CS-350 - Fundamentals of Computing Systems
Homework Assignment #5 - BUILD

Due on October 24, 2024 — Late deadline: October 25, 2024 EoD at 11:59 pm

Prof. Renato Mancuso

Renato Mancuso

BUILD Problem 1

So what is the point of scheduling after all? With this BUILD assignment, we will start to discover the importance and benefit of scheduling resources by empowering the server to change its request dispatch policy. But we will also the complexity associated with implementing job-parameter-aware scheduling policies.

Output File: `server_pol.c`

Overview. You guessed it: we will be once again building on top of what implemented so far. This time around, we will enable our server to have a configurable queue dispatch discipline. The server will still support FIFO, but now it will also give the option to use the *shortest request next* discipline, or, as called in the scheduling world, Shortest Job Next (SJN).

Design. The design is pretty much what you already guessed, but you have a number of concrete options for how things will be implemented. The entire design of the server can stay as is and the recommendation is to include the changes that implement the selection of SJN vs. FIFO policy inside the queue helper functions. The end goal is that under FIFO policy, nothing should change in the operation of the queue compared to HW4. When SJN is selected, however, you should make sure that the next `get_from_queue(...)` call dequeues and returns the request with the shortest length.

To do this, there are two main options (1) add the logic to the `add_to_queue(...)` function to perform a sorted insert; or (2) add the logic to `get_from_queue(...)` to select the request with the shortest length upon dequeuing. There are two ways to go about this. The first is to add the new request at the end of the queue as you did with the FIFO policy, and then run a sorting algorithm on the entire queue. Alternatively, you can find the place in the queue where to add the new request, insert it there, and then shift all the longer requests by one spot. Whatever you do, be careful about the queue wrap-around!

Queue Policy. Just like in HW4, the number of worker threads to activate as well as the size of the queue will passed as a command line parameter to your code. A new parameter for the queue policy must be accepted by your server. The new parameter will be `-p <policy>` where `<policy>` is either the string "FIFO" or the string "SJN".

Overall your server will be launched from command line with the following parameters:

`./server -q <size> -w <workers> -p <policy> <port_number>`, where `<size>` is a positive integer representing the maximum number of requests that can be held in the queue; `<workers>` is a positive, non-zero integer representing the number of worker threads to spawn;and `<port_number>`, just like before, is the port on which the server should bind its socket.

Desired Behavior. Apart from spawning worker threads, processing, and rejecting requests, and ensuring that the requests are added/picked form the queue following the specified policy, **three** pieces of information will need to be produced in output by your server. These are identical to what requested in HW4, but are described below again for simplicity.

First, queue status dumps and rejection notice are identical in format to HW3 and HW4. Once again, the queue dump status is printed when *any* of the worker threads completes processing of any of the requests. Just like in HW3 and HW4, do not print the queue status after a rejection, but only after the completion of a request.

Second, just like HW4, 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.

```
T<thread ID> R<request ID>:<sent time>,<req. length>,<receipt time>,<start time>,<completion time>
```

Here, `<thread ID>` is the ID of the worker thread that completed the given request. Once again, these IDs must go from 0 to `<workers> - 1`. If multiple worker threads are available to process a pending request, any one of them (but only at most one!) can begin processing the request.

NOTE: because with multiple threads the prints can get messy and interleave badly, once again use the synchronized wrapper of the `printf(...)` that I provide in the template file for this assignment. The

Problem 1 continued on next page...

function is called `sync_printf(...)` and it is identical in terms of interface to the traditional `printf(...)` call. Make sure to include the `printf_mutex` initialization code that I provide in the `main(...)` function in the template files.

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 `hw5`.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 `hw5`.zip archive at `https://cs-people.bu.edu/rmancuso/courses/cs350-fa24/codebuddy.php?hw=hw5`. 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.