Programming Assignment 5

Due Wednesday, December 9
(Note the change in the due date)

Conway's Game of Life

The mathematician John Conway invented his "Game of Life" in 1970. It's an example of a mathematical object called a *cellular automaton.* It has received a lot of attention over the years because of the interesting patterns it creates and its mathematical power.

The game is "played" on a rectangular grid or *world.* At any time each square or *cell* of the grid is *alive* or *dead.* After the initial configuration or generation 0 is either entered by the user or generated using a random number generator, successive generations are determined using the following rules.

- Any living cell with fewer than 2 living neighbors will die from loneliness.
- Any living cell with exactly 2 or 3 living neighbors will survive into the next generation.
- Any living cell with more than 3 living neighbors will die from overcrowding.
- Any dead cell with exactly 3 living neighbors will come to life in the next generation.
- Any other dead cell will remain dead in the next generation.

All of the changes occur simultaneously.

Serial Implementation

Since all of the updates to the world occur at the same time, a serial program implementing Life can use use storage for two worlds: one for the current generation and one for the next generation. The current generation world is used for counting neighbors, and the next generation is used for recording the state of each cell in the next generation. In C, the program can simply swap pointers to the block of memory being used for the current generation and the next generation. Then the storage for the old current generation can be used for recording the next new generation.

In Conway's definition of Life, the world is an infinite grid. This, of course, isn't feasible in a computer program. Four practical alternatives are commonly used:

1. Enclose the (finite) grid in a boundary of permanently dead cells. This is easy to program, but it violates the rules of Life, when a boundary cell is adjacent to three live cells.

2. Identify the top edge with the bottom and the left edge with the right edge. This creates a "toroidal" world, in which live cells on one edge can "give birth to" cells on the opposite edge.

3. Dynamically allocate and deallocate subgrids as needed.

4. Abandon the use of a rectangular grid for internal storage and simply store the coordinates of live cells.

The second option is the easiest to adapt to a multithreaded implementation. So it's the one we'll use. The program `life_tor.c` on the class website implements this version.

Programming Assignment

For programming assignment 5, you should use Pthreads to parallelize Life on a toroidal world. The parallelization should use a grid of threads with r rows and s columns. You can assume that r will evenly divide m, the number of rows in the world, and s will divide n, the number of columns. In this setting each thread is responsible for finding the next generation in a subworld consisting of m/r consecutive rows and n/s consecutive columns.

Input to the program will come from the command line and `stdin`. The command line will have the form

```
$ ./pth_life <r> <s> <m> <n> <max> <i|g>
```

The arguments should be interpreted as follows:

```
   r:  number of rows of threads
   s:  number of columns of threads
   m:  number of rows in the world
   n:  number of columns in the world
 max:  maximum number of generations the program should compute
 'i':  user will enter the initial world (generation 0) on stdin
 'g':  the program should use a random number generator to
       generate the initial world.
```

If the command line argument `'g'` is given, your program should use the `random/srandom` random number generator in the C library (defined in `stdlib.h`). The generator should be seeded with the call `srandom(1)`. The user should enter the probability of a live cell in generation 0 on `stdin`. This will be a double between 0 and 1. The program can generate probabilities with the code

```
double prob = random()/((double) RAND_MAX);
```

If this value is $\leq$ the user entered probability, then the cell should be alive. Otherwise the cell should be dead.

Output should be the same as the output of the serial program `life_tor.c`. In particular, if all the cells die out before the user input maximum number of generations have been computed, then the program should terminate.

Note that the maximum number of generations is the number of generations *after* generation 0. For example, if the user specifies that the maximum number of generations is 5. Then the program should print generations 0, 1, 2, 3, 4, and 5. This, of course, assumes that the cells don't die out in any of these generations.

Implementation Details

The serial program `life_tor.c` uses a one-dimensional array of ints to store the world, with 1 representing a live cell and 0 representing a dead cell. You're not required to use this data structure, but your program *must* print an `'X'` to represent a live cell and a space or blank to represent a dead cell.

Clearly the threads need to be synchronized after they've finished computing the contents of the world for the next generation. This should *not* be done with a Pthreads barrier. Rather your program should use a condition variable.

In order to insure that the output is correctly printed, only one thread should print any one generation, but different threads, if properly synchronized, can print different generations.

In addition to synchronizing the threads it's also necessary to determine whether any cells are living in the newly computed world: if not, the threads should return and the program should terminate.

Your program should be efficient in its implementation of the basic serial algorithm. It should also be efficient in its implementation of the thread synchronization. For example, it should *not* require multiple barriers to implement the synchronization after the threads complete the calculation of a new generation.

Submission

As usual, your final electronic copy should be committed to your CS SVN repository by 6 pm on Wednesday, December 9. (Note that this is different from the due date in the syllabus.)

Grading

1. Correctness will be 65% of your grade. Does your program correctly partition the world among the threads? Are the successive generations identical to the generations computed by the serial program? Does the program correctly detect that all the cells have died and then terminate?

2. Documentation will be 10% of your grade. Does your header documentation include the author's name, the purpose of the program, and a description of how to use the program? Are the identifiers meaningful? Are any obscure constructs clearly explained? Does the function header documentation explain the purpose of the function, its arguments and its return value?

3. Source format will be 5% of your grade. Is the indentation consistent? Have blank lines been used so that the program is easy to read? Is the use of capitalization consistent? Are there any lines of source that are longer than 80 characters (i.e., wrap around the screen)?

4. Quality of solution will be 20% of your grade. Are any of your functions more than 40 lines (not including blank lines, curly braces, or comments)? Are there multipurpose functions? Is your solution too clever – i.e., has the solution been condensed to the point where it's incomprehensible?

   Quality of solution also includes correct allocation and deallocation of storage, implementation of thread synchronization, and implementation of the basic Life algorithm.

Academic Honesty

Remember that you can discuss the program with your classmates, but you cannot look at anyone else's pseudo-code or source code. (This includes source code that you might find on the internet.) You also cannot show your source code or pseudo-code to anyone else.