CS 502/CS 3013 Operating Systems — WPI, Summer 2024
Craig E. Wills — Project 2 (25 pts)
Assigned: Thursday, May 30, 2024 — Due: Thursday, June 13, 2024 (11:59pm)

Introduction

The use of threads allows for easy sharing of resources within a process with mechanisms such as mutex locks and semaphores for coordinating the actions of threads within a process. In this project you will use these facilities to build a message passing mechanism that can be used among a set of threads within the same process. You will use the facilities of the *pthreads* library for thread and synchronization routines.

Problem

The basic idea of this assignment is to create a number of threads and to associate with each thread a "mailbox" where a single message for that thread can be stored. Because one thread may be trying to store a message in another's mailbox that already contains a message, you will need to use semaphores in order to control access to each thread's mailbox. The number of threads in your program should be controlled by an argument given on the command line to your program (use *atoi()* to convert a string to an integer). You should use the constant `MAXTHREAD` with a value of 10 for the maximum number of threads that can be created.

The main thread in your program will be known as "thread 0" with threads 1 to the number given on the command line being threads created using the routine *pthread_create()*. The index of the thread is passed as the argument. In creating the threads (you can name the routine as "*adder*" since it will be summing up values), your main thread will need to remember the thread id stored in the first argument to *pthread_create()* for later use.

Associated with each thread is a mailbox. A mailbox contains one message that is defined by the following C/C++ structure:

```
struct msg {
    int iFrom;  /* who sent the message (0 .. number-of-threads) */
    int value;  /* its value */
    int cnt;    /* count of operations (not needed by all msgs) */
    int tot;    /* total time (not needed by all msgs) */
};
```

Notice that the identifiers used in messages are in the range 0 to the number of threads. We will call this the "mailbox id" of a thread to distinguish it from the thread id returned by *pthread_create()*. In the first part of the project the type of the message can either be `REQUEST` or `REPLY` as defined below.

```
#define REQUEST 1
#define REPLY 2
```

Because each thread has one mailbox associated with it, you should define an array of mailboxes (of length `MAXTHREAD`) to store one message for each potential thread. Because mailboxes must be shared by all threads this array must be defined as a global variable. Alternately, you can dynamically allocate only enough space for the number of threads given on the command line.

Similarly, semaphores should be created to control access to each mailbox. Semaphores should be created using the routine *sem_init()*. These semaphores should also be created by the main thread before it creates the other threads. You could put all mailbox setup code in a *InitMailBox()* routine.

To handle access to each thread's mailbox you must write two procedures: *SendMsg()* and *RecvMsg()*. These procedures have the following choices for interfaces:

```
SendMsg(int iTo, struct msg *pMsg)  // msg as ptr, C/C++
SendMsg(int iTo, struct msg &Msg)   // msg as reference, C++
/* iTo - mailbox to send to */
/* pMsg - message to be sent */

RecvMsg(int iRecv, struct msg *pMsg) // msg as ptr, C/C++
RecvMsg(int iRecv, struct msg &Msg) // msg as reference, C++
/* iRecv - mailbox to receive from */
/* pMsg - message structure to fill in with received message */
```

Alternately, you can define a message to be a C++ class with *Send()* and *Recv()* methods each taking a single argument.

The index of a thread is simply its number so the index of the main thread is zero, the first created thread is one, etc. Each thread must have its own index. The *SendMsg()* routine should block if another message is already in the recipient's mailbox. Similarly, the *RecvMsg()* routine should block if no message is available in the mailbox.

Basic Objective

After initialization and creating all child threads your main program will read input (from stdin using routines such as *cin* or *fgets()*) in the form of two numbers per line:

```
3 1
4 2
6 1
5 1
7 3
9 2
```

Each line consists of a value (first number) and index of the thread to send this value to (second number). The main thread should use the routine *SendMsg()* to send the value (within a message) to the thread. When the main thread detects an error on read (such as an end-of-file (`EOF`)) it should send a termination message (message with value -1) to each child thread. After sending a termination message it should wait for as many messages as there are child threads. It should print out the result from each child, wait for each created thread to complete using *pthread_join()*, and clean up semaphores it has created.

