windows/win32/procthread/scheduling-priorities
    - Linux: https://medium.com/@chetaniam/a-brief-guide-to-priority-and-nice-values-in-the-linux-ecosystem-fb39e49815e0

A scheduler.py

This program, scheduler.py, allows you to see how different schedulers perform under scheduling metrics such as response time, turnaround time, and total wait time. Three schedulers are "implemented": FIFO, SJF, and RR.

There are two steps to running the program.

First, run without the -c flag: this shows you what problem to solve without revealing the answers. For example, if you want to compute response, turnaround, and wait for three jobs using the FIFO policy, run this:

```
prompt> ./scheduler.py -p FIFO -j 3 -s 100
```

If that doesn't work, try this:

```
prompt> python ./scheduler.py -p FIFO -j 3 -s 100
```

This specifies the FIFO policy with three jobs, and, importantly, a specific random seed of 100. If you want to see the solution for this exact problem, you have to specify this exact same random seed again. Let's run it and see what happens. This is what you should see:

```
prompt> ./scheduler.py -p FIFO -j 3 -s 100
ARG policy FIFO
ARG jobs 3
ARG maxlen 10
ARG seed 100

Here is the job list, with the run time of each job:
  Job 0 (length = 1)
  Job 1 (length = 4)
  Job 2 (length = 7)
```

Compute the turnaround time, response time, and wait time for each job. When you are done, run this program again, with the same arguments, but with -c, which will thus provide you with the answers. You can use -s <somenumber> or your own job list (-l 10,15,20 for example) to generate different problems for yourself.

As you can see from this example, three jobs are generated: job 0 of length 1, job 1 of length 4, and job 2 of length 7. As the program states, you can now use this to compute some statistics and see if you have a grip on the basic concepts.

Once you are done, you can use the same program to "solve" the problem and see if you did your work correctly. To do so, use the "-c" flag. The output:

```
prompt> ./scheduler.py -p FIFO -j 3 -s 100 -c
ARG policy FIFO
```

```
ARG jobs 3
ARG maxlen 10
ARG seed 100

Here is the job list, with the run time of each job:
  Job 0 (length = 1)
  Job 1 (length = 4)
  Job 2 (length = 7)

** Solutions **

Execution trace:
  [time   0] Run job 0 for 1.00 secs (DONE)
  [time   1] Run job 1 for 4.00 secs (DONE)
  [time   5] Run job 2 for 7.00 secs (DONE)

Final statistics:
  Job   0 -- Response: 0.00  Turnaround 1.00  Wait 0.00
  Job   1 -- Response: 1.00  Turnaround 5.00  Wait 1.00
  Job   2 -- Response: 5.00  Turnaround 12.00  Wait 5.00

  Average -- Response: 2.00  Turnaround 6.00  Wait 2.00
```

As you can see from the figure, the -c flag shows you what happened. Job 0 ran first for 1 second, Job 1 ran second for 4, and then Job 2 ran for 7 seconds. Not too hard; it is FIFO, after all! The execution trace shows these results.

The final statistics are useful too: they compute the "response time" (the time a job spends waiting after arrival before first running), the "turnaround time" (the time it took to complete the job since first arrival), and the total "wait time" (any time spent ready but not running). The stats are shown per job and then as an average across all jobs. Of course, you should have computed these things all before running with the "-c" flag!

If you want to try the same type of problem but with different inputs, try changing the number of jobs or the random seed or both. Different random seeds basically give you a way to generate an infinite number of different problems for yourself, and the "-c" flag lets you check your own work. Keep doing this until you feel like you really understand the concepts.

One other useful flag is "-l" (that's a lower-case L), which lets you specify the exact jobs you wish to see scheduled. For example, if you want to find out how SJF would perform with three jobs of lengths 5, 10, and 15, you can run:

```
prompt> ./scheduler.py -p SJF -l 5,10,15
ARG policy SJF
ARG jlist 5,10,15
```

```
Here is the job list, with the run time of each job:
  Job 0 (length = 5.0)
  Job 1 (length = 10.0)
  Job 2 (length = 15.0)
...
```

And then you can use -c to solve it again. Note that when you specify the exact jobs, there is no need to specify a random seed or the number of jobs: the jobs lengths are taken from your comma-separated list.

Of course, more interesting things happen when you use SJF (shortest-job first) or even RR (round robin) schedulers. Try them and see!

And you can always run

```
prompt> ./scheduler.py -h
```

to get a complete list of flags and options (including options such as setting the time quantum for the RR scheduler).

