CS-350 - Fundamentals of Computing Systems
Homework Assignment #2

Due on September 28, 2022 - Late deadline: September 30, 2022 at 11:59 pm

Prof. Renato Mancuso

Renato Mancuso

Problem 1

You are studying the performance of a program executing on a CPU which features a cache memory to speed up the execution of instructions. Every time the CPU executes an instruction, it can result in a cache "hit" or a cache "miss". Having a cache hit is beneficial to the overall runtime of the program. After studying the cache system, you have measured that an instruction results in a cache hit 30% of the time. This quantity is called *hit rate*. Assume that the hit rate never changes.
The program under analysis begins at the entry point; it then executes exactly 10 instructions. After that the execution might follow path A with probability 80%; or path B with probability 20%. When the program executes path A, it will execute 4 more instructions. After these, it will further execute path C with 6 more instructions or path D with 3 more instructions. From path A, execution will continue to path C with probability 30% and to path D with probability 70%. On path B, the program will execute 10 more instructions. Regardless of the path taken, the program will terminate at the end of path C, D, or B.

a) Provide the PMF for the total number of instructions that will be executed by the program, regardless of whether they result in cache hits or misses.

b) How many instructions will be executed on average across many runs of the program?

c) Assume that we want to observe one run in which the instructions on path are B being executed. How many times, on average, we should execute the program?

d) What is the probability that out of 30 consecutive executions of the program, path D will be executed 5 or more times?

e) What is the probability of observing exactly 10 cache hits in a generic run of the program?

f) What is the average number of hits you expect to see throughout the execution of the entire program?

Problem 2

During your internship at CicekSepeti, you are given the task of monitoring a web server for an online flower and gift store. You notice that at steady state the server is handling, on average, about 23 connections (a.k.a. requests) at any given point in time. You know that the rate with which connections are started is 52 connections per second and that the time it takes a thread to service a given request is 50 milliseconds. Your boss asks you to provide the following:

a) What is the average length of time that each one of these connections lasts?

b) What is the slowdown of the server due to queuing?

c) Assuming that the requests are mostly CPU-intensive, and considering how busy is the server, what can you tell about the least MPL that the server is using to process the requests? And how many CPUs the server must have at the very least to achieve steady-state?

d) What is the utilization of each of the CPUs in the server, assuming it has no more than the strictly required number of CPUs computed above? You can also assume that the workload is perfectly balanced among the CPUs.

e) How many requests per seconds must each CPU be processing?

f) What is the average time that each request spends waiting for service?

g) What assumptions did you have to make to answer the above questions?

Problem 3

You have built a server from spare parts from old machines obtained through Craigstuff. And surprise! surprise! the machine is not exactly all that reliable. So you build two other machines in the same way and now you have three machines, namely M1, M2, and M3 of questionable origins. You want to use these machines to offer a somewhat reliable service to a pool of users who will submit requests to your system. A generic request runs on a single machine and it takes exactly 3 minutes to run regardless of which machine is used.

Next, you have analyzed the behavior and failure rates of these machines and have calculated that (1) M1 has an availability of 96% and an MTBF of 2 minutes; (2) M2 has an availability of 90% and an MTBF of 5 minutes; and (3) M3 has an availability of 85% and an MTBF of 10 minutes.

A request is considered *correctly processed* by your system if it is sent to a machine that is currently in operational state (up) and is stays that way until the request completes processing at that machine.

a) You want to ensure that any incoming request will be correctly processed with a probability of at least 60%. Which ones of three machines satisfy this requirement?

b) Assume that you decide to use M1 for all the requests. How many requests, on average, will fail to be correctly processed until one is correctly handled? Exclude the last successful request from your average.

c) Consider the case when three requests arrive simultaneously and each is routed to a different machine. What is the probability that exactly two out three will complete successfully?

d) Your buddy suggests to direct 30% of the requests to M1, 20% of the requests to M2, and the rest to M3. Assuming that requests arrive at the rate of 18 requests per hour. What would be the actual throughput of the system for requests that complete successfully? Assume that a request that fails for whatever reason is not attempted again.

e) Another friend suggests[1] to split each request into three chunks of 1 minute each. Then, they propose to execute each request such that the first chunk runs for 1 minute on M1. After successful completion of that chunk, the second one runs for 1 minute on M2. And the same with the third and last chunk on M3. The full request will complete successfully only if all the chunks are successfully processed. Is this approach better than executing a request entirely on any of the individual machines?

[1] These friends cannot keep their mouth shut, amirite?

Problem 4

Coding Assignment: In this problem you will develop code to simulate a single-server queue, with selectable queuing policy. The time to process a request at the server is modeled using an exponential distribution, while the arrival process of the requests follows a Poisson distribution (Hint: *recall that if an arrival process is Poissonian, then the inter-arrival time between requests is exponentially distributed*).

a) Write a Java class called "Simulator" that implements a simple discrete event simulator with a single server capable of processing requests. This calls for at least three more classes. First, a Request class should be used to model a unit of workload to be processed in the server. Second, a Server class should be used to enqueue and process arriving requests following a given policy—FIFO or Shortest Job Next (SJN). Third, a Simulator class should be used to control the whole simulation. You are free to design the methods/attributes in these classes as you best see fit. But you should structure the general logic of the simulation following the approach presented in class.

