Threads I

CSC314 Operating Systems

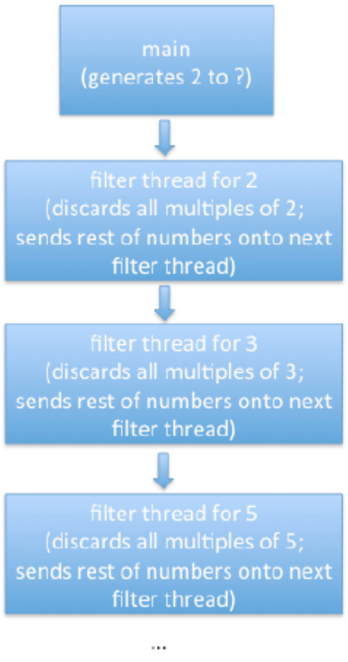
NOTE: Use the POSIX pthread and semaphore (sem_post/sem_wait) libraries to solve the following thread problems.

Problem A: Sieve of Eratosthenes

Implement a multithreaded version of the Sieve of Eratosthenes. The Sieve of Eratosthenes is an ancient algorithm to compute prime numbers, attributed to the Greek mathematician Eratosthenes of Cyrene (276 BC - 194 BC). You can read and view an animation of the Sieve here: http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes.

Here's an outline of a multithreaded version of the Sieve of Eratosthenes:

- the main thread generates an unbounded sequence of natural numbers starting at 2
- the main thread generates a SINGLE filter thread for the natural number 2 and then sends all subsequent numbers (3 to N) this filter thread via a SafeQueue data structure
- any number that is not a multiple of 2 is discarded by this filter thread
- any other number is a potential prime, so this filter thread passes the number onto the NEXT filter thread for processing, in this case the next filter thread will be for the prime number 3
- there will be a filter thread for each prime number that your Sieve uncovers so there's a filter thread for 2, 3, 5, 7, etc…
- when a new prime number is encountered, a filter thread is created for the prime number and it is connected to the previous filter thread via the SafeQueue data structure
- the first time a filter thread is created it will display the prime number that it is filtering
- the user specifies the NUMBER of prime numbers desired on the command line when executing the main program and the program terminates once that number of primes has been discovered


main
(generates 2 to ?)
filter thread for 2
(discards all multiples of 2; sends rest of numbers onto next filter thread)
filter thread for 3
(discards all multiples of 3; sends rest of numbers onto next filter thread)
filter thread for 5
(discards all multiples of 5; sends rest of numbers onto next filter thread)
...

Problem B: Scientific Simulation I

Write a non-threaded program to simulation heat dissipation on the surface of a cylinder. You can model a cylinder using a 2D HxC array of cells. H represents the height of the cylinder and C represents the circumference of the cylinder (position cylinder[x][0] is adjacent to position [x][C-1]).

Each cell has a temperature (a double) that is initialized to tinit.

Select N cells at random (hard code the number of cells and their locations for testing) and set these cells to temperatures that are higher or lower than tinit. The temperature can be different for each of these cells (use the same temperature for testing). The temperature at each of these cells will remain FIXED for the duration of the simulation.

For each iteration of the simulation the temperature of each cell is recomputed as the weighted average of:

- the previous iteration's temperature (X) using weighted coefficient C0
- the previous iteration temperatures of the 4 direct neighbor cells (N0, N1, N2, N3) using weighted coefficient C1
- the previous iteration's temperature of the 4 diagonal neighbor cells (D0, D1, D2, D3) using weighted coefficient C2

D0	N0	D1
N1	**X**	N2
D2	N3	D3

NOTE: There are three boundary conditions to consider:

- cylinder[x][0] and cylinder[x][C-1]
- cylinder[0][y] (no D0, N0, D1 available for weighted average)
- cylinder[H-1][y] (no D2, N3, D3 available for weighted average)

The simulation is complete when:
- max iterations have passed OR
- the simulation has converged on delta.

Convergence means that the difference between the previous iteration's cell temperature and the recomputed cell temperature is less than or equal to delta for all cells in the cylinder.

At the end of the simulation print out:

- the cylinder cells
- the maximum delta across the cells
- the average temperature across the cells
- the program execution time (actual cpu time used by the program not wall clock time)
- an estimate of the number of floating point operations per second (FLOPS)

To facilitate experiments, the simulator should accept H, C, tinit, c0, c1, c2, delta, max as command line arguments.

Report the output of running your program with these values in a file results.txt:

- H = C = 10, 100, 1000 (meaning 3 separate runs with 10x10, 100x100, 1000x1000)
- tinit=20C
- c0 = 0.5, c1=.1, c2=.025
- put a 40C heat source @ H/2,C/2
- put a 0C heat source @ H/4,C/4
- put a 0C heat source @ H-3, C-3
- experiment with delta and max

Here are some sample runs of the program:

```
main 10 10 20 .5 .1 .025 .001 10000
main 100 100 20 .5 .1 .025 .001 10000
main 1000 1000 20 .5 .1 .025 .001 10000

(H=10 or 100 or 1000, C=10 or 100 or 1000, tinit=20, c0=.5,
c1=.1, c2=.025, delta=.001, max=1000)
```

Problem C: Scientific Simulation II

Parallelize your program for the simulation of heat dissipation on the surface of a cylinder developed in Problem B using threads by adding the number of threads to be used to the list of command line arguments… assume the user won't ask you for more than 100 threads.

There are a wide range of design choices in the parallelization of your code and you should experiment with them. You to strive for "fast" code that aims at keeping the overhead from multithreaded execution small. For example, avoid repeated creation and termination of threads and have an eye on balancing the workload reasonably among the cooperating threads.

Some hints for implementation:

- Keep the threading solutions simple to avoid race conditions and synchronization problems.
- Compare the running time of your solution to problem B with your solution to problem C. Which solution takes less time and why?

Here is a sample run of the program that uses a max of 50 threads:

```
main 10 10 20 .5 .1 .025 .001 10000 50
```

Thanks to Dr. Clemens Grelck at the University of Amsterdam for these excellent concurrent programming assignments.