CSCI 4273/5273
Programming Assignment 3
Fall 2014

Due date: 10/31/2014

Goal: The goal of this programming assignment is to design and implement three system-level libraries that provide support for implementing a network system. This assignment requires familiarity with `pthreads` thread package. Please review pthreads before attempting this assignment.

Grade: 8% of your final grade is allocated for this assignment.

Team: You may work on this assignment in teams of up to two students. You will be using the libraries you build as part of this assignment for your programming assignment 4, which will also be a team assignment.

Good performance is a crucial requirement in the implementation of a computer network. While the old programming adage "first get it right and then make it fast" is valid in many computing software, it is usually necessary to "design for performance" in computer networks. It is therefore important to understand various factors that impact network performance.

Recall that a network system is comprised of several layers with one or more protocols at each layer. There are several common functonalities needed in the implementation of all or most of these protocols. For example, all protocols need to manipulate messages by adding/stripping headers, several protocols need to implement timeout mechanisms, etc. The goal of this assignment is to design and implement a suite of three libraries that support the implementation of these common functionalities. The underlying idea is that you implement these common functionalities carefully and efficiently once, and "efficient" protocol implementation is then simply a matter of using these libraries appropriately. We will design and implement three network libraries:

1. Thread pool library
2. Message Library
3. Event Scheduler Library

Thread Pool Library

A network system is typically comprised of several threads running concurrently. Each thread incurs overhead in terms of thread creation and destruction as well as memory needed for its stack. A thread pool is a set of threads that are created during initialization and remain available to execute a function whenever needed. Our goal is to design and implement a class `ThreadPool` that manages a set of threads. This class provides five (public) member functions (you may add more (private) functions as needed for your implementation):

`ThreadPool(size_t` *threadCount*`)`
`~ThreadPool( )`
`int dispatch_thread(void` *dispatch_function*`(void*), void *`*arg*`)`
`bool thread_avail( )`

The first function is a constructor function that creates a `ThreadPool` object consisting of a set of *threadCount* threads (default value: 10). The second function is the destructor function. The third function dispatches a thread from the thread pool to execute the *dispatch_function*(). After completing the execution of the *dispatch_function*(), the thread returns to the thread pool. The *dispatch_function*() function has one parameter, *arg*. Finally, the fourth function returns true if a thread is currently available in the thread pool, and false otherwise.

Message Library

All network protocols need to manipulate messages by adding/striping headers. Our goal is to design and implement a class `Message` that manages messages. This class provides the following (public) member functions (again, you may add more (private) member functions as needed for your implementation):

`Message( )`
`Message(char*` *msg*`, size_t` *len*`)`
`~Message( )`
`void msgAddHdr(char *`*hdr*`, size_t` *len*`)`
`char * msgStripHdr(int` *len*`)`
`int msgSplit(Message&` *secondMsg*`, int` *len*`)`
`void msgJoin(Message&` *secondMsg*`)`
`size_t msgLen( )`
`void msgFlat(char *`*buffer*`)`

The first two functions are constructor functions that create a message object and the third function is the destructor function. The fourth function (`msgAddHdr())` attaches a header (*hdr*) of *len* bytes to the front of the message object. The fifth function (`msgStripHdr())` strips *len* bytes from the front of the message object. It returns a pointer to the location of the stripped bytes (header). The sixth function (`msgSplit())` splits a message into two messages. The original message is reduced to length *len* bytes from the beginning and the *secondMsg* is the remaining part of the original message. This functiom returns 1 on success and 0 on failure. The seventh function (`msgJoin())` joins the original message with *secondMsg*. The *secondMsg* will be an empty message (message with length zero bytes) after the function returns. The eighth function (`msgLen())` returns the length of the message. Finally, the last function copies the message to the *buffer*. This function assumes that there is sufficient memory allocated in *buffer*.

The first rule in implementing a message library is to avoid copying data from one buffer into another. In fact, if this is the only rule your implementation adheres to, your implementation will probably be quite efficient. The reason is that every time you copy data—where

copying data implies a loop that loads every word of one buffer into a CPU register and then stores it into another memory location—you dramatically affect the end-to-end throughput of the network. So, it is critical that operations that manipulate messages do not touch the data in a message (as much as possible), but rather, only manipulate pointers.

Event Scheduler Library

Another common function that protocols need is to schedule events to occur at some time in the future. For example, a protocol implementing reliable message delivery needs to schedule a retransmission event to occur at some future time. Our goal is to design and implement a class `EventScheduler` that provides the following (public) member functions (again, you may add more (private) member functions as needed for your implementation):

`EventScheduler(size_t` *maxEvents*`)`
`~EventScheduler( )`
`int eventSchedule(void` *evFunction*`(void *), void *`*arg*`, int` *timeout*`)`
`void eventCancel(int` *eventId*`)`

The first function is a constructor function that creates an EventScheduler object that can schedule up to *maxEvents* (default value: 10) events. The second function is the destructor function. The third function schedules an event to execute the function *evFunction*( ) with one argument *arg* after a delay of *timeout* time units (microseconds/milliseconds/seconds depending on the granularity of the timer provided in the threads package you use). This function returns an integer that is used as the event id of the event scheduled. Finally, the third function cancels a previously scheduled event whose event id is *eventId*.

Assignment submission

1. Submission deadline is Friday, October 31 midnight. No extensions will be given unless there is a valid excuse.
2. Submit a single zip file via the submission link on Moodle. Your zip file must include a README file. Include all files that are needed for compiling and running your program.
3. DO NOT include any object files in your submission.
4. In the README file, provide the following information: Your name; instructions on how to compile and run your program; current status of your program: whether it compiles or not, known bugs/limitations/unusual features, what parts of the program work, etc.; any other information that will be useful in grading your program.
5. Some test programs will be provided. Please include your output for all test programs in the zip file.