CS-350 - Fundamentals of Computing Systems
Homework Assignment #3

Due on October 6, 2021 — Late deadline: October 8, 2021 EoD at 11:59 pm

Prof. Renato Mancuso

Renato Mancuso

Problem 1

The enterprise database system called *ChupaData* has been deployed over a single-CPU server. The database is being used to store the prices of various Halloween decorations. The database is implemented such that requests are queued in a as they arrive and processed in FIFO order. To process each transaction, a worker thread is spawned, which takes 0.13 seconds on average to complete processing. During the month of October, the DB receives an average of 5 requests per second. But during the month of November, the traffic is expected to double.

a) Compute the capacity of the deployed *ChupaData* DB.

b) If we were to observe the system throughout the month of October, how likely it is to find out that both worker threads are busy processing transactions?

c) Is it fair to advertise our system in October by saying that a generic transaction will start receiving service, on average, after being queued for less than 100 milliseconds?

d) Is it wise to keep the same exact system configuration for the month of November? Motivate your answer.

e) In order to prepare for the increase in traffic in November, you decide to re-deploy the DB on a faster single-CPU machine. What should be the minimum speedup required to handle the November Halloween craze?

f) Assume that you are able to find a machine where each request can be processed in 92 milliseconds. What per-request response time average you expect in November after this upgrade?

g) With the upgrade considered in Question f), what do you expect to be the average number of requests concurrently handled (i.e., either in service or queued) by the DB throughout the month of November?

h) Consider again the upgrade considered in Question f). What is the probability that a generic request performed by some random user in November will begin processing immediately, as soon as it is received.

Problem 2

Good news!! You were able to secure some multi-million funding for your *UnibrowMe* startup. Your system works as follows. Your users upload a selfie and the proprietary UnibrowMe algorithm generates an altered picture where the user is depicted sporting a majestic https://imgur.com/t/electronics/9jgFBID. Now it is time to perform system tuning. After observing the system for a while you have noticed that users submit pictures with an average size of 3 MiB (1 MiB = 1024 KiB; and 1 KiB = 1024 bytes). Every day, you are already receiving about 26,000 UnibrowMe requests. Your investors are eager to know the following:

a) How long should your system spend processing each individual request (on average) so that the system ends up handling on average about 20 requests (either being currently processed or queued) at any given time?

b) Assume that you were able to tune your system so that processing each request takes about 2.5 s, what is the time that each user will end up waiting on average before receiving their unibrow'd selfie?

From now on, consider the system tuned according to what mentioned in Question b).

c) What is the slowdown of the system due to queuing?

d) What is the maximum number of requests that the system can sustain on a daily basis?

e) How much memory will the requests in the system occupy on average, considering that (1) queued requests occupy as much memory as the size of the input image; and (2) requests currently being processed occupy 3× as much until their processing is completed.

f) The manual of the CPU you have deployed states that you should use water-cooling system if your system is going to be busy for more than 48 seconds every minute. Should the investors budget for water cooling?

The investors got back to you saying that you either support way more users per day or the deal is off. Overnight, you devise the following optimization. You organize your system in two stages. At the first stage, pictures are compressed down to 450 KB. After compression, they are sent to the main processing server for *unibrowization* generation. This way, the final processing stage can take as little as 1.1 s. Now you worry about the following:

g) How long should compression take so that the system can sustain 70,000 requests per day?

h) Assume that compression takes 0.9 s, what is going to be the average time it takes your whole system (compression+processing) to take a new request in and produce the corresponding output when your system reaches 70,000 requests per day?

i) With the same assumption on compression time, do you need cooling at the main processing server?

Problem 3

Good news! Right after passing CS350 with flying colors, you have been hired by *TurtleServah Inc.*, a company known to provide processing services with sustainable energy footprint. The company allocates a single-CPU server machine to each customer. Then, given the characteristics of the workload from the customer under analysis, your job is to set the speed of the CPU in the allocated machine to meet the desired quality of service (QoS) while minimizing the power consumption of the machine.
TurtleServah defines the QoS with its customers as the average latency to process each request submitted by that customer. Moreover, the workload submitted by all the customers is always CPU-intensive (i.e. the impact of resources other than the CPU on the time it takes to process any request is negligible). All the machines are able to process 200 requests per second (on average) at the maximum clock speed of 1 GHz. A decrease in clock speed results in a linear increase in service time. For instance, setting the clock speed to 0.5 GHz means that the average service time is doubled.
When a new customer wants to open a contract, you first check that you can actually handle their traffic at the maximum CPU speed. Consider the following customers and compute if at the maximum CPU speed you are able to guarantee the QoS terms.

