CS 460 Programming Assignment #5 MultiThreaded Game of Life

Date Assigned: April 7, 2014
Date Due: April 21, 2014
Points: 75

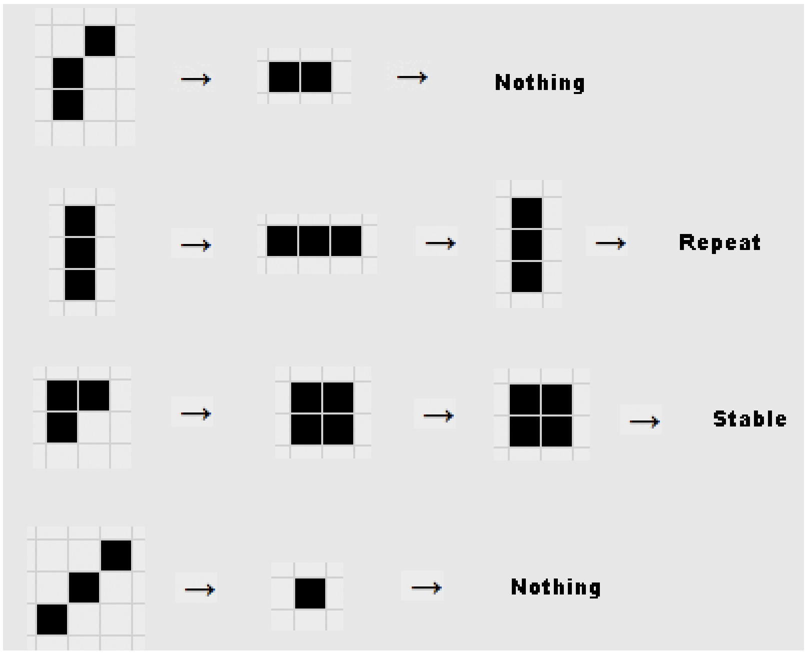
Goal:
Learn more about POSIX threads, synchronization, and mutexes.

Description:

You will implement a multithreaded version of John Conway's Game of Life. The Game of Life is a simulation of organisms living, dying, and being born on a grid as described by the four rules:

1. Any live cell with fewer than two live neighbours dies, as if by loneliness.
2. Any live cell with more than three live neighbours dies, as if by overcrowding.
3. Any live cell with two or three live neighbours lives, unchanged, to the next generation.
4. Any dead cell with exactly three live neighbours comes to life.

Each rule is applied instantaneously at a generation. That means that the result of a rule does not affect other organisms until the following generation. Here are some examples:



http://www.math.cornell.edu/~lipa/mec/lesson6.html

A description of Conway's Game of Life can be found at
http://en.wikipedia.org/wiki/Conway's_Game_of_Life

Your Program:

Your program will read command line options to determine the input and output files, as well as the number of thread and number of generations to run. See the **Execution Format** section below. The input and output files have the same format. The height and width of the board are not necessarily equal.

Immediately before writing the game board back to a file, display the total number of births and total number of deaths that occurred over all generations. You also need to display the time, in seconds to 2 decimal places, your program took to run all the generations. This time must not include any File I/O.

Your program will need to run each generation using multiple threads. For *n* threads, each thread must do (about) *1/n* of the total work. By using more than one thread, the runtime of your program must decrease if you are using an SMP machine or multicore processor.

How you divide up the work among the threads is your design decision. There are many correct ways to do this and any that achieve a speed up and produces a correct answer will be acceptable.

Execution Format:

```
CS460_Life inputFile outputFile #Threads #Generations flag

where flag can be:

-d   displays each generation, starting with the initial generation, to the
     screen one screen at a time waiting for the user to hit return between
     screens

-g   displays for each generation, the total number of births and deaths for
     that generation as well as a total number of births and deaths for all
     generations

-a   displays for all generations the total number of births and deaths
```

If incorrect command line options, or a nonexistent input file is given, display a nice usage message.

File Format:

The format of the file is as follows:

boardcolumns
boardrows
cell values where a 0 indicates an empty spot, a 1 indicates a spot with an organism in it.

Sample file:

```
12
12
0 1 0 1 1 0 1 0 1 0 0 0
1 1 1 0 0 1 1 0 0 1 1 1
and so on
```

Constraints:

You may assume the height and width of the board are always evenly divisible by the number of threads to run (the smallest board is 12x12).

Sample Output:

```
bart$ ./CS460_Life bigTable.life bigTable.gen2.life 4 2 -g
Generation 1:   DEATHS: 8182132 BIRTHS: 2974039
Generation 2:   DEATHS: 2601626 BIRTHS: 1571647

TOTAL DEATHS: 10783758     TOTAL BIRTHS: 4545686

Time: 15.12 seconds
```

Functions (and such) you will (probably) need:

pthread_create(), pthread.h, pthread_exit(), pthread_kill(), pthread_mutex_lock(), clock_gettime(), pthread_mutex_unlock(), pthread_mutex_init(), fopen(), read(), write(), close(), fcntl.h, printf(), etc.

Execution Timing:

For timing our programs, we will ignore any File I/O; therefore,

1. do any initializations and load the initial table
2. start timer [clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &starttime);]
3. execute the number of generations
4. stop timer [clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &endtime);]
5. write the final table after the specified execution of generations and perform cleanup
6. print results as in the Sample Output above

Note: For –d, only display the table and nothing else. The –d option is used for debugging tables that fit on the screen from generation to generation.

Name your project **CS460_Life_PUNetID**.

Makefile Targets:

CS460_Life: build the executable CS460_Life at the root of the project.

clean:

valgrind: Run the following:

```
valgrind -v --leak-check=yes ./CS460_Life smallTable.life smallTableValgrind.life 1 4
```

Testing:

I will discuss testing your code on various machines at a later date.

You can restrict your executable to one core using the following command:

```
taskset -c 1 ./CS460_Life ......
```

Running Valgrind on your code *may* produce a 100x slowdown!

Valgrind may report that pthread_create leaks memory if you don't call pthread_join.

From the shell the command: **cat /proc/cpuinfo** will display information about the current machine's CPU.

Notes:

- Sample input and output files will be posted on the class schedule next week. You will need to link against libpthread.so, ie: `gcc –o CS460_Life CS460_Life.o -lpthread`
- Your project needs to be well documented and broken into well-defined functions.
- Minimize global variables!
- Build a struct to pass to your thread! Large boards with many generations may take many tens of minutes to complete.
- Using the **H** command inside the linux utility *top* will show you each thread instead of each process.
- If you want to display a game board as an image (black pixel for a living cell and white for an empty spot) use the commands:
- echo P1 | cat - board.life | display -
- To display large boards make take many tens of seconds. man display for more info!