B mlfq.py

This program, `mlfq.py`, allows you to see how the MLFQ scheduler presented in this chapter behaves. As before, you can use this to generate problems for yourself using random seeds, or use it to construct a carefully-designed experiment to see how MLFQ works under different circumstances. To run the program, type:

```
prompt> ./mlfq.py
```

Use the help flag (-h) to see the options:

```
Usage: mlfq.py [options]

Options:
  -h, --help            show this help message and exit
  -s SEED, --seed=SEED  the random seed
  -n NUMQUEUES, --numQueues=NUMQUEUES
                        number of queues in MLFQ (if not using -Q)
  -q QUANTUM, --quantum=QUANTUM
                        length of time slice (if not using -Q)
  -a ALLOTMENT, --allotment=ALLOTMENT
                        length of allotment (if not using -A)
  -Q QUANTUMLIST, --quantumList=QUANTUMLIST
                        length of time slice per queue level, specified as
                        x,y,z,... where x is the quantum length for the
                        highest priority queue, y the next highest, and so
                        forth
  -A ALLOTMENTLIST, --allotmentList=ALLOTMENTLIST
                        length of time allotment per queue level, specified as
                        x,y,z,... where x is the # of time slices for the
                        highest priority queue, y the next highest, and so
                        forth
  -j NUMJOBS, --numJobs=NUMJOBS
                        number of jobs in the system
  -m MAXLEN, --maxlen=MAXLEN
                        max run-time of a job (if randomly generating)
  -M MAXIO, --maxio=MAXIO
                        max I/O frequency of a job (if randomly generating)
  -B BOOST, --boost=BOOST
                        how often to boost the priority of all jobs back to
                        high priority
  -i IOTIME, --iotime=IOTIME
                        how long an I/O should last (fixed constant)
  -S, --stay            reset and stay at same priority level when issuing I/O
```

```
  -I, --iobump          if specified, jobs that finished I/O move immediately
                        to front of current queue
  -l JLIST, --jlist=JLIST
                        a comma-separated list of jobs to run, in the form
                        x1,y1,z1:x2,y2,z2:... where x is start time, y is run
                        time, and z is how often the job issues an I/O request
  -c                    compute answers for me
```

There are a few different ways to use the simulator. One way is to generate some random jobs and see if you can figure out how they will behave given the MLFQ scheduler. For example, if you wanted to create a randomly-generated three-job workload, you would simply type:

```
prompt> ./mlfq.py -j 3
```

What you would then see is the specific problem definition:

```
Here is the list of inputs:
OPTIONS jobs 3
OPTIONS queues 3
OPTIONS allotments for queue  2 is   1
OPTIONS quantum length for queue  2 is  10
OPTIONS allotments for queue  1 is   1
OPTIONS quantum length for queue  1 is  10
OPTIONS allotments for queue  0 is   1
OPTIONS quantum length for queue  0 is  10
OPTIONS boost 0
OPTIONS ioTime 5
OPTIONS stayAfterIO False
OPTIONS iobump False


For each job, three defining characteristics are given:
  startTime : at what time does the job enter the system
  runTime   : the total CPU time needed by the job to finish
  ioFreq    : every ioFreq time units, the job issues an I/O
              (the I/O takes ioTime units to complete)

Job List:
  Job  0: startTime   0 - runTime  84 - ioFreq   7
  Job  1: startTime   0 - runTime  42 - ioFreq   3
  Job  2: startTime   0 - runTime  51 - ioFreq   4

Compute the execution trace for the given workloads.
```

```
If you would like, also compute the response and turnaround
times for each of the jobs.

Use the -c flag to get the exact results when you are finished.
```

