CSE 370 Lab 8, Spring 2004

Objectives:

1. Quick and dirty introduction to POSIX threads and synchronization.
2. Gain experience in use of the monitor object design pattern.

Description: The lab for next will will extend the Perlisism browser to use multiple threads of control, so as to minimize the effects of blocking in the user interface. This lab aims to give you some experience in the use of POSIX threads and (elementary) thread synchronization. Copy the files in /user/cse370/Labs/Lab8 to your local directory. The file client.cc contains a small textual program that acts as a client of the Perlisism server (from last week’s lab). The client requests the first 100 Perlisisms, prints each to standard output as received, and then exits. Note that this code uses the class BufferedNetworkStream to encapsulate all of the low-level network communication details. Take a moment to familiarize yourself with client.cc.

This lab involves:

1. Modifying client.cc to use two threads—one to continuously invoke the serviceNetwork operation on the streamManager object, and one to continuously try to retrieve new data from the streamManager object.

2. Modifying class BufferedNetworkStream to make it thread safe by introducing synchronization primitives around each of its methods.

For background on task 1, we refer you to an online tutorial of POSIX threads, specifically sections 2 and 3 in:


which gives a quick intro to thread creation and the use of locks (mutexes) to synchronize multiple threads in order to avoid race conditions. Note: This tutorial references some features of POSIX threads that are not available on our systems (e.g., pthread_delay_np, and pthread_attr_default). For this lab, you will not need to use delays, and you may safely use the null value (0) whenever a situation calls for the use of pthread_attr_default.

For the first task, modify client.cc by:

1. Writing a new function, call it gatherData, that contains an infinite loop whose body has one statement: streamManager.serviceNetwork().
2. Remove the above statement from the function myEventLoop.

3. Modify function main so that it creates a new thread in which to run the function gatherData.

When you’ve finished, show this code to your TA. Note: This program contains a race condition, because the streamManager object might be concurrently accessed and updated by both threads. Consequently, some executions of your program might fail; however, such errors should be intermittent, and we will fix them in the second task.

The second task concerns the protection of data in objects that are accessed concurrently by multiple threads. In general, we want to prevent two separate threads from executing methods on the same object at the same time, as this concurrent execution can corrupt the data in the object and make the methods return invalid results. In the multi-threaded version of client.cc, the streamManager object can be accessed concurrently by both the main thread (which is executing myEventLoop) and the new thread you created (which is executing gatherData). There is a general technique for protecting such shared objects by synchronizing the methods of the class. The key idea is to extend class BufferedNetworkStream with a new private data member of type pthread_mutex_t. Then, at the beginning of every method, you will invoke the function:

```c
pthread_mutex_lock(&mutex);
```

to get the lock, thereby permitting other threads from entering, and at the end of every method, you must then invoke the function:

```c
pthread_mutex_unlock(&mutex);
```

to release this lock. Create a new class called MonitorBufferedNetworkStream, which derives from BufferedNetworkStream, adding the private declaration:

```c
mutable pthread_mutex_t mutex;
```

and over-riding each virtual function as follows. Consider, the function:

```c
unsigned BufferedNetworkStream::size() const
```

Override this function in MonitorBufferedNetworkStream with the method:

```c
unsigned MonitorBufferedNetworkStream::size() const
{
    unsigned sz=0;
```
pthread_mutex_lock(&mutex);
sz = BufferedNetworkStream::size();
pthread_mutex_unlock(&mutex);
return sz;
}

That is, each method in the derived class should get the lock, then invoke the base-class method, then release the lock. When finished, show your solution to the TA.