Each child thread should sum up the values of messages that it receives. It uses the routine *RecvMsg()* to wait for messages. For example, the `ith` thread would use *RecvMsg(i, &msg)* (or *RecvMsg(i, msg)* if using call-by-reference) where message is declared as

```
struct msg msg; /* message structure to contain received message */
```

If the value of a message is non-negative then the child thread adds the value to its total and sleeps for one second (using Unix library call *sleep(3)*). If the value is negative then the child thread quits receiving messages and immediately (it does not sleep here) sends its total, count of add operations and total execution time back to the main thread (thread 0). The child thread then exits. The total execution time should be determined using the Unix library call *time(3)*, which maintains a running count of the time in seconds since January 1, 1970. You can use this library call (within the *adder()* routine) to measure the time from when the *adder()* thread is created until it receives its termination message.

The output of the program with the sample data given above would be as below (the order of the lines may vary depending on the order of messages received by the main thread). The amount of time for each thread can vary depending on the load of the system and could be larger than shown.

```
The result from thread 2 is 13 from 2 operations during 2 secs.
The result from thread 3 is 7 from 1 operations during 2 secs.
The result from thread 1 is 14 from 3 operations during 3 secs.
```

Hints

For the program, you should proceed by initially fixing the number of child threads to one so there is just one producer of messages (the main thread) and just one consumer (a child thread). After getting your program to work then increase the value of threads. If your program is written correctly then you will only need to change the command line. Access to a mailbox is just a producer/consumer problem. Your implementation of *SendMsg()* and *RecvMsg()* should be similar to implementations of this problem that we have looked at in class.

Your program should perform error checking on the input data. If the number of values on a line is not two then you may assume end-of-file has been reached. If a line contains a negative value as the first entry or a number that is not between one and number of threads as the second entry then you should print an appropriate error message and skip this line. You are expected to generate your own test data for checking your program and should turnin this test data with your program.

Additional Work

Satisfactory completion of the basic objective of this assignment is worth 21 of the 25 points. For the additional points, you need to implement another send primitive *NBSendMsg()* with the same arguments as *SendMsg()* except that it should not block if the recipient mailbox already contains a message. In this case *NBSendMsg()* should return immediately with a return code of -1. It should return a value of zero if the message is placed in the mailbox. Using the *man* command on *sem_wait()* should provide information on another routine *sem_trywait()*, which might be useful for this work.

You then need to demonstrate the use of the *NBSendMsg()* routine. To do this aspect of the program you will need to change how the main program works (there should be no change to the *adder()* threads). The main program should turn on this option if a second argument to the command is "`nb`" such as

```
% mailbox 3 nb
```

In this case the main program should use the *NBSendMsg()* routine to send requests to each of four *adder()* threads. If *NBSendMsg()* returns without error then this version of the program will work exactly like the basic version. However, if the *NBSendMsg()* routine fails to deliver a message (as would be the case for the "`5 1`" line in the sample data) then you should add that request to a queue of undelivered requests (you need to create this queue).

When your main program completes reading all of the input, it should cycle trying to send (using *NBSendMsg()*) undelivered requests from the queue. It should continue in this manner until all messages have been delivered. In addition, the main program should immediately send (using *NBSendMsg()*) a termination message to all threads for which it has delivered all requests. In the sample data the termination message would attempted to be sent to threads 2 and 3 immediately after reading all input data, and to thread 1 after successfully delivering the "`5 1`" request. It should try to deliver termination messages to a thread until it succeeds.

After all termination messages have been delivered, the main program should wait to receive the totals from each thread, print them out and clean up all resources. The output you should receive for this portion of the project with the sample data is:

```
The result from thread 3 is 7 from 1 operations during 1 secs.
The result from thread 2 is 13 from 2 operations during 2 secs.
The result from thread 1 is 14 from 3 operations during 3 secs.
```

where each *adder()* thread returns a summary message just as soon as it receives the termination message. With the use of the *NBSendMsg()* routine, the main thread never blocks and may send requests to threads sooner than in the basic version of this assignment. Remember that only the main program needs to change for this version and that only the main program should use the *NBSendMsg()* primitive for sending messages.

Submission of Project

Please compress all the files together as a single .zip archive for submission. In addition to source files and makefile that compiles your code, you should include a `typescript` file (created using *script* program) showing sample execution of your program on various test inputs.

Please upload your .zip archive via InstructAssist with the project name of *proj2*.