CS 502 Operating Systems — WPI, Fall 2003
Craig E. Wills — Project 2 (30 pts)
Assigned: Monday, September 22, 2003 — Due: Monday, October 13, 2003

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 the threads within the 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.

To exercise these routines you will first build a multi-threaded program to add up a range of numbers. Once working the primary purpose of the project is to build a game of life program where work is distributed among a set of threads.

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 parent 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()*. In creating the threads, your program 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 a message that is defined by the following C/C++ structure:

```
struct msg {
    int iFrom;  /* who sent the message (0 .. number-of-threads) */
    int type;   /* its type */
    int value1;  /* first value */
    int value2;  /* second value */
};
```

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 `RANGE` or `ALLDONE` as defined below.

```
#define RANGE 1
#define ALLDONE 2
```

Because each thread has one mailbox associated with it, you should define an array of mailboxes (of length `MAXTHREAD+1`) 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. These semaphores should be created using the routine *sem_init()* that is available by linking with the "rt" library in Unix/Linux. These semaphores should also be created by the main thread before it creates the other threads. All creation of semaphores for the mailboxes should be done in the routine *InitMailbox()*, which you need to write.

To handle access to each thread's mailbox you must write two procedures: *SendMsg()* and *RecvMsg()*. These procedures have the following 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 iFrom, struct msg *pMsg) // msg as ptr, C/C++
RecvMsg(int iFrom, struct msg &Msg) // msg as reference, C++
/* iFrom - mailbox to receive from */
/* pMsg/Msg - message structure to fill in with received message */
```

The index of a thread is simply its number so the index of the parent 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.

Part I

After setting up the mailboxes and creating the threads your program will need to exercise these routines. As a simple test, you will use multiple threads to add up the numbers between one and an integer given on the command line. Once a thread is created, it should wait for a message of type `RANGE` from the parent thread (mailbox id 0) by calling *RecvMsg()* with its mailbox id as the first argument. When it receives such a message, the child thread should add the numbers between value1 and value2 and return the result to the parent thread with a message of type `ALLDONE`. The routine *pthread_exit()* can be used to terminate a thread before the end of the code if need be.

Once the parent thread has received `ALLDONE` responses from all created threads, it should print the summary total of all response and wait for each created thread to complete using *pthread_join()*. Once each thread has completed the parent should clean up semaphores it has created and terminate.

The solution to the first part of the project should be called *addem* with a sample invocation shown below.

```
addem 10 100
The total for 1 to 100 using 10 threads is 5050.
```

Routines

In summary, the routines you must write with the given interface:

- InitMailbox() – return 0 on success, -1 on failure. Initializes semaphores.
- CleanupMailbox() – return 0 on success, -1 on failure. Cleans up all semaphores. Should be called whenever the parent thread exits.
- SendMsg(iTo, pMsg) – sends the message to the destination mailbox.
- RecvMsg(iFrom, pMsg) – receives a message from the source mailbox.

Part II: John Conway's Game of Life

Successful completion of part I of the project is worth 15 of the 30 project points. For the second part of the project, you should save a copy of your part I code and modify it for part II. Part II, which should be called *life*, will play a distributed version of the Game of Life.

The Game of Life was invented by John Conway. The original article describing the game can be found in the April 1970 issue of Scientific American (`http://www.sciam.com/`), page 120. The game is played on a grid of cells, each of which has eight neighbors (adjacent cells). A cell is either occupied (by an organism) or not. For boundary cases, assume cells outside of the grid are unoccupied. The rules for deriving a generation from the previous one are these:

- Death. If an occupied cell has 0, 1, 4, 5, 6, 7, or 8 occupied neighbors, the organism dies (0 or 1 of loneliness; 4 thru 8 of overcrowding).
- Survival. If an occupied cell has two or three neighbors, the organism survives to the next generation.
- Birth. If an unoccupied cell has three occupied neighbors, it becomes occupied.

Once started with an initial configuration of organisms (Generation 0), the game continues from one generation to the next until one of three conditions is met for termination:

1. all organisms die, or
2. the pattern of organisms is unchanged from one generation to the next, or
3. a predefined number of generations is reached.

The straightforward way in which to implement the program is to maintain two separate two-dimensional arrays for even and odd generations. These can be maintained as global variables. For the project you should define a variable `MAXGRID` as the maximum number of rows or columns in the grid. `MAXGRID` should be set to 40.

Distributed Version

The distributed version of the game follows the same rules as the standard game, but rather than have a single-threaded process evaluate all cells for each generation, a distributed version will use multiple threads to perform the work. This approach could improve performance on a multi-processor, but introduces added complexity on synchronizing the activities of the threads.

You will use thread 0 to coordinate the activities of one or more worker threads, which are actually doing the work of playing the game. Each worker thread computes the new generation of the game for an assigned range of rows as initially specified by thread 0. The number of rows assigned to each thread by thread 0 should be roughly equal. After each generation, each thread reports results back to thread 0 and waits for a `GO` message from thread 0 before continuing to the next generation. The message types needed for the program are defined below along with the algorithm for each worker thread on how they are used.

