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

Due on October 3, 2024 — Late deadline: October 5, 2024 EoD at 11:59 pm

Prof. Renato Mancuso

Renato Mancuso

BUILD Problem 1

In this BUILD problem, you will refine your server's capabilities to manage incoming requests while safeguarding its internal resources when facing high traffic. More specifically, the server will be able to reject requests when its internal queue will grow beyond a fixed limit.

Output File: `server_lim.c`

Overview. With the server design you used in BUILD assignment #2, adding a request to a queue that is full results in an error that has undefined behavior. Fortunately for you, I never tested your code with parameters that lead to that error condition. In this problem, we will extend the server to correctly handle queue overflows via explicit rejections. This is crucial for any server that wants to preserve its ability to remain operational when faced with a surge in incoming requests!
The idea is simple: if a request arrives at a queue that is full, the server should reject the request and reply to the client immediately, telling the client that the request has been rejected.
Design. The design will be as follows. The 8-bit acknowledge field in response is being used now! The acknowledge field, say `ack`, will be set to 1 for rejected requests, and 0 for accepted (and completed) requests. The other two fields, i.e. the request ID and reserved, shall follow the same requirements as specified by previous assignments.
Next, the logic of the server will be as follows. Upon receiving a new request right after a successful `recv(...)` call, the server should try to enqueue the request. If the operation fails due to a full queue, the server should immediately reply to the client that the request has been rejected using a response where the `ack` field has been set to 1.
It is okay (and encouraged!) to implement the reject handling logic directly inside the parent thread.
Queue Sizing. In order to decide what size the queue should have, your code must be able to accept a new parameter from the command line. The new parameter will be passed to your code from the command line as `-q <size>` where `<size>` is a positive integer representing the maximum number of requests that can be held in the queue.
Overall your server will be launched from command line with the following parameters:
`./server_lim -q <size> <port_number>`, where `<port_number>`, just like before, is the port on which the server should bind its socket.
There are many ways to parse the input parameters to a program. The quick & dirty way consists in looping over the `char ** argv` vector of parameters and checking that the string `"-q"` is found in position i. In which case, the string in position $(i + 1)$ will be the `<size>` value to parse and use to initialize the queue size.
A better and more robust approach used by good system programmers is to use the `getopt(...)` call. The manpages will tell you all about it! But it suffices to say that in this case you should set the `optstring` parameter to `"q:"` and then check that the return value of `getopt(...)` is equal to the character 'q'. If the check passes, the string to parse for the size of the queue will be pointed by the `optarg` global variable defined for you by the library.
Desired Behavior. Apart from processing and rejecting requests **three** pieces of information will need to be produced in output by your server.

1. Like before, whenever the worker thread completes processing of a request, it will have to print the report in the format below, which is identical to what you printed in Part 2 of `hw2`. This is printed on its own line.

```
R<request ID>:<sent timestamp>,<request length>,<receipt timestamp>,<start timestamp>,<completion timestamp>
```

Here, `<request ID>` is the ID of the request as sent by the client; `<sent timestamp>` is the timestamp at which the request was sent by the client; `<request length>` is the length of the request as sent by the client; `<receipt timestamp>` is the timestamp at which the parent process received the request; `<start timestamp>`

Problem 1 continued on next page...

is the timestamp at which the worker thread de-queued the request and started processing it; and `<completion timestamp>` is the timestamp at which the worker thread completed processing of the request and sent a response back to the client. All the timestamps should be expressed in seconds with decimal digits with enough precision to capture microseconds.

2. If a request is rejected, the server should print out the following information:

```
X<request ID>:<sent timestamp>,<request length>,<reject timestamp>
```

Notice the "X" placed before the request ID to indicate that the request was not responded to, but rejected. Moreover, the `<reject timestamp>` should be the timestamp at which the decision to reject the request is taken.

3. Just like before, right after the **completion or rejection** of a given request, but **before** doing anything else (e.g., picking up the next request to process from the queue) the server must print out the current status of the queue, on its own line, according to the following format:

```
Q:[R<request ID>,R<request ID>,...]
```

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 `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-fa24/codebuddy.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.