Machine Problem 4: High Concurrency without too many Threads

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 off a large number of worker threads and have each handle a separate request channel, in this machine problem we have a single *event handler thread* manage all the 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 MP3 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 to and from the request channels.

2. (BONUS) 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.

You will be given the same source code of the data server as in MP3 (in file `dataserver.cpp`) to compile and then to execute as part of your program (i.e. in a separate process).

The Assignment

You are to write a program (call it `client.cpp`) that first forks off 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 are 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 having a single event handler thread manage multiple request channels is to use the `select()` system call. As we discussed in class, 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 MP3, 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 need access to the file descriptors used to read and write data from and to the request channel. The class `RequestChannel` 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 of a channel, your code may then read the data either by accessing `RequestChannel::cread()` of that channel. Similarly, the next request can be sent to the request channel using `RequestChannel::cwrite()`.
- 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 have to split up the `cwrite()` from the `cread()`: You send the data to the server by issuing a `cwrite()`, and then you have to wait in `select()` for the file descriptor to become "active" before calling `cread()`.
- In this MP the client uses `select()` only to read data that comes back from the server. The client sends data to the server directly, i.e. using the `cwrite()` command. The `select()` call therefore only defines the read set, and leaves the write set and the error set to be NULL.
- the structure of the client is very simple, and it consists of three stages:
  1. Create the request channels.
  2. "Prime the channels" by writing one request into each channel.
  3. Keep blocking on the `select()` call, read and handle the response, and write the next request into the "active" channel.
  4. When the event handler receives a "quit" request from the request threads, it can start winding down the request channels: Whenever it gets the response back from a channel, it can close the channel, as there won't be any more requests going out.
- Use your `Semaphore` and `BoundedBuffer` classes from MP3.
- Be careful when you exceed 125 request channels. You may exceed OS limitations on number of open files.

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 againt 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.C`.
- 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.C` and the source code for the dataserver (in file `dataserver.C`).
- The directory `Solution` must also contain a working makefile, which generates an executable `client` and an executable `dataserver`. The functionality of the client is identical to the client in MP3. Compared to MP3, 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 MP3? Does increasing the number of request channels still improve the performance? 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 MP3 as a function of varying numbers of request channels (i.e., worker thread in the case of MP3).