Machine Problem 2: First Steps with Client-Server Processing

Introduction

In this machine problem, we set the stage for a high-performance data query and processing system for an ICU patient monitoring system. This system consists of a *client* program that sends data requests to a *data server* program and then processes and "visualizes" the returned data. In this machine problem we find our way around the system, collect some baseline performance data, and sets up some infrastructure for subsequent machine problems.

The client program (to be implemented as program `client.C` is to fork off a process, which is supposed to load the data server program (the source code is provided in file `dataserver.C`. The client then issues requests to the server through what we will be calling *request channels*. Request channels are a simple inter-process communication mechanism, and they are provided by the C++ class `RequestChannel`. Request channels provide the following interface:

```
RequestChannel(const string _name, const Side _side);
/* Creates a "local copy" of the channel specified by the given name.
   If the channel does not exist, the associated IPC mechanisms are
   created. If the channel already exists, this object is associated
   with the channel. The channel has two ends, conveniently called
   "SERVER_SIDE" and "CLIENT_SIDE". If two processes connect through
   a channel, one has to connect on the server side and the other on
   the client side. Otherwise the results are unpredictable. */

~RequestChannel();
/* Destructor of the local copy of the channel.
   By default, the Server Side deletes any IPC mechanisms associated
   with the channel. */

string send_request(string _request);
/* Send a string over the channel and wait for a reply. This function
   returns the reply to the caller. */

string cread();
/* Blocking read of data from the channel. Returns a string of characters
   read from the channel. Returns NULL if read failed.
   THIS FUNCTION IS LOW-LEVEL AND IS TYPICALLY NOT USED BY THE CLIENT. */

int cwrite(string msg);
/* Write the data to the channel. The function returns the number of
   characters written to the channel.
   THIS FUNCTION IS LOW-LEVEL AND IS TYPICALLY NOT USED BY THE CLIENT. */
```

You will be given a basic framework code, which you will be extending in a series of machine problem. This framework consists of the following files:

`simpleclient.C`: This file contains a simplified but runnable form of the client. You will be using this file and add functionality to it in this and subsequent machine problems.

`dataserver.C`: This file contains the data server, described below.

`reqchannel.H/C`: These files contain the C++ `RequestChannel`.

The data server (in file `dataserver.C`) is to be compiled and then executed as part of your program (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.

The Assignment

This MP consists of several parts:

Part 1: Write a *program* (call it `client.C`, derive it from the given file `simpleclient.C`) that first forks off a process, then loads the provided data server, and finally sends a series of requests to the data server. Before terminating, the client sends a `quit` request and waits for the `bye` response from the server before terminating.

Part 2: Write a brief *report* that compares the overhead of sending a request to a separate process (using the function `send_request()` in the client compared to handling the request in a local function. Measure the invocation delay of a request (i.e. the time between the invocation of a request until the response comes back.) Compare that with the time to generate the data on server (call to function `generate_data`. Submit a report that compares the two and that shows the overhead of generating the data in a separate process.

Part 3: Implement a C++ class `Mutex`, which supports the functions `Lock` and `Unlock`. The mutex shall be implemented using POSIX mutexes. (A file `mutex.H` is supplied.)

Part 4: Implement a C++ class `MutexGuard`, which will have only a constructor and a destructor, and no other functions. The constructor takes an object of class `Mutex` as argument and locks it. The destructor unlocks the mutex. This counter-intuitive class will allow us to define critical sections lexicographically as by taking advantage of block scoping, as shown by the following code:

```
<some code here>
{ auto mg = MutexGuard(m);  // Mutex m has been declared elsewhere
    < This is a critical section protected by mutex m >
} // Here we leave the block, and destructor for mg is called,
  // thus unlocking mutex m
```

We note that if the code leaves the "protected" block for any reason (exception, crash, etc.) the mutex is unlocked. (A file `mutex_guard.H` is supplied.)

Part 5: Implement the C++ class `Semaphore`, which supports the usual semaphore operations. (A file `semaphore.H` is supplied.)

What to Hand In

Detailed hand-in instructions will follow.