CpSc 418 **Homework 4** Due: Nov. 29, 2012, 11:59pm

5% extra credit if solution submitted by 11:59pm on Nov. 25.

Please submit your solution using the handin program as:
cs418 hw4

1. **Scan (20 points)**
Consider the decaying average operation:

$$y_i = \sum_{j=1}^{i} \alpha^{i-j} x_j$$

In Erlang:

```
average_d(List, Alpha) -> average_help(List, Alpha, 0).
average_help([], _, _) -> [];
average_help([X | Tl], Alpha, V) ->
   V2 = Alpha*V + X,
   [V2 | average_help(Tl, Alpha, V2)].
```

   (a) **(5 points)** Let `x = [0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610]`. Draw a picture showing how decaying average can be computed using scan with four processors and $\alpha = 0.1$.

   See figure 1. I put in more detail than required for full credit. In particular, I sketched out what the `Leaf1` and `Leaf2` functions do. Just drawing the edges for the two trees and labeling them with the values that are sent along them is sufficient. Also, I divided the list into unequal size pieces. This let me show how different values for $\alpha^{\texttt{length(List)}}$ can be handled. My guess is that most solutions will divide the list into four segments of length four. That will get full credit as well.

   (b) **(15 points)** Write a parallel version using the `wtree:scan` function.

   See hw4.erl.

2. 0-1 Principle – extended (**20 points**)
By showing that other operations commute with compare-and-swap, we can extend the 0-1 principle wider classes of networks:

   (a) (**5** points) Let

   ```
   swap(A, B) -> {B, A}
   ```

   For any monotonic function, `F`, show that `F` commutes with `swap`.

F(swap(A,B))	=	*F({B,A})*	*definition of swap*
	=	*{F(B), F(A)}*	*extension of F to tuples*
	=	*swap(F(B), F(A))*	*definition of swap*

   Thus, `F` commutes with `swap`.

   (b) (**15** points) For sorted lists, `A` and `B`, let

   ```
   my_merge(A, B) -> lists:split(length(A), lists:merge(A, B)).
   ```

   For any monotonic function (i.e. non-decreasing), `F`, show that `F` commutes with `my_merge`.

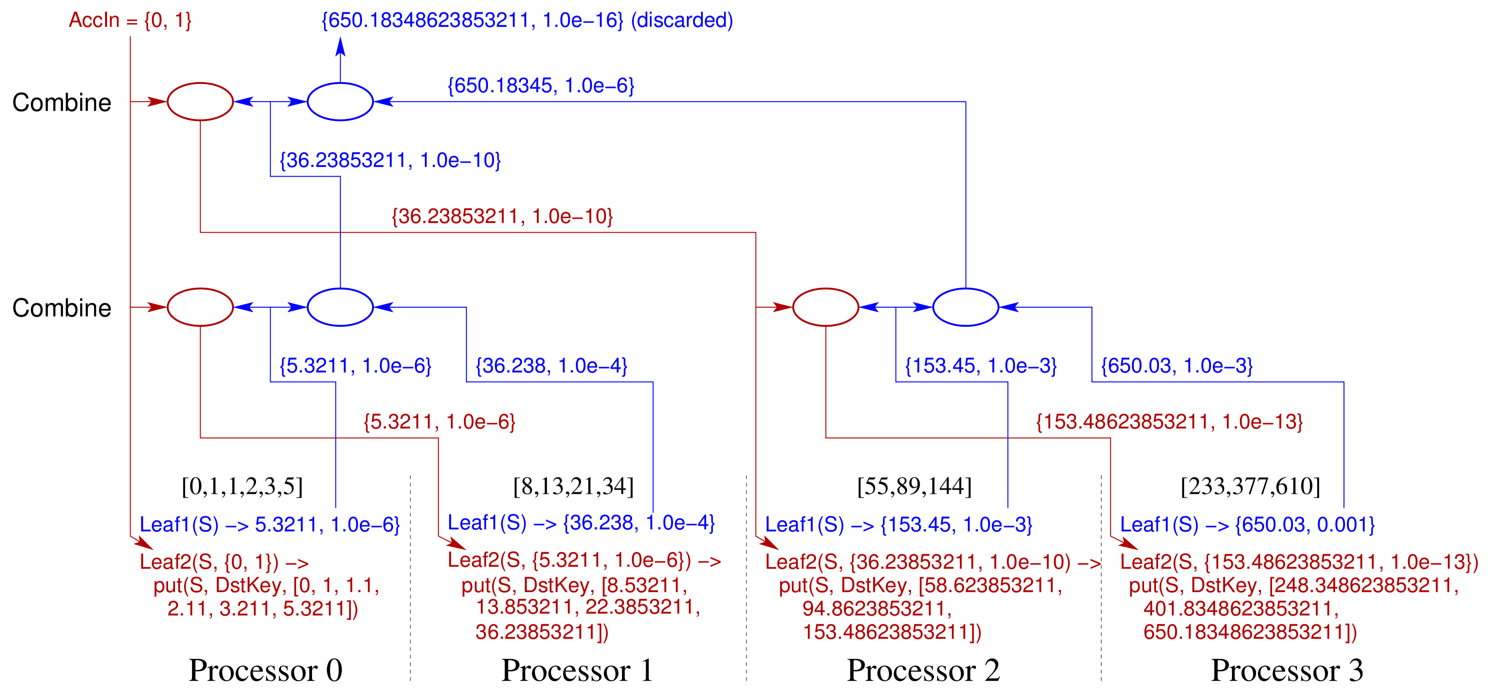
AccIn = {0, 1}
{650.18348623853211, 1.0e−16} (discarded)
Combine
{650.18345, 1.0e−6}
{36.23853211, 1.0e−10}
{36.23853211, 1.0e−10}
Combine
{5.3211, 1.0e−6}
{36.238, 1.0e−4}
{153.45, 1.0e−3}
{650.03, 1.0e−3}
{5.3211, 1.0e−6}
{153.48623853211, 1.0e−13}
[0,1,1,2,3,5]
[8,13,21,34]
[55,89,144]
[233,377,610]
Leaf1(S) –> 5.3211, 1.0e−6}
Leaf2(S, {0, 1}) –> put(S, DstKey, [0, 1, 1.1, 2.11, 3.211, 5.3211])
Leaf1(S) –> {36.238, 1.0e−4}
Leaf2(S, {5.3211, 1.0e−6}) –> put(S, DstKey, [8.53211, 13.853211, 22.3853211, 36.23853211])
Leaf1(S) –> {153.45, 1.0e−3}
Leaf2(S, {36.23853211, 1.0e−10) –> put(S, DstKey, [58.623853211, 94.8623853211, 153.48623853211])
Leaf1(S) –> {650.03, 0.001}
Leaf2(S, {153.48623853211, 1.0e−13}) put(S, DstKey, [248.348623853211, 401.8348623853211, 650.18348623853211])
Processor 0
Processor 1
Processor 2
Processor 3

