CS-350 - Fundamentals of Computing Systems
Homework Assignment #6 - BUILD

Due on November 8, 2023 — Late deadline: November 9, 2023 EoD at 11:59 pm

Prof. Renato Mancuso

Renato Mancuso

BUILD Problem 1

In all the assignments we have seen so far, the workload submitted by the client has been essentially fake. The client just asks the server to busywait for a given amount of time. That is great to understand how metrics such as response time and utilization evolve in a controlled environment, but ultimately that is not what workload looks like in the real world! With this assignment, we jump into a more realistic workload model where runtimes are unknowns and where inter-task coordination (beyond simple queue sharing) becomes important.

Output File: `server_img.c`

Overview. Of course, we are not going to scrap everything we have build together so far. On the contrary, we are building on top of it once again. With this assignment, we are turning our server into a image processing server! In a nutshell, the server will be able to accept requests from a client that can carry a full payload (raster images), perform operations over said images, and then return back to the client the result of said operations. The server is still designed to be multi-threaded (although for this assignment we will focus on getting only a single thread to do the right thing) with a shared queue, but for simplicity only FIFO request dispatching will be supported in the server—for now, at least.

Design. Please read carefully this section about the design of the server and familiarize yourself with the new `common.h` files and with the new libraries that have been provided to complete this assignment, namely the `imglib` and `md5sum` libraries (header files + implementation).
As previously mentioned, the client will now send requests to the server that contains operations to be performed over raster images. A raster image is just a collection of `<width>`×`<height>` pixels, where each pixel is a 32-bit integer encoding the Red-Green-Blue (RGB) values of each pixel. Take a look at the definition of the `struct image` data type defined in `imglib.h`.
Thus, the new structure of a request will be the following: (1) a unique 64-bit integer encoding the request ID (`req_id`); (2) a `timespec` field encoding the timestamp at which the request was sent by the client (`req_timestamp`); (3) a 8-bit opcode of an operation to be carried out in the server (`img_op`); (4) a 8-bit value set to 0 or 1 to indicate whether the operation requested by the client should overwrite or not the original image (`overwrite`); (5) a 64-bit image identifier (`img_id`). These changes to the `struct request` data type have already been made for you in the `common.h` file.
The design at a high-level is not too different than before. Your server will accept request from a client, and add them to a queue (or reject them if the queue is already full). Worker threads will then dequeue requests, decode what operation to perform based on the content of the request, and send back a result to the client. If you have written your code using the template provided for you in the past assignments, no changes to the FIFO queue should be needed to correctly operate with the new request structure. Most of the new logic will be placed in the worker thread to correctly process the requests.
For this assignment, the emphasis will be on a correct **single-threaded** implementation of the image server. But as we will see, images to be processed by the server will need to be stored in the server memory and be accessible based on their ID by the various threads. Thus, the advise is that you start planning for the case in which the code will be extended to *correctly* support concurrent operation on the stored images.

Image Operations. Of course, the main question is *what are the operations supported by the server and that can be requested by the client?* Here is a list with the necessary details.
(1) **Image Registration:** Opcode: `IMG_REGISTER`. The server initially has no images stored in memory. Thus, one of the first things that the client will do is to register a new image with the server. To do so, the `img_op` will be set by the client to `IMG_REGISTER`, while the values of `overwrite` and `img_id` can be ignored by the server.

Problem 1 continued on next page. . .

The client will **immediately** follow that request with the serialized content of an image. A worker thread must use the provided `struct image * recvImage(int conn_socket)` function to correctly receive an image on the same connection socket used to receive the client requests. The server must then generate an ID for the image.

IMPORTANT: Because of this, the recommendation is to handle this type of request and its response directly in the parent thread, thus bypassing the queue. The side effect is that operations of type `IMG_REGISTER` might not be processed in FIFO order, but that is allowed here.

Server Response. After fully receiving the image payload, the server must reply to the client with the following information: (A) the same 64-bit request ID (not image ID!!) that was sent in the client's request using the `req_id` field; (B) the 64-bit image ID created by the server to identify the newly registered image via the response's `img_id` field; and (C) an acknowledgment field (`ack`) set to 0 or 1 if the request is accepted or rejected, respectively.

(2) **Image Rotation:** Opcode: `IMG_ROT90CLKW`. When the client sets `img_op` to this opcode, the client will request a rotation by 90 degrees clockwise of the image ID specified by the client in the `img_id` field of the request. If the `overwrite` field in the request is set to 0, the operation should be performed on a copy of the original image (with a newly generated image ID); if it is set to 1, the original image should be modified instead. The server can use the `rotate90Clockwise(...)` function of the `imglib` to perform the operation.

