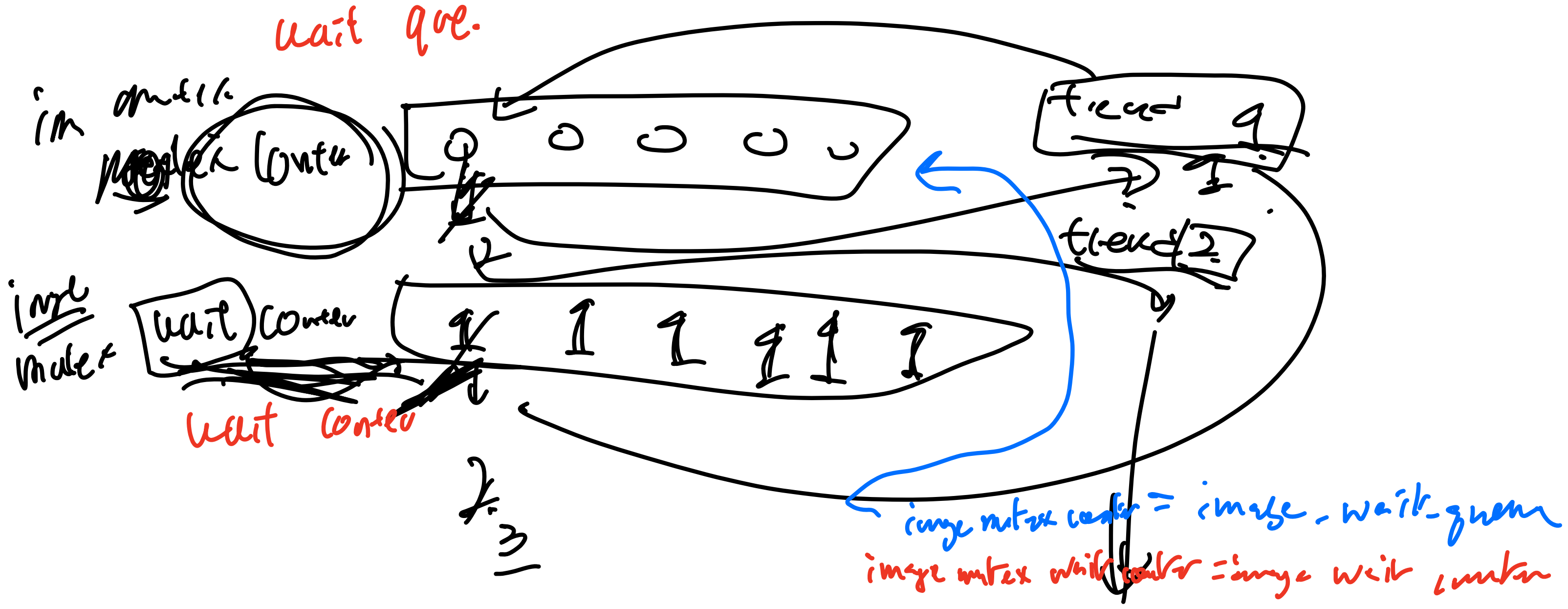
wait que.
im mutex
mutex counter
img mutex
wait counter
wait counter
thread 1
thread 2
image mutex counter = image_wait_queue
image mutex wait counter = image wait counter

CS-350 - Fundamentals of Computing Systems
Homework Assignment #8 - EVAL

Due on November 14, 2024 — Late deadline: November 15, 2024 EoD at 11:59 pm

Prof. Renato Mancuso

Renato Mancuso

EVAL Problem 1

In this EVAL problem, we will study how the nature of input requests can impact the parallelism that can be exploited in multi-threaded applications.

a) Let's start easy for this part of the assignment and establish a clear evaluation methodology and baseline. In particular, we will define a base request pattern and then gradually extend it to understand how the server runtime is impacted. For this part, we will test with only one worker, but the same exact client input scripts you generate for this part will be used for the next part.

The base command template to use for this part is the following:

```
/usr/bin/time -v ./server_mimg -q 1500 -w 1 -p FIFO 2222 & ./client 2222 -I ./images/ \
-L <SCRIPT>
```

This command assumes that both `server_mimg` and the `client` (v4.2!) are in the same current folder, and that the `images` folder contains ALL the test images, **including all the cute cat BMPs!**. There should be a total of 16 images in there.

The `<SCRIPT>` will be comprised of three parts: (1) Intro, (2) Mid, (3) Outro. Intro and Outro will not change, but we will progressively enxted the Mid section of the script. Be careful about not omitting the comma ","at the end of these pieces when concatenating Intro, Mid, and Outro.

(1) Intro (register 10 images):

```
0:R:1:6,0:R:1:7,0:R:1:8,0:R:1:9,0:R:1:10,0:R:1:11,0:R:1:12,0:R:1:13,0:R:1:14,0.5:R:1:15,
```

