Machine Problem 3: Dealing with a Reluctant Data Server

Introduction

In this machine problem we continue our work with the data server. Here the data server is slightly more realistic, and this in three ways:

1. Before generating a reply to a data request, the data server has to go and look up the requested data. This causes a delay in responding to a data request. This delay is in the order of a few milliseconds.

2. Instead of simply echoing back the content of the data request, the data server now returns some actual data. For this MP, the data returned to a `data` request is a number between 0 and 99. Specifically, a request of the type `data Joe Smith` will return a number between 0 and 99. The next request for data on Joe Smith will return another number.

3. The data server can now support multiple request channels, which in turn are served concurrently. Whenever the client issues a request `newthread` the server creates a new request channel and returns the name of the new channel to the client. The client then can create the client side of the request channel and start sending requests over the new channel.

You are to improve on the client program from MP2 as follows:

1. Have the client issue multiple requests concurrently to the data server. This is done by spawning multiple *worker threads*, and have each thread separately issue requests to the data server.

2. Have the client maintain a histogram of returned values for each person. (We have three persons: Joe Smith, Jane Smith, and John Doe.) Each person has a *statistics thread* associated, and worker threads deposit their new value with the appropriate statistics thread. The statistics thread keeps the histogram updated as new data comes in.

3. Collect, say 10000 data values for each person. In order to keep the number of worker threads independent from the number of persons, have the data requests originate from so called *request threads*, one for each person. These should be deposited into a bounded buffer, and the worker threads consume these requests. Whenever the buffer is empty, the worker threads wait.

You will be given the source code of the data server (in file `dataserver.C`) to compile and then to execute as part of your program (i.e. in a separate process). This program handles three types of incoming requests:

`hello`: The data server will respond with `hello to you too`.

`data <name of item>`: The data server will respond with data about the given item.

`quit` : The data server will respond with `bye` and terminate. All IPC mechanisms associated with established channels will be cleaned up.

`newthread`: The data server creates a new request channel and returns the name of the request channel back to the client. The client can then create the client-side of the channel and start sending requests through this channel, preferably in a separate thread.

The Assignment

You are to write a program (call it `client.C`) that first forks off a process, then loads the provided data server (similarly to MP2,) and finally sends a series of *data requests* to the data server. The client should have three types of cooperating threads:

- **Request Threads:** Each person should have one request thread, which issues a sequence of data requests.
- **Worker Threads:** In addition, the client has a number of worker threads, which are independent from the request threads. The job of the worker threads is to grab the data requests from the request threads and forward them to the data server. We separate request threads from worker threads because we do not want the number of requests threads define the level of concurrency in the system. (If we have a million patients, say, this would completely overwhelm the server if each request thread were to set up a separate connection to the data server.) By having a separate set of worker threads, we control the number of connections to the data server and the level of concurrency, independently of the number of request threads.
- **Statistics Threads:** For each person we have one statistics thread. The role of the statistic thread is to collect and visualize the data returned from the data server for the given 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 and the number of worker threads are to be passed as arguments to the invocation of the client program. The request threads generate the requests and deposit them into a bounded buffer, and the worker threads remove requests from this buffer to forward to the data server. The size of this buffer is passed as an argument to the client program. When the replies come back, they are forwarded to the appropriate statistics thread. The best way to forward these replies is again by depositing them into bounded buffers, in this case one per statistics thread.
The client program is to be called in the following form:

```
client -n <number of data requests per person>
       -b <size of bounded buffer between request and worker threads>
       -w <number of worker threads>
```

The Solution Design

We will be discussing a more detailed specification and a solution approach in the lab. The solution will make heavy use of bounded buffers. We will first focus on the architecture of the solution, then figure out how to start up the various threads. When this works, we will think about how to cleanly terminate all threads when there are no more requests.

We will pay attention to come up with a solution that is resilient to future changes to the underlying architecture. For example, the client may be partitioned to run on two or more machines, with a visualization hosts rendering the collected data, and multiple client hosts sending requests to the data server.

You are free in how you want to specify the interface for your bounded buffer class. Some of you may want to have a generic bounded buffer using templates. Other may prefer to stay away from generics.

A few Points

A few points/hints to think about:

- While you develop/debug your client, don't fork off the server process yet. It is easier to debug the client by having the client run in a different window than the server. In this way, the outputs from the two processes are easier to read, as they don't interleave. Only after you convinced yourself that things are working well, have the client fork off the server as you learned from MP2.
- Make sure that your semaphore from MP2 works. You will be using it to implement a bounded buffer class, which you then can use to synchronize and pass data between request threads and the worker threads and between the statistics threads and the worker threads.
- Make sure that you clean up everything correctly at the end. When the work is done, worker threads (or somebody on their behalf) have to first send a `quit` request to the data server and then close the request channels. After all the worker threads are done, the main thread will have to send a `quit` request to the data server and then close the request channel and quit.
- Depending on the configuration of your system, you may be limited in how many worker threads you can create before you run into difficulties. One limitation is the number of open file descriptors that you can have. The Request Channels between the client and the server each require two open-file descriptors. If you are limited to 256 open files max, which is often the case, you can create only up to about 126 worker threads (2 * 126 file descriptors for each thread, plus 3 file descriptors for the process, which makes for a grand total of 255 file descriptors in this case) before you get an error. See how many worker threads you can create!

What to Hand In

- You are to hand in one file ZIP file, with called `Submission.zip`, which contains a directory called `Submission`. This directory contains all needed files for the TA to compile and run your program. This directory should contain **at least** the following files: `client.C`, `dataserver.C`, `reqchannel.H/C`, `semaphore.H/C`, `bounded_buffer.H/C`, and `makefile`.
- The expectation is that the TA, after typing `make`, will get a fully operational program called `client`, which then can be tested.
- Submit a report, called `report.pdf`, which you include as part of the directory `Submission`. In this report you present a brief performance evaluation of your implementation of the client.
- Measure the performance of the system with varying numbers of worker threads and sizes of the buffer. Does increasing the number of worker threads improve the performance? By how much? Is there a point at which increasing the number of worker threads does not further improve performance? Submit a report that compares the performance as a function of varying numbers of worker threads and buffer sizes.