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 two 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, 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, a number of *worker threads*, 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 and the number of worker threads 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 worker threads>
```

A few Points

A few points to think about:

- You are provided a class definition for a `Semaphore`. Implement semaphores to control the bounded buffer between the request threads and the worker threads.
- Similarly, use semaphores to synchronize 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.

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.C`, `reqchannel.H`, and `makefile`.
- Your submission directory should also contain the files `Semaphore.H` and `Semaphore.C`, which implement the semaphore class.
- 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.