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

Due on November 14, 2024 — Late deadline: November 16, 2024 EoD at 11:59 pm

Prof. Renato Mancuso

Renato Mancuso

BUILD Problem 1

We now have a server capable of performing non-trivial operations on images! All we have to do now is to make the server multi-threaded. And yes, we already have the infrastructure to spawn multiple threads. So what's missing exactly?

Output File: `server_mimg.c`

Overview. As mentioned above, the main idea is to allow multple workers to perform operations on images in parallel. Everything you have developed in BUILD 6 will be reused, but we must make sure that when multiple worker threads are spawned, the correctness of the operations on the images is still guaranteed. But what could be jeopardizing the correctness of these operations? Let us consider a concrete case. Imaging the following sequence of requests queued at the server: (1) Retrieve image A, (2) blur image A, (3) detect the vertical edges of image A, and (4) send back the result of the operations performed on image A. With only one worker, the operations are carried out in this sequence and the result sent back to the client is an image with the cumulative operations (2) and (3) correctly applied to the source image. With 2 workers (unless we fix our implementation) we could have worker #1 and #2 working in parallel on operations (1) and (2) with the result of the operations being some weird mix of the two operations.
In this assignment, the goal is to allow safe multi-threading where the semantics of sequential operations on the images is preserved even if multiple threads are spawned and operate in parallel. For this task, we will use semaphores to perform inter-thread synchronization.

Design. One of the main problem that we have to solve is un-arbitrated access to shared data structures. To verify that there is a problem unless we insert synchronization primitives accordingly, start with your (or my) solution for HW6, rename it appropriately, and enable spawning more than 1 worker threads.
Then, run the following simple experiment. First, run the client to generate the sequence of operations listed above with 1 worker thread and look carefully at the output report generated by the client:

-Register -blur -vertedged -Retrieve.

```
./server_mimg -q 100 -w 1 2222 & ./client -I images/ -L 1:R:1:0,0:b:1:0,0:v:1:0,0:T:1:0 2222
```

0 1 2 3

You will notice that the first hash reported by the client (`9f3363f0249c15163d52e60fd9544c31`) is simply the hash of the original `test1.bmp` image. The second (and last) hash reported by the client is the hash (`00e4fc4b9c7c71ee2ca3946053f78793`) of the blur+vertical edge detection operations applied in sequence to the image. However, if we increase the number of worker to 2, the final hash will be different! For instance, when running:

```
./server_mimg -q 100 -w 2 2222 & ./client -I images/ -L 1:R:1:0,0:b:1:0,0:v:1:0,0:T:1:0 2222
```

1)

The last hash obtained on the reference machine changes to `b5932c2bcb0a64121def911286c706e2`, but might be something else entirely on a different machine. Also in some cases, the server crashes entirely.
To solve the problem, the cleanest way is to introduce semaphore-based synchronization between threads. In order to define a semaphore, 1) you should use the type `sem_t` defined in `semaphore.h`. Before a semaphore can be used, it must be initialized. - 정의 2) - init ㄱ세마포어가 프로세스간 공유가되냐 안되냐
This can be done with the `sem_init(sem_t * semaphore, int pshared, unsigned int init_value)`. Here, `semaphore` is pointer to the semaphore to be initialized, `pshared` can be set to 0, and `init_value` is a non-negative initialization value of the semaphore, following the semantics we have covered in class. → 초기값
Once your semaphore has been correctly initialized (make sure to check for the error value of the `sem_init(...)` call!), the wait and signal operations can be performed over it, following the semantics we have discussed 1 or 0

3) wait 4) signal

0 이면 thread간 공유

handle ...tion 에서 sem_wait sempost

11

in class. To wait on a semaphore, you must use the `sem_wait(sem_t * semaphore)` call; to signal on a semaphore, you must use the `sem_post(sem_t * semaphore)`.

Shared Data Structures. Of course, the main question is *what data structures must be protected?*. Here is a list of things that can be problematic, but your own implementation could be different, so try to map the statement below to your own code.

Array 처럼 / register 하면 만들어야 됨

(1) **Image Objects:** One obvious place to start is to protect the image objects that are registered with the server and upon which operations are requested by the client. We want to prevent different worker to simultaneously change the content of an image, so a good idea is to introduce one semaphore per each registered image! These must be created and/or initialized dynamically at image registration time.
(2) **Image Registration Array:** Another shared data structure is the global array of registered images. Concurrent operations over that array is not a good idea, so all the threads will need to synchronize when trying to access that shared data structure.
(3) **Connection Socket:** What? The connection socket has always been shared, so why is that a problem now? The truth is that it has always been a problem, but we did not care because the responses from the workers to the client were always a one-shot `send(...)` operation. But now, there are cases where the server follows a two-step approach in the protocol it follows with the client. For instance, when handling an `IMG_RETRIEVE` operation, a worker first provides a positive acknowledgment of completed request and then the payload of the image being retrieved. What if another worker starts sending other data while a retrieve operation is in progress? Careful: the same goes for the parent when handling `IMG_REGISTER` operations.
(4) **Request Queue and STDOUT Console:** We already know that the shared request queue and the shared STDOUT console require the use of semaphores to ensure correctness. Perhaps take inspiration from the use of semaphores in those cases to handle the other shared data structures listed above.