(2) Mid (repeating pattern):

```
0:r:1:0,0:b:1:0,0:s:1:0,0:v:1:0,0:h:1:0,0:r:1:0,0:b:1:0,0:s:1:0,0:v:1:0,0:h:1:0,\
0:r:1:1,0:b:1:1,0:s:1:1,0:v:1:1,0:h:1:1,0:r:1:1,0:b:1:1,0:s:1:1,0:v:1:1,0:h:1:1,\
0:r:1:2,0:b:1:2,0:s:1:2,0:v:1:2,0:h:1:2,0:r:1:2,0:b:1:2,0:s:1:2,0:v:1:2,0:h:1:2,\
0:r:1:3,0:b:1:3,0:s:1:3,0:v:1:3,0:h:1:3,0:r:1:3,0:b:1:3,0:s:1:3,0:v:1:3,0:h:1:3,\
0:r:1:4,0:b:1:4,0:s:1:4,0:v:1:4,0:h:1:4,0:r:1:4,0:b:1:4,0:s:1:4,0:v:1:4,0:h:1:4,\
0:r:1:5,0:b:1:5,0:s:1:5,0:v:1:5,0:h:1:5,0:r:1:5,0:b:1:5,0:s:1:5,0:v:1:5,0:h:1:5,\
0:r:1:6,0:b:1:6,0:s:1:6,0:v:1:6,0:h:1:6,0:r:1:6,0:b:1:6,0:s:1:6,0:v:1:6,0:h:1:6,\
0:r:1:7,0:b:1:7,0:s:1:7,0:v:1:7,0:h:1:7,0:r:1:7,0:b:1:7,0:s:1:7,0:v:1:7,0:h:1:7,\
0:r:1:8,0:b:1:8,0:s:1:8,0:v:1:8,0:h:1:8,0:r:1:8,0:b:1:8,0:s:1:8,0:v:1:8,0:h:1:8,\
0:r:1:9,0:b:1:9,0:s:1:9,0:v:1:9,0:h:1:9,0:r:1:9,0:b:1:9,0:s:1:9,0:v:1:9,0:h:1:9,
```

(3) Outro (retrieve 10 images):

```
0:T:1:0,0:T:1:1,0:T:1:2,0:T:1:3,0:T:1:4,0:T:1:5,0:T:1:6,0:T:1:7,0:T:1:8,0:R:1:9
```

For the first run, concatenate Intro, Mid, and Outro with no spaces between them. For the second run, concatenate Intro, 2× Mid, and Outro. For the third run, concatenate Intro, 3× Mid, and Outro. And so on, up to a total of 10 runs. The last run will have a script comprised of Intro, 10× Mid, and Outro.

Now, plot the total runtime of each run on the y-axis and the number of times the Mid section was repeated (aka the run number) on the x-axis. You can measure the total server runtime by either adding an extra `clock_gettime(...)` call at the beginng/end of your handle connection function, or by looking at the wall-clock time reported by the `time` utility.

Problem 1 continued on next page. . .

b) For this part, repeat the same commands for the 10 runs above, but only change the number of workers passed to the server. Use `-w 10` instead of `-w 1`.

Produce the same plot as in Part 1, and compare the new line with the performance you measured in Part 1. Comment on the speedup that you observe thanks to the multi-threading that was introduced. Is it more or less than what you expected?

c) For this part, we will run an experiment very similar to Part 2. In particular, we will still use `-w 10` but we will change how the Mid section is constructed. The new Mid section will be:

(2) Mid (repeating pattern):

```
0:r:1:0,0:r:1:1,0:r:1:2,0:r:1:3,0:r:1:4,0:r:1:5,0:r:1:6,0:r:1:7,0:r:1:8,0:r:1:9,\
0:b:1:0,0:b:1:1,0:b:1:2,0:b:1:3,0:b:1:4,0:b:1:5,0:b:1:6,0:b:1:7,0:b:1:8,0:b:1:9,\
0:s:1:0,0:s:1:1,0:s:1:2,0:s:1:3,0:s:1:4,0:s:1:5,0:s:1:6,0:s:1:7,0:s:1:8,0:s:1:9,\
0:v:1:0,0:v:1:1,0:v:1:2,0:v:1:3,0:v:1:4,0:v:1:5,0:v:1:6,0:v:1:7,0:v:1:8,0:v:1:9,\
0:h:1:0,0:h:1:1,0:h:1:2,0:h:1:3,0:h:1:4,0:h:1:5,0:h:1:6,0:h:1:7,0:h:1:8,0:h:1:9,\
0:r:1:0,0:r:1:1,0:r:1:2,0:r:1:3,0:r:1:4,0:r:1:5,0:r:1:6,0:r:1:7,0:r:1:8,0:r:1:9,\
0:b:1:0,0:b:1:1,0:b:1:2,0:b:1:3,0:b:1:4,0:b:1:5,0:b:1:6,0:b:1:7,0:b:1:8,0:b:1:9,\
0:s:1:0,0:s:1:1,0:s:1:2,0:s:1:3,0:s:1:4,0:s:1:5,0:s:1:6,0:s:1:7,0:s:1:8,0:s:1:9,\
0:v:1:0,0:v:1:1,0:v:1:2,0:v:1:3,0:v:1:4,0:v:1:5,0:v:1:6,0:v:1:7,0:v:1:8,0:v:1:9,\
0:h:1:0,0:h:1:1,0:h:1:2,0:h:1:3,0:h:1:4,0:h:1:5,0:h:1:6,0:h:1:7,0:h:1:8,0:h:1:9,
```

Other than that, construct the full commands just like in Part 1. Next, produce one last plot with the new performance trend across the 10 runs. This plot should have three lines: one for Part 1, one for Part 2, and one for the experiment in this part.

Notice that the hashes returned by the client at the end of each run on all the experiments so far *should* match. Thus, the same exact operations are being performed on the 10 images.

Compare now the runtime trend of the system in Part 2 vs. Part 3. What is the additional speedup? Where is the additional improvement coming from if we are performing the same exact operations over the same images?

d) No more experiments here! But just a conceptual question. You see that both the experiments in Part 2 and 3 perform the same exact operations on the images. Yet, the server exhibits different performance.

Is there a better strategy to schedule the requests in the queue that could allow the server to exhibit the same performance on both experiments? No need to overthink this. Provide an exact *scheduler policy* in your answer in a way that, ideally, a person reading your answer should be able to implement the policy you are describing.