```
#define RANGE 1
#define ALLDONE 2
#define GO 3
#define STOP 4
#define ALLZERO 5
#define STATIC 6
#define GENDONE 7  // Generation Done

Receive RANGE message from thread 0 to obtain range of rows.
for (gen = 1; gen <= cGen; gen++) {
    Receive message from thread 0.
    If type is STOP then exit the thread.
    If type is GO then play a generation of the game on the thread's
        portion of rows.
    If portion is all zeros then send ALLZERO message to thread 0.
    If portion is unchanged then send STATIC message to thread 0.
    Otherwise send GENDONE message to thread 0.
}
Send ALLDONE message to thread 0.
```

Thread 0 controls when each generation of the game is played using `GO` messages to ensure that all threads have played a generation before proceeding to the next generation. There is no specific order in which worker threads must play each generation. Each thread must only update the cells for its range of rows, but for rows adjacent to those for another thread it is fine for a thread to read the values of cells in rows outside of its range.

It is the responsibility of thread 0 to decide when the game is done using the termination rules previously specified. Thread 0 must combine the results for all worker threads to decide if the game is done in which case it sends a `STOP` message to all threads or if the game is not done in which case it sends a `GO` message to all. Be careful in combining the respective results from each thread because even though one thread may have no or static life, that may not be the case for the regions of all threads.

The file containing Generation 0 will be an ASCII file consisting of a sequence of 0's and 1's indicating whether a cell is vacant or not. There will be a single space between each digit. For example, if the file contains

```
0 1 0 0
0 0 1 1
1 0 0 0
```

then the world consists of three rows and four columns. Your program should ensure that neither the number of rows nor columns exceeds `MAXGRID`. If so it should print a message and terminate.

Your program should accept 3-5 command line arguments with the following syntax:

```
life threads filename generations print input
```

with the following meaning:

- threads: number of threads with value `1-MAXTHREAD`.
- filename: file containing generation 0. Note that if the generation contains fewer rows than the given number of threads, you should set the number of threads to the number of rows so each thread has at least one row to work on.
- generations: the maximum number of generations to play with a value greater than zero.
- print: an optional argument with value of "y" or "n" on whether each generation (including generation 0) should be printed before proceeding to the next generation. The default value is "n".
- input: an optional argument with value of "y" or "n" on whether keyboard input should be required before proceeding to the next generation. The default value is "n".

Regardless of whether intermediate generations are printed, thread 0 should print the total number of generations and final configuration before waiting for all threads to complete and cleaning up semaphores. An example invocation with the previous example used for contents of "gen0" would be:

```
life 3 gen0 10 y
Generation 0
0 1 0 0
0 0 1 1
1 0 0 0

Generation 1:
0 0 0 0
0 1 1 0
0 0 0 0

The game ends after 2 generations with:
0 0 0 0
0 0 0 0
0 0 0 0
```

Part III: The Game of Life with a Barrier Implementation

Successful completion of parts I and II of the project are worth 22 of the 30 project points. For the third part of the project, you should save a copy of part II and create a program called *barrierlife*. In this part, the input and output for the project are the same as part II, but the implementation will differ to *not* use message passing rather you will be implementing and using a *barrier*.

A synchronization mechanism to coordinate the actions of a group of threads (or processes) is a *barrier*. When a thread within the group reaches a synchronization point (the barrier), it is blocked until all threads have reached the barrier. When all threads have reached the barrier then they are all allowed to proceed.

For this part of the project you need to create three routines:

- InitBarrier(N) - create a barrier for N threads. The value of N used should be equal to the number of worker threads. This routine creates and initializes needed semaphores and/or mutexes.
- Barrier() - blocks the calling thread until all N threads have reached the barrier point at which time the routine returns for all threads.
- CleanupBarrier(N) - cleanup created semaphores and/or mutexes.

Successful implementation of these routines using pthread synchronization primitives are worth an additional four points. For the remaining four points you need to modify your worker thread algorithm to use *no* message passing. You should modify the creation of a thread to pass the range of rows for the thread to use. Waiting for a message at the beginning of each loop should be replaced by a call to the *Barrier()* routine. You will need to create a distributed method for worker threads to determine when the game is done. You may create additional shared variables, but you may not use any message passing for this portion of the project.

Submission of Project

Use `/cs/bin/turnin` to turn in your project using the assignment name "proj2". Should have separate source files for *addem*, *life*, and *barrierlife*. A Makefile with rules for creating each executable is also expected.