a) Customer A: desired throughput is 60 requests per second, with average per-request latency of 10 milliseconds;

b) Customer B: desired throughput is 30 requests per second, with average per-request latency of 5 milliseconds;

c) Customer C: desired throughput is 100 requests per second, with average per-request latency of 20 milliseconds;

Consider now only the customers with whom you can stipulate a contract of service. For each of them, determine the following:

d) What is the slowest CPU clock setting that you can use while still meeting the terms of the contract?

e) After setting the CPU at the clock setting considered at the previous step, what is the expected number of customer's requests being queued (but not in service!)?

f) For the customer under analysis, what is the probability that a random request will not experience queuing and will be served immediately by the allocated machine? Again, considering the new clock settings.

Problem 4

Coding Assignment: In this problem you will develop code to simulate a dual-server queuing system. A primary server performs the first part of processing, while a secondary server takes care of the second stage of processing. Request service times at all the servers are 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 dual-server architecture with only one entry and one exit for the requests. That means that each requests arrives at the primary server; upon completion is directed to the secondary server; and once done at the secondary server leaves the system.

The simulator should have a method with the following prototype: `void simulate(double time)`, that simulates the arrival and execution of requests inside the system 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`. **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 4 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 at the system;

(c) average service time at the primary server;

(d) average service time at the secondary server;

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 arrives at the system (`ARR`), initiates service at a given server X (`START X`) where X is 0 for primary, and 1 for secondary server. Once a request completes service at the primary server, it is redirected (`NEXT 1`) to the secondary server. It then completes service (`DONE 1`) at secondary server. Make sure to output the listed statistics at the end. The output must look like this:

```
R0 ARR: <timestamp>
R0 START 0: <timestamp>
R1 ARR: <timestamp>
R2 ARR: <timestamp>
R0 NEXT 1: <timestamp>
R0 START 1: <timestamp>
R1 START 0: <timestamp>
R1 NEXT 1: <timestamp>
R2 START 0: <timestamp>
R0 DONE 1: <timestamp>
R1 START 1: <timestamp>
```

```
UTIL 0: <utilization of primary server>
UTIL 1: <utilization of secondary server>
QLEN 0: <avg. queue length of primary server>
QLEN 1: <avg. queue length of secondary server>
TRESP: <avg. response time of requests>
TWAIT: <avg. time waiting in any queue (not in service)>
```

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

b) Modify the code of the simulator written in the first part of this problem to add split-points for requests. Split points represent alternative paths for the traffic flowing in the system. For this part, we will introduce two split points. (1) The first split-point is added right after the primary server. A request that completes processing at the primary server will leave the system (without processing at the secondary server) with a given probability $P_{0,exit}$. (2) The second split-point is added after the secondary server. A request that completes processing at the secondary server will go back to the primary server for more processing, as if it were a brand new request. This happens with probability $P_{1,0}$.

Write a Java class called "Simulator" that implements a discrete event simulator for the whole system. 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`. **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 6 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 at the system;

(c) average service time at the primary server;

(d) average service time at the secondary server;

(e) probability $P_{0,exit}$;

(f) probability $P_{1,0}$;

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 arrives at the system (`ARR`), initiates service at a given server X (`START X`) where X is 0 for primary, and 1 for secondary server. It might completes service (`DONE X`) at server X, or be sent to server Y (`NEXT Y`) for more processing. Make sure to output the listed statistics at the end. The output must look like this:

```
R0 ARR: <timestamp>
R0 START 0: <timestamp>
```

```
R1 ARR: <timestamp>
R2 ARR: <timestamp>
R0 NEXT 1: <timestamp>
R0 START 1: <timestamp>
R1 START 0: <timestamp>
R1 NEXT 1: <timestamp>
R2 START 0: <timestamp>
R0 NEXT 0: <timestamp>
R1 START 1: <timestamp>

UTIL 0: <utilization of primary server>
UTIL 1: <utilization of secondary server>
QLEN 0: <avg. queue length of primary server>
QLEN 1: <avg. queue length of secondary server>
TRESP: <avg. response time of requests>
TWAIT: <avg. time waiting in the queue (not in service)>
RUNS: <avg. number of servers visited by a request>
```

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 `hw3`.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 `hw3`.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 `hw3`.zip archive at `https://cs-people.bu.edu/rmancuso/courses/cs350-fa21/scores.php?hw=hw3`. 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.