Desired Output. The expected server output is pretty much what you already constructed in HW6. Here is it summarized again for reference. You should print queue status dumps, rejection and completion notices. Queue status dumps and rejection notice are identical in format to HW5 and HW6. Once again, the queue dump status is printed when *any* of the worker threads completes processing of any of the requests.
Just like HW6, when a request successfully completes service, the **thread ID** of the worker thread that has completed the request will need to be added at the beginning of the line following the format below. You can assign thread ID = (number of workers + 1) to the parent thread. If multiple worker threads are available to process a pending request, any one of them (but only at most one!) can begin processing the next request.

```
T<thread ID> R<req. ID>:<sent ts>,<img_op>,<overwrite>,<client img_id>,<server img_id>,<receipt ts>,
<start ts>,<compl. ts>
```

Here, `<img_op>` is a string representing the requested operation over an image. For instance, if the operation was `IMG_REGISTER`, then the server should output the string "IMG_REGISTER" (no quotes) for this field. `<overwrite>` should just be 0 or 1, depending on what the client requested. `<client img_id>` should be the image ID for which the client has requested an operation. If the server is ignoring any of these values in the response, set these fields to 0. Finally, `<server img_id>` should report the image ID on which the server has performed the operation requested by the client. Recall that this might be different from what sent by the client if `overwrite = 0` in the client's request, but it must be the same if `overwrite = 1`.

Additional Help. You might have noticed, from the commands recommended above, that the client (v4.2) now allows you to define a script of image operation requests. This is useful to test the correctness of your server under a controlled workload.
To use this feature, you should still provide the path to the folder containing the test images using the `-I <path to images folder>` parameter. Next, you should also provide the `-L <images request script>` parameter, where the `<images request script>` is a comma-separated list of image operations with the following format:

Problem 1 continued on next page. . .

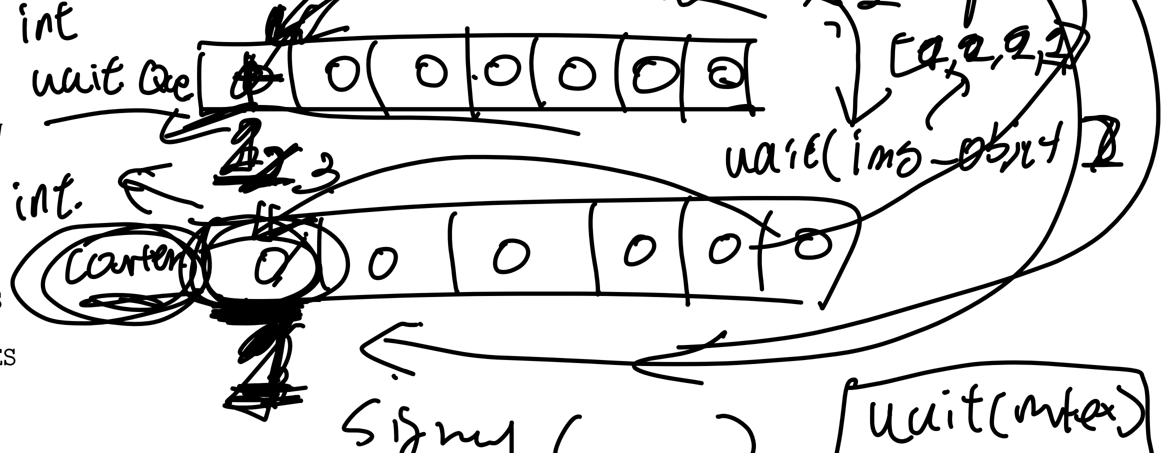
Conner ++c

while(flg[i]) client

`<time to next operation>:<opcode char>:<overwrite>:<image ID>`.

Here, `<time to next operation>` is a number of seconds that will elapse between this and the next operation in the script.
Next, `<opcode char>` is a single case-sensitive (!!) letter that identifies which operation to be performed (see list below).

