HW 3: **Scheduling**

CS 162

Due: October 17, 2021 @ 11:59PM

Contents

1 Introduction

As you may have noticed, this homework is somewhat different than the other ones you have encountered (or will encounter) in this class: rather than a systems-level coding exercise, it's an Jupyter notebook! This means that the homework will be more conceptual than others in this course; of course, there's still coding to do, albeit in Python rather than in C. If you've taken EE16a, you should be familiar with Jupyter notebooks; if not, you may want to check out an online tutorial like this one[1] to get your bearings.

The easiest way to run this notebook is on your personal machine — doing so on the instructional machines is possible, but more difficult. In order to do so, you'll need to make sure your local environment includes Python3 as well as the Jupyter[2] software that will run the notebook itself. Finally, you will need to install the Python packages NumPy[3] and Matplotlib[4].

To complete this assignment, submit your finished Jupyter notebook to Gradescope.

2 Implementing Scheduling Algorithms

For both of these problems, we intend that you implement the schedulers in the IPython notebook and then run the simulation to obtain the CPU log output. We will not collect your code from the notebook, however, so you are free to execute the scheduling algorithm manually to obtain the log output if you wish.

2.1 Shortest Remaining Time First

Implement the SRTF scheduler. Whenever two threads have the same time remaining until completion, break ties by their order of arrival in FIFO fashion; the tests for this component rely on this behavior.
Run it on `workload3` in the IPython notebook with a quantum of 2; when it is complete, report the CPU log output here. Cell 21 sets this up for you.

2.2 Multilevel Feedback Queue

Implement the MLFQ scheduler. Use two queues, one high-priority queue for interactive tasks and one low-priority queue for CPU-bound tasks, each serviced in a round-robin fashion. Each CPU burst begins in the high-priority queue; if its quantum expires before the CPU burst ends, it is demoted to the low-priority queue. Note that, unlike in the model used in discussion, the low priority queue is **not** serviced in first-come-first-served fashion.
Run it on `workload3` in the IPython notebook, using a quantum of 4 in the low-priority queue and a quantum of 2 in the high-priority queue, and produce the CPU log output here. Cell 27 sets this up for you.

[1] https://realpython.com/jupyter-notebook-introduction/
[2] https://jupyter.org/install
[3] https://numpy.org/
[4] https://matplotlib.org/

3 Approaching 100% Utilization

Consider a sequence B_i of CPU bursts, where (for simplicity) each CPU burst has a fixed length T_S. The first burst, B_0, arrives at time $t = 0$ (i.e., ArrivalTime$(B_0) = 0$). For $i \geq 1$, the arrival time of B_i is given by ArrivalTime$(B_i) =$ ArrivalTime$(B_{i-1}) + X_i$, where the X_i are i.i.d. exponentially distributed random variables with parameter λ.

3.1 Queuing Theory

To allow the CPU bursts to execute concurrently, we model each CPU burst as executing in its own task.

(a) Is this an open system or a closed system? Explain.

(b) What value of λ should we choose, such that the mean time between arrivals is equal to T_S? Explain.

(c) What value of λ should we choose, such that the system runs at 50% utilization on average? Explain.

3.2 Utilization Simulation

Now, set up a simulation in the IPython notebook to model the system with a large number of CPU bursts. Fix some value for T_S. Vary the arrival rate (i.e, λ) starting at a small number, increasing it to approach the value you determined in Part (b). Run the simulation at various points along the way, with multiple trials for each point, **making sure to choose some points where λ is very close to the value you determined in Part (b)**.

(d) As you vary λ as described as above, what happens to CPU utilization? Show a line plot with the arrival rate on the x axis and CPU utilization on the y axis, depicting the results.

(e) As you vary λ as described as above, what happens to the response time for each CPU burst? Show a line plot with the arrival rate on the x axis and response time on the y axis, depicting the results. Be sure to consider both the median response time and the 95th percentile.

(f) Does using a different scheduler affect your answer to Part (e)? (Hint: If you choose to look at SRTF, think carefully about how your implementation breaks ties and how that might affect the median response time.)

(g) Qualitatively explain why running a system at close to 100% throughput results in poor latency.

4 Fairness for CPU Bursts

Consider two periodic tasks, S and T. Each task consists of a series of CPU bursts; after each burst, the task will proactively yield to the scheduler, but we will assume for simplicity that they have zero I/O time between CPU bursts. Let S_i (respectively, T_i) be the random variable that denotes the length of the ith CPU burst. **Assume that the CPU burst lengths (i.e., S_i and T_i) are i.i.d.**

4.1 FCFS Scheduler

Sam, a student in CS 162, wishes to run S and T concurrently on a single CPU. Noticing that S and T use the same burst length distribution, he reasons that the CPU will be shared fairly between the two tasks, even if the scheduler does not enforce fairness. Thus, he chooses a simple FCFS scheduler.

(a) Explain why the length of the FCFS queue never exceeds 2.

(b) What is $\Pr[S_1 < T_1]$?

(c) Suppose S has run for m CPU bursts, where m is large. Using the Central Limit Theorem, characterize $\text{CPUTime}(S)$, the total CPU time spent on S, as a normal distribution parameterized by $\mathbb{E}[S_i]$, $\text{Var}(S_i)$, and m.

(d) After the scheduler has run a large number of CPU bursts, what is $\Pr[n \cdot \text{CPUTime}(S) < \text{CPUTime}(T)]$? Use your approximation from Part (c) and write your answer in terms of $\Phi(x)$, the CDF of the Standard Normal Distribution.

4.2 Quantifying Unfairness

Part (d) quantifies unfairness. For example, the probability that T receives at least 5% more CPU time than S corresponds to $n = 1.05$.

For simplicity, assume now that $\mathbb{E}[S_i] = \sqrt{\text{Var}(S_i)}$. This is satisfied by, for example, the exponential distribution.

(e) Using software of your choice, (e.g. Maple, WolframAlpha, Python, graphing calculator), calculate the probability that one task receives at least 10% more CPU time than the other, for $m = 100$. Be sure to consider both tails of the distribution (i.e., T may get more CPU time than S OR S may get more CPU time than T). How does this change if you use $m = 10000$? Was Sam's reasoning that the CPU allocation will be fair with FCFS correct?

(f) Run a simulation in the IPython notebook to confirm your result from Part (e). Describe the simulation you performed and produce a graph supporting your conclusion.

(g) (Optional Stretch) How would the result be different if Sam were to use a preemptive round-robin scheduler with a small quantum?