CpSc 418

Homework 3

Due: Nov. 9, 2012, 11:59pm

5% extra credit for if solution submitted by 11:59pm on Nov. 5.

Please submit your solution using the handin program. Submit the program as
cs418 hw3
This requires you to have an account on the UBC Computer Science undergraduate machines. If you need an account, go to: `https://www.cs.ubc.ca/students/undergrad/services/account`
to request one.
Your submission should consist of the following files:

`hw3.erl` – Erlang source (ASCII text). All functions requested in this assignment must be exported by this module.

`hw3.txt` – plain, ASCII text, or `hw3.pdf` – PDF.

The first file, `hw3.erl`, will be your solution to the programming part of the assignment, neatly commented.
The second file, `hw3.txt` or `hw3.pdf`, will be a plain text file of PDF file with your solution to the written part of the assignment. Handwritten solutions may be scanned and included in the hw3.pdf file.

1. **Mutual exclusion (15 points)** Figure 1 shows Dekker's mutual exclusion algorithm as presented in the October 4 lecture. A programmer decided to "simplify" the algorithm by changing the `while`-loop at line 3 to an `if`-statement – see figure 2. Here's their reasoning:

   > Now, consider the `while`-loop at lines 3–9. If the loop-body (lines 4–8) is executed by thread 0, then `turn` must be set to 0 when the while-loop at line 6 exits and execution proceeds to line 7. This means that thread 1 set turn to 0 at line 11 and will then set `flag[1]` to `false` at line 12 without entering its critical section. Nothing in lines 7–9 changes the value of `flag[1]`. This means that, `flag[1]` will be `false` (or about to be set to `false`) when thread 0 completes executing lines 7–9, and thread 0 will exit the `while` loop. Therefore, the loop body is executed at most once, and the `while`-loop can be replace by an `if`-statement.

   (a) **Counter-example trace (10 points)** Show that the modified version of the algorithm as shown in figure 2 does not guarantee mutual exclusion. In particular, show a counter-example trace like that on slide 31 of the October 4 slides (web only). Your trace should start with:

   ```
   PC0 = PC1 = 0;
   flag[0] = flag[1] = false;
   turn = 0;
   ```

   and end in a state with

   ```
   PC0 = PC1 = 10;
   ```

   (b) **Short explanation (5 points)** Write a short explanation (less than 50 words) of why the modified version doesn't work.

2. **Peterson's Algorithm (15 points)** Figure 3 shows Peterson's mutual exclusion algorithm. Peterson's insight is that when a thread tries to enter its critical section, it first sets `turn` to give the *other* thread priority. If both threads try to enter at roughly the same time, the last thread to try will set turn to give priority to the earlier thread.

   (a) **An Invariant (10 points)** Let

$$\begin{array}{rcll} I & = & \forall k \in \{0,1\}. & \texttt{flag[k]} = (3 \leq \texttt{PC}_k \leq 9) \\ & & \wedge & ((6 \leq \texttt{PC}_k \leq 8) \wedge \neg\texttt{tmp}_k) \Rightarrow (\neg\texttt{flag[}\overline{k}\texttt{]} \vee (\texttt{turn} = k) \vee (\texttt{PC}_{\overline{k}} = 3)) \\ & & \wedge & (\texttt{PC}_k = 8) \Rightarrow \neg\texttt{tmp}_k \end{array}$$

thread 0:

```
PC0= 0: while(true) {
PC0= 1:   non-critical code
PC0= 2:   flag[0] = true;
PC0= 3:   while(flag[1]) {
PC0= 4:     if(turn != 0) {
PC0= 5:       flag[0] = false;
PC0= 6:       while(turn != 0);
PC0= 7:       flag[0] = true;
PC0= 8:     }
PC0= 9:   }
PC0=10:   critical section
PC0=11:   turn = 1;
PC0=12:   flag[0] = false;
PC0=13: }
```

thread 1:

```
PC1= 0: while(true) {
PC1= 1:   non-critical code
PC1= 2:   flag[1] = true;
PC1= 3:   while(flag[0]) {
PC1= 4:     if(turn != 1) {
PC1= 5:       flag[1] = false;
PC1= 6:       while(turn != 1);
PC1= 7:       flag[1] = true;
PC1= 8:     }
PC1= 9:   }
PC1=10:   critical section
PC1=11:   turn = 0;
PC1=12:   flag[1] = false;
PC1=13: }
```