- R: `IMG_REGISTER`
- r: `IMG_ROT90CLKW`
- b: `IMG_BLUR`
- s: `IMG_SHARPEN`
- v: `IMG_VERTEDGES`
- h: `IMG_HORIZEDGES`
- T: `IMG_RETRIEVE`

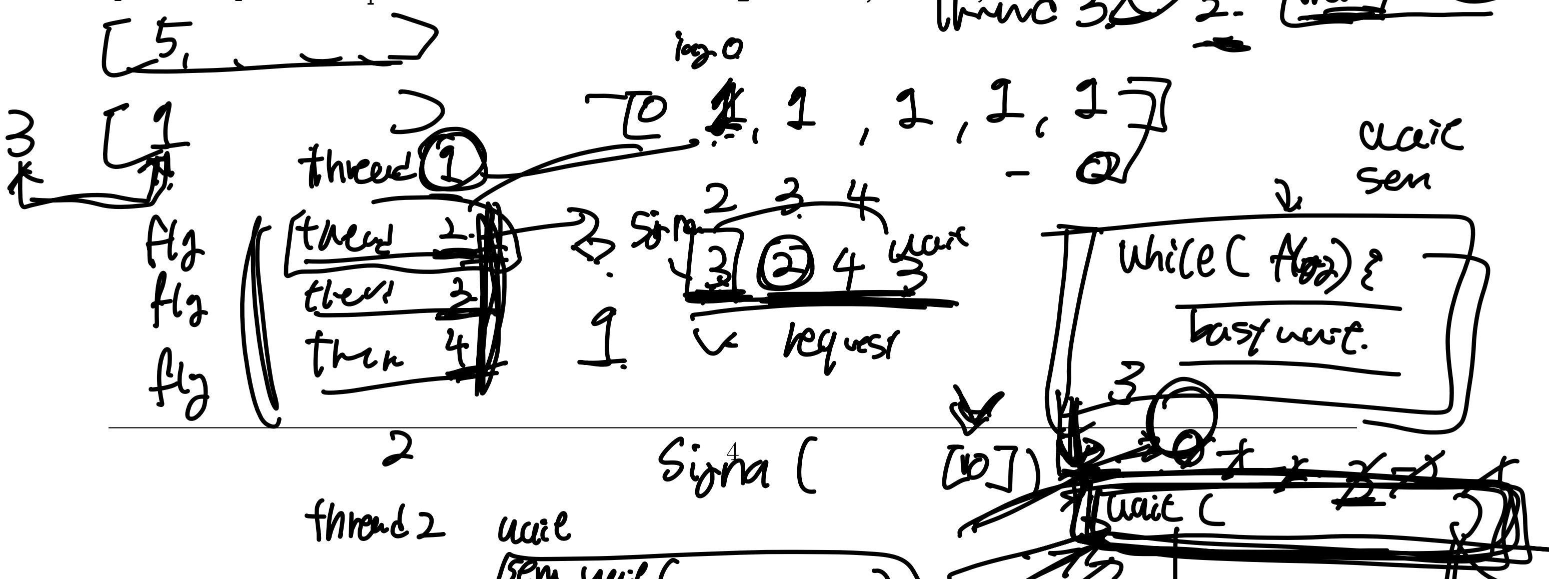


The `<overwrite>` field should always be set to 1 (for simplicity we do not handle cases with overwrite = 0). Finally, the `<image ID>` should be the ID on which the operation should be performed. This field has a special meaning in the case of `IMG_REGISTER` operations. Only in this case, it tells the client which one of the files scanned in the images folder should be registered with the server. In all the other cases, an ID = n tells the client to request the operation on the n^{th} image that it has registered with the server.
When a script is requested at the client, the client will conveniently report how it has understood the script. For instance, when using the script:

