CS-350 - Fundamentals of Computing Systems
Homework Assignment #2 - BUILD

Due on September 28, 2023 — Late deadline: September 30, 2023 EoD at 11:59 pm

Prof. Renato Mancuso

Renato Mancuso

BUILD Problem 1

In this BUILD problem, you are tasked with improving the design of your server to allow it to spawn a worker thread that will be devoted to request processing. For now, the whole point of this part is to spawn a thread and keep it alive.

Output File: `server_mt.c`

Overview. With the server design you used in BUILD assignment #1, once the server is awaiting a packet with a `recv(...)` operation, there is nothing else that the server can do. Indeed, the server is *blocked* waiting for new requests. And it also means that during the busywait loop that corresponds to the processing of a request, the server is unable to promptly receive new packets.
In order to solve both problems, design a (simple) multi-threaded implementation of your server!
Design. The design will be as follows. First recall the general structure of your current single-threaded server. The process performs the following series of initialization operations: (1) a `socket(...)` call; (2) a `bind(...)` call; (3) a `listen(...)` call; an (4) `accept(...)` call. If the `accept(...)` call is successful, we know that a connection with a client has been established successfully.
After that, the server runs the `handle_connection(...)` function that is basically a loop of `recv(...)` calls. Whenever a `recv(...)` returns, we have a valid packet (or an error that terminates the connect). We will now redesign the behavior of the `handle_connection(...)` function.
The new logic will be as follows. Right before starting the `recv(...)` loop (just do not do that inside the loop!), we will start a new *thread* that will be dedicated to processing requests. This is typically called a "worker" thread. We will refer to the original process that creates the thread as the *parent* process, and to the worker thread as the *child* process[1].
Thread Creation. In order to create and start the worker thread in the parent process, use the `clone(...)` system call as described in class. (1) The first parameter will have to be the name of a function, say `worker_main` (i.e., a function pointer) written by you that return an `int` and takes a `void *` as the only parameter. This function will serve as the "main" function of the child thread. (2) The second parameter to the `clone(...)` will be a piece of memory that the child will use as its stack memory. You can pass any piece of memory you allocate for the purpose. Just make sure it is larger than 4 KB. *NOTE:* because the stack grows upward, say you have allocated 4 KB starting at address `void * child_stack`, what you need to pass to the `clone` function is a pointer to the **bottom** of the stack, i.e. `child_stack + 4096`. (3) Set the third parameter (`flags`) to `(CLONE_THREAD | CLONE_VM | CLONE_SIGHAND | CLONE_FS | CLONE_FILES | CLONE_SYSVSEM)` (see man pages to understand why). (4) Finally, pass as the fourth argument any pointer that you want to pass to the worker thread, casted to `(void *)`. What you pass here will be passed to the worker thread via the parameter of the `worker_main` function.
Desired Behavior. For this part, simply define the behavior of the `worker_main` as follow. Upon spawning, the worker thread will output the string:
`[#WORKER#] <timestamp> Worker Thread Alive!`
Where `<timestamp>` is the current time taken using `clock_gettime(...)` in seconds with enough decimal points to capture microseconds.
Next, it will enter a forever loop that only does three things in each iteration: (1) busywait for exactly 1 second; (2) print out the following message:
`[#WORKER#] <timestamp> Still Alive!`
and finally (3) sleep for exactly 1 second. Then rinse and repeat. That's it!
Other than the creation of the worker thread, the parent will still perform all the operation that it was performing in `hw1`, that is get requests and busywait for the requested lenght, and output a full report of the incoming requests, their timestamps and length. Again, with the same exact format as before.

[1] Funnily enough, this is actually the official terminology used to refer to the relationship between processes/threads!

BUILD Problem 2

In this BUILD problem, you will enable your server to perform its own queue management! This includes taking snapshots of the length of the queue of pending packets, and also (in future assignments) to reorder the items in the queue of pending requests. Queue management is a BIG deal because it enables all the fancy queue management strategies that enable modern system to be smart about how they process incoming requests.

Output File: `server_q.c`

Overview. We will now delegate request handling to the worker thread instead of the parent thread. The goal is to have the following structure.
Structure. The parent process will only perform `recv(...)` calls, while the worker thread will NOT do any useless 1-second sleep or busywait like in Part , and also get rid of any print where the worker declares to be alive. Instead, the worker thread will process the requests sent by the client by busywait'ing for the requested amount of time. Because requests are `recv`'d in the parent, any packet correctly `recv`'d by the parent will be put in a **queue** located in memory. The child worker thread will then pop/remove requests from the queue and handle them by performing a busywait of the appropriate length just like in `hw1`. You are free to use your own favorite data structure to implement the shared request queue.
Queue Sharing. Because regardless of how you implement the queue, it will be shared between parent and child, we need to protect the queue against corruption happening due to unrestricted simultaneous activity of parent and child processes. You will learn all about how to do this a bit later into the semester, so, for now, two *template* functions to handle queue addition and removal are provided: (1) `int add_to_queue(struct request, ...)` and (2) `struct request get_from_queue(...)`. Because these are just template functions, you will have to implement their internals. But make sure to follow the comments I left for you in the code to know which parts NOT to touch.
The function `int add_to_queue(struct request, ...)` should be used in the parent. It takes a request and some other parameters of your choice to add a request to the shared queue. The suggested semantics of the return value is to return 0 in case of successful insert, and -1 in case of error (e.g., if the queue is full).
The function `struct request get_from_queue(...)` should be used in the child to retrieve the next element from the queue. You might want to return `NULL` if the queue is empty, but that should never happen because the provided code in the template of the two functions makes sure that the child is blocked if there is nothing in the queue.
Desired Behavior. Apart from processing the requests, the worker thread will print out **two** pieces of information.

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 similar to *but a bit different* than what you printed in `hw1`. 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>` 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.

Problem 2 continued on next page...

2. Right after starting to process of a given request, but **before** picking up the next request to process (if any) from the queue, the client 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>,...]
```

Here, `<request ID>` is the ID of the request as sent by the client and queued for later processing. So for instance, say that we receive only two requests, both of length 10 and at times 0 and 1, respectively. The full output will look something like this (assuming no overheads):

```
R0:0.000000,10.000000,0.000000,0.000000,10.000000
Q:[R1]
R1:1.000000,10.000000,1.000000,10.000000,20.000000
Q:[]
```

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 `hw2`.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 `hw2`.zip archive at `https://cs-people.bu.edu/rmancuso/courses/cs350-fa23/codebuddy.php?hw=hw2`. 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.