Figure 1: Dekker's Algorithm

thread 0:

```
PC0= 0: while(true) {
PC0= 1:   non-critical code
PC0= 2:   flag[0] = true;
PC0= 3:   if(flag[1]) { //while changed to if
PC0= 4:     if(turn != 0) {
PC0= 5:       flag[0] = false;
PC0= 6:       while(turn != 0);
PC0= 7:       flag[0] = true;
PC0= 8:     }
PC0= 9:   }
PC0=10:   critical section
PC0=11:   turn = 1;
PC0=12:   flag[0] = false;
PC0=13: }
```

thread 1:

```
PC1= 0: while(true) {
PC1= 1:   non-critical code
PC1= 2:   flag[1] = true;
PC1= 3:   if(flag[0]) { //while changed to if
PC1= 4:     if(turn != 1) {
PC1= 5:       flag[1] = false;
PC1= 6:       while(turn != 1);
PC1= 7:       flag[1] = true;
PC1= 8:     }
PC1= 9:   }
PC1=10:   critical section
PC1=11:   turn = 0;
PC1=12:   flag[1] = false;
PC1=13: }
```

Warning: this code does not guarantee mutual exclusion!

Figure 2: Modified Dekker's Algorithm

Initially: `PC`$_0$ = `PC`$_1$ = 0; `flag[0]` = `flag[1]` = `false`; `turn` = `0`.

thread 0:

```
PC0= 0: while(true) {
PC0= 1:   non-critical code
PC0= 2:   flag[0] = true;
PC0= 3:   turn = 1;
PC0= 4:   do {
PC0= 5:     tmp0 = flag[1];
PC0= 6:     tmp0 = tmp0 && (turn == 1);
PC0= 7:   } while(tmp0);
PC0= 8:   critical section
PC0= 9:   flag[0] = false;
PC0=10: }
```

thread 1:

```
PC1= 0: while(true) {
PC1= 1:   non-critical code
PC1= 2:   flag[1] = true;
PC1= 3:   turn = 0;
PC1= 4:   do {
PC1= 5:     tmp1 = flag[0];
PC1= 6:     tmp1 = tmp1 && (turn == 0);
PC1= 7:   } while(tmp1);
PC1= 8:   critical section
PC1= 9:   flag[1] = false;
PC1=10: }
```

Figure 3: Peterson's Mutual Exclusion Algorithm

where $\overline{k} = 1 - k$ (i.e. it is the index of the "other" thread).

Prove that I is an invariant of the program from figure 3.

Note: my version of Peterson's algorithm is a bit more pedantic than, for example, the version on wikipedia. They replace my loop at lines 4...7 with

```
    while(flag[k̄] && (turn == k̄));
```

I introduced the local variables `tmp`$_0$ and `tmp`$_1$ to show that the algorithm works even if `flag[`$\overline{k}$`]` or `turn` is modified by the other thread while computing the condition for continuing the loop.

(b) **Mutual Exclusion (5 points)** Prove that the invariant, I, from part (a) ensures mutual exclusion. In other words, show

$$I \quad \Rightarrow \quad \neg((\texttt{PC}_0 == 8) \wedge (\texttt{PC}_1 == 8))$$

3. **Mesh Networks (25 points)**

(a) **2-dimensional meshes (5 points)** Let $m > 0$ be an integer, and consider a 2D mesh network consisting of $N = m^2$ processors. For simplicity, assume that m is even. Each processor can be identified with a tuple, (i, j), where $0 \leq i, j < m$, and processor (i, j) has links to

processor $(i + 1, j)$, if $i < m - 1$;
processor $(i - 1, j)$, if $i > 0$;
processor $(i, j + 1)$, if $j < m - 1$;
processor $(i, j - 1)$, if $j > 0$.

Assume that each link can receive one message on each incoming link and send one message on each outgoing link using one unit of time.

Show that if each processor with $i < m/2$ sends distinct messages to each processor with $i \geq m/2$, then the time to convey these messages to their destinations is $O(N^{3/2})$.

