Programming Assignment 5: High Concurrency without too many Threads

Due: 11/11/18 at 11:59pm

Introduction

In this programming assignment, we try to further improve the performance of the client by reducing the thread management overhead required to handle the worker threads. We do this by replacing the collection of worker threads by a single event handler: Instead of creating a large number of worker threads and having each handle a separate request channel, in this programming assignment we have a single event handler thread manage all the channels for data communication with the data server. (The communication over the control channel is still handled by the main thread of the client). You are to improve on the client program from PA4 as follows:

Instead of spawning multiple worker threads, and have each thread separately communicate to the data server, spawn a single event handler thread, which handles all data request channels.

Please use your PA4 as the starter code.

The Assignment

You are to write a program (call it client.cpp) that first forks of a process, then loads the provided data server, and finally sends a series of requests to the data server. The client should consist of a number of request threads, one of reach person, one event handler thread, and a number of statistics threads, one for each person. The number of persons is fixed to three in this PA (Joe Smith, Jane Smith, and John Doe). The number of data requests per person is to be passed as arguments to the invocation of the client program. As explained earlier, the request threads generate the requests and deposit them into a bounded buffer. The size of this buffer is passed as an argument to the client program. The client program is to be called in the following form:

```
./client -n <requests/person> -b <bounded buffer size> -w <number of request
   channels>
```

A few Points

A few points to think about:

- The magic to have a single event handler thread manage multiple request channels is to use the `select()` system call. We would decouple the `RequestChannel::cwrite()` and `cread()` pair. Instead, the we would send `cwrite()` on all channels and then try to do `cread()` on them. But, each channel would take a random time to return with some data from the server (i.e., because the server puts a random delay before each). Therefore, we need to read the data in the "correct" sequence and the `select()` function will help us with that.

The `select()` call monitors multiple file descriptors and returns to indicate the file descriptor(s) that show activity. In this way you can have a single thread handle multiple file descriptors, i.e. multiple request channels. This is different from PA4, where we had a separate thread for each request channel.

- Have either the main thread or the event handler thread create the request channels before the event handler thread starts issuing select calls.
- Since the select call uses file descriptors, we have to make the file descriptors used to read and write data to the request channel accessible to the user. The class RequestChannel now provides two functions (`read_fs()` and `write_fs()` that return the read and write file descriptor of the request channel, respectively. These file descriptors can be used to monitor activity on the request channels. If activity has been detected on the read file descriptor, only then your code may read the data by calling `RequestChannel::cread()`. In addition, the next request from the request buffer can be sent to the request channel using `RequestChannel::cwrite()`. Note that, in PA4, we could just issue the `cread()` without considering all these and wait for the data to arrive. This was alright, because even with one thread stuck in a `cread()`, other threads would make progress.

What to Hand In

- Submit the solution directory containing all files and also a makefile.
- Analyze the performance of your implementation in a report, called report.pdf. Measure the performance of the system with varying numbers request channels and sizes of the buffer. How does the performance compare to your implementation in PA4? Does increasing the number of request channels still improve the performance? If so, by how much? Is there a point at which increasing the request channels does not further improve performance? Submit a report that compares the performance to that of your solution in PA4 as a function of varying numbers of request channels (i.e., worker thread in the case of PA4).

Reading Resources

For understanding the `select()` function, read from `http://beej.us/guide/bgnet/`. This will also be useful for the next programming assignment.

Rubric

1. Event handler thread (30 pts)
    - Priming the channels (i.e., sending initial requests): 10 pts

- Re-preparing the `fd_set` during every iteration or using a backup: 10 pts
- Proper termination (showing histogram at the end): 10 pts

2. BoundedBuffer (10 pts)
3. Not having global variables (5 pts)
4. Handling the case when $w > 3n$ (5 pts)
5. Cleaning up fifo files and all dynamically allocated objects (10 pts)
6. Correct counts in the histogram (20 pts)
7. Report (20 pts)
   - Should show plots of runtime under varying n, b, w. Especially, we want to see variations in b (range $[1, 200]$) and w (range $[1, 500]$) after setting $n = 10$K at least.
   - Compare the shape of the runtime curve with that of PA4. Do you see any difference? Why?