Server Response. The server will respond to the client with the following information: (A) the same 64-bit request ID (not image ID!!) that was sent by the client in its request (`req_id` field). (B) In the `img_id` field: (i) if `overwrite = 1`, the *same* 64-bit image ID used by the client in its request, otherwise (ii) the image ID created by the server to identify the copy of the original image on which the operation was performed. (C) An acknowledgment field (`ack`) set to 0 or 1 if the request is accepted or rejected due to a full queue, respectively. IMPORTANT: the server will NOT send the actual payload of the processed image back to the client at this stage.

(3) **Image Blur:** Opcode: `IMG_BLUR`. When the client sets `img_op` to this opcode, the client will request a blur of the image ID specified by the client in the `img_id` field of the request. The use of the `overwrite` field is identical to the `IMG_ROT90CLKW` opcode case. The server can use the `blurImage(...)` function of the `imglib` to perform the operation.

Server Response. Same as in the `IMG_ROT90CLKW` opcode case.

(4) **Image Sharpen:** Opcode: `IMG_SHARPEN`. When the client sets `img_op` to this opcode, the client will request sharpening of the image ID specified by the client in the `img_id` field of the request. The use of the `overwrite` field is identical to the `IMG_ROT90CLKW` opcode case. The server can use the `sharpenImage(...)` function of the `imglib` to perform the operation.

Server Response. Same as in the `IMG_ROT90CLKW` opcode case.

(5) **Detect Vertical Edges:** Opcode: `IMG_VERTEDGES`. When the client sets `img_op` to this opcode, the client will request vertical edge detection for the image ID specified by the client in the `img_id` field of the request. The use of the `overwrite` field is identical to the `IMG_ROT90CLKW` opcode case. The server can use the `detectVerticalEdges(...)` function of the `imglib` to perform the operation.

Server Response. Same as in the `IMG_ROT90CLKW` opcode case.

(6) **Detect Horizontal Edges:** Opcode: `IMG_HORIZEDGES`. When the client sets `img_op` to this opcode, the client will request horizontal edge detection for the image ID specified by the client in the `img_id` field of the request. The use of the `overwrite` field is identical to the `IMG_ROT90CLKW` opcode case. The server can use the `detectVerticalEdges(...)` function of the `imglib` to perform the operation.

Server Response. Same as in the `IMG_ROT90CLKW` opcode case.

(7) **Image Retrieval:** Opcode: `IMG_RETRIEVE`. When the client sets `img_op` to this opcode, the client will request the content of an image with the specified ID (via the `img_id` field) to be sent by the server. The `overwrite` field should be ignored by the server.

Server Response. Regardless of whether the server accepts or rejects the request, the server must reply with (A) the same 64-bit `req_id` sent by the client in the corresponding request, (B) the 64-bit `img_id` the request

Problem 1 continued on next page...

refers to (as set by the client), and (C) a 8-bit `ack` field set to 0 for accepted requests, and 1 for rejected requests (due to a full queue).
If the request is rejected, the three fields above are the only information replied by the server. If the request is accepted, the server must immediately send back the content of the image with the corresponding image ID indicated in the `img_id` of the client's request. To do so, the server can use the `sendImage(...)` function included in the `imglib` library.

Desired Output. Apart from spawning worker threads, processing, and rejecting image processing requests, and ensuring that the requests are added/picked form the queue following a FIFO policy, **three** pieces of information will need to be produced in output by your server. These are similar to what requested in the previous assignments and described below.
The main difference is that, instead of the request length, the values of the `img_op`, `overwrite`, and `img_id` fields set by the client are printed instead. Moreover, the `img_id` field in the server response is also reported in the output.
Queue status dumps and rejection notice are identical in format to HW5. Once again, the queue dump status is printed when *any* of the worker threads completes processing of any of the requests. Just like in HW5, do not print the queue status after a rejection, but only after the completion of a request.
Second, just like HW5, 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. 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. It seems that your server will do a lot of work on images, but how can you visualize the result of these operations?
Fortunately, the `imglib` includes a handy function called `saveBMP(...)` where you can pass a pointer to a `struct image` to turn into a file and a file name. If successful, the function will create a file in Bitmap (BMP) format that can be opened with your favorite image visualization tool. Whenever you want it, you can ask your server to output the image it is working on as a file. Careful not to overdo it because BMPs take a lot of disk space! Also careful not to overwrite any of the original input images.
For the record, the client uses the function `loadBMP(...)` under the hood to load the Bitmap files from disk into memory to be sent to the server.
Also, in its final report, the client provides the MD5 hash for all the images it has sent and retrieved from the server.

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 `hw6`.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 `hw6`.zip archive at `https://cs-people.bu.edu/rmancuso/courses/cs350-fa23/codebuddy.php?hw=hw6`. 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.