Machine Problem 5: High Concurrency without too many Threads

Due 4/18/16: Total 100 points (plus 20 pts for bonus)

Introduction

In this machine problem 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 forking of a large number of worker threads and having each handle a separate request channel, in this machine problem 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 MP4 as follows:

1. 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.
2. (BONUS 20pts) Have the client periodically show a simple display of the histograms. This is to be implemented by first installing a timer signal handler that displays the histograms and then by periodically generating a timer signal.

Please use the given code of MP4 (MP4\_sources.zip) as the starter code. In addition, you will need to take your BoundedBuffer and Semaphore implementations from MP4 to use in this machine problem.

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 MP (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 <number of data requests per person>

-b <size of bounded buffer in requests>

-w <number of request channels to be handled by event handler thread>

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. 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 MP4, 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, say the read file descriptor, your code may then read the data either by accessing RequestChannel::cread() or by reading directly from the file descriptor returned by RequestChannel::read\_fs(). Similarly, the next request can be sent to the request channel using RequestChannel::cwrite() or by writing to file descriptor speci\_ed by RequestChannel::write\_fs().
* You will quickly notice that you will not be able to use the RequestChannel::send\_request() function, which is basically nothing more than a cwrite() followed by a cread() anyway. The reason for this is because you will have to wait in select() for the file descriptor to become \active" before calling cread().
* Use your Semaphore and BoundedBuffer classes from MP4.

If you decide to go for the BONUS, you will have to be careful about a few points:

* Install a signal handler for the SIGALARM signal. This signal is generated periodically after initializing a timer with a call to setitimer(). The job of the signal handler is to draw the current histograms.
* You will have to make all blocking calls resilient against signals by handling EINTR errors correctly. This includes (i) the creations of the request channels (you may want to avoid this by having the timer start firing only after the channels have been established), (ii) read and write operations to the channels, and (iii) possibly other blocking operations. Note: This may require you to modify the code in RequestChannel.cpp.
* Be aware that you are handling process-wide signals in a threaded environment. You may not know a priori which thread is going to handle the timer signal. In many cases, this is not too much of a problem, but you may still want to be aware of this.

What to Hand In

* You are to hand in a directory, called Solution, with all files that are part of your solution. This directory should contain, among other files, your file client.cpp and the given dataserver.cpp.
* The directory Solution must also contain a working make file, which generates an executable client and an executable dataserver. The functionality of the client is identical to the client in MP4. Compared to MP4, the new client creates a single event handler thread and handles the request channels using the select() system call.
* If you go for the BONUS option, you may need to update the request channel implementation to be resilient against signals. If you decide to go for the BONUS, please specify this clearly in your report, and list what part of the program you changed to make it resilient against signals.
* 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 MP4? 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 MP4 as a function of varying numbers of request channels (i.e., worker thread in the case of MP4).