(b) **d-dimensional meshes (5 points)** Let $m > 0$ and $d > 0$ be integers, and consider a d-dimensional mesh network consisting of $N = m^d$ processors. For simplicity, assume that m is even. Each processor can be identified with a tuple, $(i_0, i_1, \ldots, i_{d-1})$, where $0 \leq i_k < m$, and there is a link between a pair of processors iff all but one of their indices are identical, and they differ by ± 1 in the index for which they are different. Assume that each link can receive one message on each incoming link and send one message on each outgoing link using one unit of time.

Show that if each processor with $i_0 < m/2$ sends distinct messages to each processor with $i_0 \geq m/2$, then the time to convey these messages to their destinations is $O(N^{1+\frac{1}{d}})$.

(c) **d-dimensional messages (cont.) (5 points)** Now consider a d-dimensional mesh, and let A and B be any partitioning of the processors into to sets of size $N/2$. Show that if each processor in A sends distinct messages to each processor in B, then the time to convey these messages to their destinations is $O(N^{1+\frac{1}{d}})$.

(d) **How big is a d-dimensional mesh? (10 points)** Use the result from part (c) to show that if a d-dimensional mesh of $N = m^d$ processors is implemented in our 3-dimensional universe, then the volume of the mesh of processors is $O(N^{\frac{3}{2}(1-\frac{1}{d})})$. Compare this with the result for a hypercube from the October 9 lecture.

4. **Reduce (38 points)** Given any solution (yours, the solution set, a friend's, etc. – but give proper attribution if it's not yours) for finding all prime numbers that are less than or equal to N,

(a) **(10 points)** Write code to compute the sum of all primes that are less than or equal to N. You should measure the time that it takes to compute the sum given that the distributed list of primes has already been computed. Use the `wtree` module from the CpSc 418 erlang library. You should to use `wtree:create` to create a worker pool where the workers are organized as a binary tree, and `wtree:reduce` for the reduce operation to compute the sum. Note that the worker-pool returned by `wtree:create` can be used by any of the functions from module `workers` as well as those from `wtree`.

In a bit more detail, you should write a module called `hw3` that exports the function `sum/2` where

```
sum(W, Key) -> Total
```

`W` is a worker pool, `Key` is the name for the distributed list of primes, and `Total` is the sum of the primes in that distributed list.

For example, then the primes that are less than or equal to 100 are:

```
2,  3,  5,  7,  11, 13, 17, 19, 23, 29, 31
37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79
83, 89, 97
```

Assume that the primes that are less than or equal to 100 are stored in a distributed list that's associated with the atom `p100` for worker pool `W`, and that `p90` is a distributed list for the primes that are less than or equal to 90.

```
hw3:sum(W, p100).
```

should print

```
1060
```

Likewise,

```
hw3:sum(W, p90).
```

should print

```
963
```

(b) **(5 points)** Measure the speed-up of your implementation of `sum` compared with `lists:sum` (assuming that you have already retrieved all of the primes into a single list) when running on gambier.ugrad.cs.ubc.ca with 64 worker processes.

- What is the speed-up for the value of N for which the sequential version takes 1 second?
- What is the smallest value of N for which the parallel version achieves 80% of the speed up that you reported above?

(c) **(15 points)** Write code to find the pair of consecutive primes with the largest gap, for the primes that are less than or equal to N. If there is more than one such pair, return the first such pair.

In a bit more detail, module `hw3` should export `largest_gap/2` where

```
largest_gap(W, Key) -> {P1, P2}
```

`W` is a worker pool, `Key` is the name for the distributed list of primes, and `{P1, P2}` is the pair of primes in that distributed list with the largest gap.

Continuing the example from part (a),

```
hw3:largest_gap(W, p100).
```

should print

```
{89,97}
```

If the primes that are less than or equal to 90 are stored in a distributed list that's associated with the atom `p90`, then

```
hw3:largest_gap(W, p90).
```

should print

```
{23,29}
```

There are seven pairs of consecutive primes less than 90 with a gap of 6, but `{23,29}` is the *first* (i.e. smallest) such pair.

(d) **(8 points)** Write a sequential version of `largest_gap`. Measure the speed-up of your implementation of `largest_gap` compared with your sequential version (assuming that you have already retrieved all of the primes into a single list) when running on gambier.ugrad.cs.ubc.ca with 64 worker processes.

- What is the speed-up for the value of N for which the sequential version takes 1 second?
- What is the smallest value of N for which the parallel version achieves 80% of the speed up that you reported above?