   The simulator should have a method with the following prototype: `void simulate(double time)`, that simulates the arrival and execution of requests at a generic server for `time` milliseconds, where `time` is passed as a parameter to the method. The class should be implemented in its own java file, named `Simulator.java`. The simulator class will internally use a exponentially-distributed random number generator, a timeline and two load generators. For these, you can re-use/adapt the classes you wrote as part of the previous assignment. **Please be mindful of the capitalization in all the files you submit.**

   Apart from implementing the `simulate(...)` method, the class should also include a `main(...)` function. The `main(...)` function should accept 5 parameters from the calling environment (in the following order):

   (a) length of simulation time in milliseconds. This should be passed directly as the `time` parameter to the `simulate(...)` function.

   (b) average arrival rate of requests of class X (λ_x);

   (c) (ignored, passed as 0. Will be used in HW3)

   (d) average service time at the server for requests of class X (Ts_x);

   (e) (ignored, passed as 0. Will be used in HW3)

   (f) queuing policy which can either be "FIFO" or "SJN".

   If you are not familiar with how to pass parameters (a.k.a. command-line arguments) from the calling environment to a Java program, take a look at this: `https://www.journaldev.com/12552/public-static-void-main-string-args-java-main-method`.

   It is responsibility of the `main(...)` function to internally invoke the implemented `simulate(...)` function **only once**. In this first version, the `simulate(...)` function will need to print in the console the simulated time at which each request of class X arrives at the system (`ARR`), initiates service (`START`), and completes service (`DONE`).

   To generate a sequence of `ARR` events with the right inter-timing, reuse the loadGenerator approach from HW1. In this case, the releaseRequest(...) should do TWO things: (1) just like before, generate the next arrival using λ_x; and (2) create a new Request object and pass it to the Server. Thus, the Server class should have some receiveRequest(...) method that can be called by the loadGenerator for this purpose.

   Under the FIFO policy, the order of service is the same as the order of arrival. In this case, the output must look like this:

Problem 4 continued on next page...

```
X0 ARR: <timestamp> LEN: <length>
X0 START: <timestamp>
X1 ARR: <timestamp> LEN: <length>
X2 ARR: <timestamp> LEN: <length>
X0 DONE: <timestamp>
X1 START: <timestamp>
X1 DONE: <timestamp>
X2 START: <timestamp>
X2 DONE: <timestamp>
```

where `<timestamp>` is the simulated time in milliseconds at which the event occurred printed in decimal format, and `<length>` is the service time of the request. **Be extremely careful to follow the format described above. CodeBuddy will be very sensitive to output formatting issues.**

Under the SJN policy, if two or more requests are queued, the one with the shortest service time is picked for execution. Thus, if SJN is selected and assuming that `<x2_length> ¡ <x1_length>`, the correct output will be:

```
X0 ARR: <timestamp> LEN: <length>
X0 START: <timestamp>
X1 ARR: <timestamp> LEN: <x1_length>
X2 ARR: <timestamp> LEN: <x2_length>
X0 DONE: <timestamp>
X2 START: <timestamp>
X2 DONE: <timestamp>
X1 START: <timestamp>
X1 DONE: <timestamp>
```

b) Modify the code of the simulator written in the first part of this problem to measure utilization, average queue length, and average response time of requests. The simulator remains the same in terms of interface, accepted parameters, and simulation logic. However, in addition to printing out the simulation trace just like before, the simulator should add three lines at the end of its output, i.e. after the trace. This should be formatted as:

```
UTIL: <utilization>
QAVG: <avg. number of requests in the system>
WAVG: <avg. number of requests waiting for service>
TRESP: <avg. response time of requests>
TWAIT: <avg. time spent in the queue>
```

where `<utilization>` is a decimal number between 0 and 1, `<avg. queue length>` is a decimal number, and `<avg. response time of requests>` is a decimal number expressed in milliseconds.

Submission Instructions: in order to submit this homework, please follow the instructions below for exercises and code.
The solutions for Problem 1-3 should be provided in PDF format, placed inside a single PDF file named `hw2.pdf` and submitted via Gradescope. Follow the instructions on the class syllabus if you have not received an invitation to join the Gradescope page for this class. You can perform a partial submission before the deadline, and a second late submission before the late submission deadline.

The solution for Problem 4 should be provided in the form of Java source code. To submit your code, place all the `.java` files inside a compressed folder named `hw2`.zip. Make sure they compile and run correctly according to the provided instructions. The first round of grading will be done by running your code. Use CodeBuddy to submit the entire `hw2`.zip archive at `https://cs-people.bu.edu/rmancuso/courses/cs350-fa22/codebuddy.php?hw=hw2`. You can submit your homework multiple times until the deadline. Only your most recently updated version will be graded. You will be given instructions on Piazza on how to interpret the feedback on the correctness of your code before the deadline.