CS 3013 Operating Systems — WPI, A Term 2020
Craig E. Wills — Project 3 (25 pts)
Assigned: Tuesday, September 22, 2020 — Due: Tuesday, October 6, 2020

Important: This project is to be done by each individual student.

Introduction

In this project you will be using threads along with semaphores as a synchronization mechanism to 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. Your program must be coded in C or C++.

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()*. The index of the thread is passed as the argument. 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 iSender; /* sender of 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. 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.

To handle access to each thread's mailbox you must write two procedures: *SendMsg()* and *RecvMsg()*. These procedures have one of 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/Msg - 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 */
```

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 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, wait for each created thread to complete using *pthread_join()*, and clean up semaphores it has created.

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.
```

Part II: John Conway's Game of Life

Successful completion of part I of the project is worth 14 of the 25 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 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 an outline of the algorithm for each worker thread on how they are used.

```
#define RANGE 1
#define ALLDONE 2
#define GO 3
#define GENDONE 4  // Generation Done

Receive RANGE message from thread 0 to obtain range of rows.
for (gen = 1; gen <= cGen; gen++) {
    Receive GO message from thread 0 and play a generation
        of the game on the thread's portion of rows.
    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.

The file containing Generation 0 will be an ASCII text 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 0
1 0 0 1
```

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. It is suggested that you read a single line of input as a time into a character array using *cin.getline()* or *fgets()* and then parse the characters in the array to initialize generation 0.

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 number of generations to play. Number must be 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 (missing lines indicated with `...`):

```
% ./life 3 gen0 10 y
Generation 0
0 1 0 0
0 0 1 0
1 0 0 1

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

Generation 2:
0 0 0 0
0 0 0 0
0 0 0 0

...

The game ends after 10 generations with:
0 0 0 0
0 0 0 0
0 0 0 0
```

Optimization of Distributed Version

Correct implementation of the distributed version of life is worth an additional seven points. For the remaining four points of the project you need to optimize your distributed game of life so instead of always playing the specified number of generations, it now terminates when one of three conditions is met:

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.

Applying this optimized algorithm to the previous example would cause the game to end after 2 generations rather than the 10 generations shown for the unoptimized version of the game.

In order to implement this optimized version, you will need to introduce additional message types so each worker thread does not just indicate it is done with a generation (by sending message type `GENDONE`), but rather the thread indicates whether one or both of the alternate conditions for termination exists on its range of rows.

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.

Make

A useful program for maintaining large programs that have been broken into many software modules is *make*. The program uses a file, usually named `Makefile`, that resides in the same directory as the source code. This file describes how to "make" a number of targets specified. The following is a portion of the `Makefile` linked on the course Web page. Typing "`make pcthreads-C`" causes the executable file *pcthreads-C* to be created from `pcthreads.C`.

You should copy the sample `Makefile` and modify it for your project. At the minimum, you should have a target to create *addem*, *life*, *all* (which creates both), and *clean* (which removes object and executable files). Note the sample `Makefile` can be used to compile both C and C++ versions of the sample programs. You will only need to use one language in your project.

As a matter of explanation the first three lines below simply define and give values to *make* variables. The remaining text describes how to make the target "pcthreads-C" and pcthreads-c". WARNING: The operation line(s) for a target MUST begin with a TAB (do not use spaces). See the man page for *make*, *g++* and *gcc* for additional information.

```
LIB=-lpthread
CC=gcc
CCPP=g++

pcthreads-C: pcthreads.C
        $(CCPP) pcthreads.C -o pcthreads-C $(LIB)

pcthreads-c: pcthreads.c
        $(CC) pcthreads.c -o pcthreads-c $(LIB)
```

Submission of Project

Use InstructAssist to turn in your project using the assignment name "proj3". You should have *separate* source files for *addem* and *life* and have a Makefile that compiles them into executables of these names. You must also turnin a script showing test execution of your program(s).