This generates a random workload of three jobs (as specified), on the default number of queues with a number of default settings. If you run again with the solve flag on (-c), you'll see the same print out as above, plus the following:

```
Execution Trace:

[ time 0 ] JOB BEGINS by JOB 0
[ time 0 ] JOB BEGINS by JOB 1
[ time 0 ] JOB BEGINS by JOB 2
[ time 0 ] Run JOB 0 at PRIORITY 2 [ TICKS 9 ALLOT 1 TIME 83 (of 84) ]
[ time 1 ] Run JOB 0 at PRIORITY 2 [ TICKS 8 ALLOT 1 TIME 82 (of 84) ]
[ time 2 ] Run JOB 0 at PRIORITY 2 [ TICKS 7 ALLOT 1 TIME 81 (of 84) ]
[ time 3 ] Run JOB 0 at PRIORITY 2 [ TICKS 6 ALLOT 1 TIME 80 (of 84) ]
[ time 4 ] Run JOB 0 at PRIORITY 2 [ TICKS 5 ALLOT 1 TIME 79 (of 84) ]
[ time 5 ] Run JOB 0 at PRIORITY 2 [ TICKS 4 ALLOT 1 TIME 78 (of 84) ]
[ time 6 ] Run JOB 0 at PRIORITY 2 [ TICKS 3 ALLOT 1 TIME 77 (of 84) ]
[ time 7 ] IO_START by JOB 0
IO DONE
[ time 7 ] Run JOB 1 at PRIORITY 2 [ TICKS 9 ALLOT 1 TIME 41 (of 42) ]
[ time 8 ] Run JOB 1 at PRIORITY 2 [ TICKS 8 ALLOT 1 TIME 40 (of 42) ]
[ time 9 ] Run JOB 1 at PRIORITY 2 [ TICKS 7 ALLOT 1 TIME 39 (of 42) ]
...

Final statistics:
  Job  0: startTime   0 - response   0 - turnaround 175
  Job  1: startTime   0 - response   7 - turnaround 191
  Job  2: startTime   0 - response   9 - turnaround 168

  Avg  2: startTime n/a - response 5.33 - turnaround 178.00
```

The trace shows exactly, on a millisecond-by-millisecond time scale, what the scheduler decided to do. In this example, it begins by running Job 0 for 7 ms until Job 0 issues an I/O; this is entirely predictable, as Job 0's I/O frequency is set to 7 ms, meaning that every 7 ms it runs, it will issue an I/O and wait for it to complete before continuing. At that point, the scheduler switches to Job 1, which only runs 2 ms before issuing an I/O. The scheduler prints the entire execution trace in this manner, and finally also computes the response and turnaround times for each job as well as an average.

You can also control various other aspects of the simulation. For example, you can specify how many queues you'd like to have in the system (-n) and what the quantum length should be for all of those queues (-q); if you want even more control and varied quanta length per queue, you can instead specify the length of the quantum (time slice) for each queue with -Q, e.g., -Q 10,20,30] simulates a scheduler with three queues, with the highest-priority queue having a 10-ms time slice, the next-highest a 20-ms time-slice, and the low-priority queue a 30-ms time slice.

You can separately control how much time allotment there is per queue too. This can be set for all queues with -a, or per queue with -A, e.g., -A 20,40,60 sets the time allotment per queue to 20ms, 40ms, and 60ms, respectively. Note that while the chapter talks about allotments in terms of time, here it is done in terms of number of time slices, i.e., if the time slice length for a given queue is 10 ms, and the allotment is 2, the job can run for 2 time slices (20 ms) at that queue level before moving down in priority.

If you are randomly generating jobs, you can also control how long they might run for (-m), or how often they generate I/O (-M). If you, however, want more control over the exact characteristics of the jobs running in the system, you can use -l (lower-case L) or –jlist, which allows you to specify the exact set of jobs you wish to simulate. The list is of the form: x1,y1,z1:x2,y2,z2:... where x is the start time of the job, y is the run time (i.e., how much CPU time it needs), and z the I/O frequency (i.e., after running z ms, the job issues an I/O; if z is 0, no I/Os are issued).

For example, if you wanted to recreate the example in Figure 8.3 you would specify a job list as follows:

```
prompt> ./mlfq.py --jlist 0,180,0:100,20,0 -q 10
```

Running the simulator in this way creates a three-level MLFQ, with each level having a 10-ms time slice. Two jobs are created: Job 0 which starts at time 0, runs for 180 ms total, and never issues an I/O; Job 1 starts at 100 ms, needs only 20 ms of CPU time to complete, and also never issues I/Os.

Finally, there are three more parameters of interest. The -B flag, if set to a non-zero value, boosts all jobs to the highest-priority queue every N milliseconds, when invoked as such:

```
prompt> ./mlfq.py -B N
```

The scheduler uses this feature to avoid starvation as discussed in the chapter. However, it is off by default.

The -S flag invokes older Rules 4a and 4b, which means that if a job issues an I/O before completing its time slice, it will return to that same priority queue when it resumes execution, with its full allotment intact. This enables gaming of the scheduler.

Finally, you can easily change how long an I/O lasts by using the -i flag. By default in this simplistic model, each I/O takes a fixed amount of time of 5 milliseconds or whatever you set it to with this flag.

You can also play around with whether jobs that just complete an I/O are moved to the head of the queue they are in or to the back, with the -I flag. Check it out, it's fun!