CS333: Operating Systems Lab

Lab 4: Synchronization

Goal

The goal of this lab is to learn process and thread synchronization. In this lab, you will modify the file server built in Lab 2, to make it multi-threaded. The earlier server spawned a new process for every new request, which is not the most optimal way to design servers, given the extra overhead incurred for process creation. In the lab, you will change the server to create a pool of a fixed number of threads, and assign incoming requests to free threads on the fly. This server implementation will require you to use locks and conditional variables to synchronize between the various threads.

Before you start

- Understand locking and conditional variables in the `pthreads` API. Your code will heavily rely on locks and conditional variables.
- Review the client-server protocol and test setup from Lab 2. We will only be modifying the server in this lab; you should be able to reuse the client and the test setup from the previous labs.

Multi-threaded server with a fixed thread pool

In this lab, we will write a new server, `server-mt.c`. This server provides the same interface to the client as your `server-mp` from Lab 2. However, the internal architecture of the server is different in this lab. The server spawns a specified number of worker threads at the beginning of its execution. For every new client that connects to the server, the server runs `accept`, obtains the new socket file descriptor corresponding to this new client, and stores this new socket file descriptor in a request queue. The worker threads dequeue these sockets from the request queue, read the command sent by the user from the socket, serve the specified file to the user, close the socket, and go back to fetching another request from the queue. The worker threads sleep on a conditional variable if the request queue is empty, and are woken up by the main process once requests come in.

Much like the older server, this server also takes the port number to listen on as the commandline argument, and listens for incoming file get requests from clients on a TCP socket at this port. In addition, `server-mt` also takes the number of worker threads and the request queue size as additional arguments. If the request queue size is specified as zero, the server maintains an unbounded queue of requests. For a non-zero value of the request queue, the server maintains a bounded queue of the specified size. If the number of requests in the queue exceeds this limit, the server process sleeps on a conditional variable and stops accepting new client connections. The server wakes up and starts accepting new connections once some requests are dequeued by the worker threads.

A sample run of `server-mt` is shown below. The following command starts a multi-threaded server listening on port 5000. The server process spawns 10 worker threads, and maintains a queue of at most 500 pending requests.

```
$./server-mt 5000 10 500
```

Exercises

Much like in Lab 2, please check your code thoroughly for correctness. Check with a simple client (say, the shell-based client of Lab 4) to see that your server is serving files correctly. Next, check with multiple clients at a time, and check that the clients are indeed downloading all the files correctly (say, by dumping the file contents somewhere). Also make sure that your code doesn't leak any memory, and doesn't cause segmentation faults when run with a reasonably large number of clients.

Once you have confidence in the correctness of your code, do the following exercises, and record your observations in your report. For all these exercises, please use the same setup as used in Lab 2. Use the `multi-client` program from Lab 2 as your client. Run all experiments for 120 seconds, and always use zero think time at the clients. All clients will request one of the 10,000 `fooK.txt` files (either the same file, or a random file), from the file repository you created for the previous labs. Also, remember to clear the disk buffer cache between successive experiments, much like in Lab 2.

1. Simulate 20 concurrent users in `multi-client`, all accessing the same fixed file from the server (so that the file is mostly served from the disk buffer cache in memory, and not from the disk). Vary the number of worker threads N at the server from 1 through 10. Use an unbounded request queue (i.e., set the queue size to 0 in the commandline argument). Observe the average throughput (average number of file get requests served per second, across all client threads) reported by `multi-client` at the end of the experiment. What is the minimum value of N that gives you the maximum server throughput? In other words, what is the minimum number of worker threads at the server required to fully saturate the server?

2. Repeat exercise 1 above, but use a very large number of users (1000 concurrent users) at the client, and a very small request queue size (say, a queue size of 1) at the server. Now, run your server a few times for varying values of the number of worker threads N. Explain what happens when you run the experiments. In particular, do you see all client requests being successfully served, or are some client requests denied service at the server?

Submission and Grading

You may solve this assignment in groups of one or two students. You must submit a tar gzipped file, whose filename is a string of the roll numbers of your group members separated by an underscore. For example, `rollnumber1_rollnumber2.tgz`. The tar file should contain the following:

- `report.pdf`, which contains your answers to the exercises above. Be clear and concise in your writing.
- Your code `server-mt.c`. Please make sure your code is well documented and readable.
- An optional makefile to build your code.

- `readme.txt` describing your experimental setup, instructions on how to run your code, and sample resuts from test runs.

Evaluation of this lab will be as follows.

- We will run your code and check that it serves files correctly, by running it against our client.
- We will read your code and check for correctness of your implementation.
- We will read through your answers to the exercises to make sure they are consistent with your code and experimental setup.