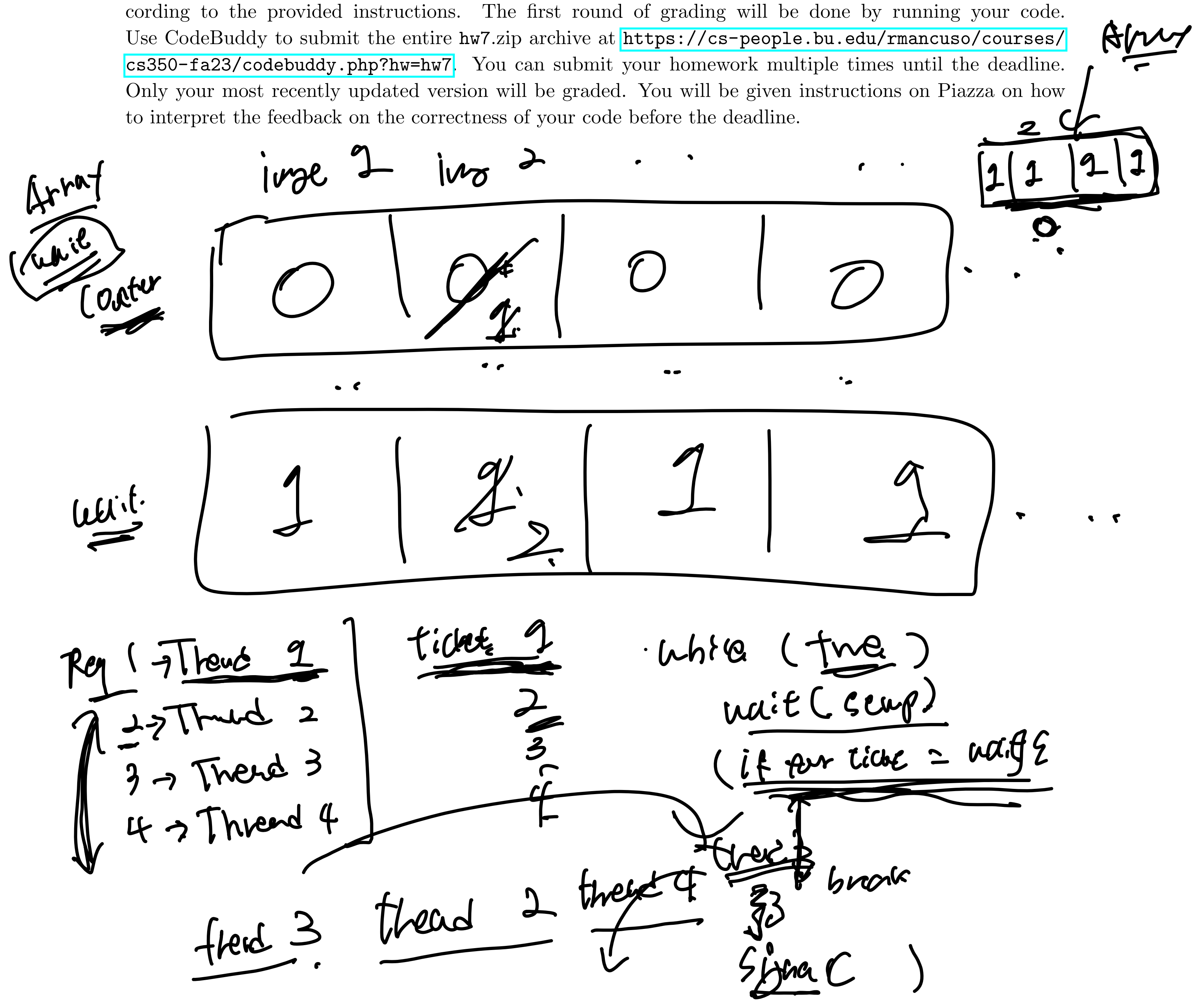
```
1:R:1:2,2:b:1:0,0:T:1:0
```

the client will report:

```
[#CLIENT#] INFO: Reading BMP 0: test1.bmp | HASH = 9f3363f0249c75163d52e60fd9544c31
[#CLIENT#] INFO: Reading BMP 1: test2.bmp | HASH = b6770726558da9722136ce84f12bfac8
[#CLIENT#] INFO: Reading BMP 2: test3.bmp | HASH = f2ac174476fb2be614e8ab1ae10e82f0
[#CLIENT#] INFO: Reading BMP 3: test4.bmp | HASH = 0caaef67ac[illegible]2bd85f25
[#CLIENT#] INFO: Reading BMP 4: test5.bmp | HASH = 5597b44[illegible]1292d[illegible]1c86a3380
[#CLIENT#] INFO: Reading BMP 5: test6.bmp | HASH = 11552ac97535bd4433891b63ed1dd45d
[#CLIENT#] Next Req.: +1.000000000 - OP = IMG_REGISTER, OW = 1, ID = 2
[#CLIENT#] Next Req.: +2.000000000 - OP = IMG_BLUR, OW = 1, ID = 0
[#CLIENT#] Next Req.: +0.000000000 - OP = IMG_RETRIEVE, OW = 1, ID = 0
```



Submission Instructions: in order to submit the code produced as part of the solution for this homework assignment, please follow the instructions below.
You should submit your solution in the form of C source code. To submit your code, place all the .c and .h files inside a compressed folder named hw7.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 hw7.zip archive at https://cs-people.bu.edu/rmancuso/courses/cs350-fa23/codebuddy.php?hw=hw7. 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.



Array
img 1 img 2
Array
wait
Counter
0 | 1 | 0 | 0
wait
1 | 2 | 1 | 1
Req 1 → Thread 1
2 → Thread 2
3 → Thread 3
4 → Thread 4
ticket 1
2
3
4
while (true)
wait(semp)
(if our ticket = wait{
thread
break
}}
signal)
thread 3 thread 2 thread 4