Figure 1: The scan computation for decaying average

If `L` is a list, I'll write `F(L)` as a shorthand for `map(F, L)`. A simple, and full-credit solution is to observe that `merge(A,B)` produces a list whose elements are the elements of `A` and the elements of `B` in ascending order. Because `F` is monotonic, the elements of `F(merge(A,B))` are in ascending order, and they are the elements of `F(A)` combined with `F(B)`. Therefore, Because `F` is monotonic, the elements of

$$\texttt{F(merge(A,B))} \quad = \quad \texttt{merge(F(A), F(B))}$$

Thus, for any $0 \leq N \leq \mathit{length}(A) + \mathit{length}(B)$, The first N elements of `F(merge(A, B))` are the same as the first N elements of `merge(F(A), F(B))`, and likewise for the last N element. In particular, this applies for $N =$ `length(A)`. Therefore,

$$\texttt{F(my\_merge(A,B))} \quad = \quad \texttt{my\_merge(F(A), F(B))}$$

as required.
A more formal proof could be written by defining `merge` and writing a proof by induction on the length of lists `A` and `B`, but that level of formality is not required in this class.

3. Parallel sorting (**40 points**)
In this problem, you will show that the array Z is sorted into lexicographical order.

(a) (**10 points**) Show that at the end of phase 1, each block B_k has at most one dirty row.

At the end of phase 1, B_k is sorted into ascending or descending lexicographical order. I'll describe the ascending case. The argument for the descending case is equivalent. Let $N_0(A_k)$ denote the number of 0s in A_k. Because B_k is in lexicographical order:

$$\begin{array}{rcll} B_k(i,j) & = & 0 & \text{, if } i < (N_0(A_k) \,\mathsf{div}\, \sqrt{N}) \text{ or } (i = (N_0(A_k) \,\mathsf{div}\, \sqrt{N})) \wedge (j = (N_0(A_k) \bmod \sqrt{N})) \\ B_k(i,j) & = & 1 & \text{, otherwise} \end{array}$$

Thus, the only dirty row of B_k is the one with $i = (N_0(A_k) \,\mathsf{div}\, \sqrt{N})$, and even that row is clean in the case that $j = (N_0(A_k) \bmod \sqrt{N})$. Thus B_k has at most one dirty row.

(b) (**10 points**) Show that at the end of phase 3, all dirty rows of D are contained in a band of height at most $P/2$.

After each processors sorts its columns, all of the columns are sorted. Consider pairs of B blocks of the form B_{2k}, B_{2k+1}. There are $P/2$ such pairs of blocks. Because block B_{2k} was sorted by an even indexed processor, it was sorted int ascending order. Therefore, its dirty row (if any) has its zeros on the left and its ones on the right. Likewise, the dirty row of block B_{2k+1} has ones on the left an zeros on the right. When the columns for these two blocks are sorted, there is at most one dirty row remaining (as with Shear sort). There are $P/2$ such pairs of blocks. Thus, there are at most $P/2$ dirty rows at the end of the column sort. Furthermore, all clean rows of zeros have been moved to the rows with small indices and all clear rows of ones have been moved to the rows with large indices. Therefore, the dirty rows are contiguous and form a band of height at most $P/2$.

(c) (**10 points**) Show that the operations in phases 5 and 7 will not make a clean row dirty.

Blocks are mapped to processors so that every processor sorts blocks with row indices in the order obtained at the end of phase 4. This is ensured because processor 0 does nothing in phase 5. For any block sorted by any processor, rows of clean zeros are already at the low row indices, and they will stay there and clean. Likewise, rows of clean ones are the high row indices and will stay there and clean. The only rows that are affected by these phases are the dirty rows.

(d) (**10 points**) Show that the band of dirty rows of D is contained within a block that is sorted in phase 5 or phase 7.

Let $M = \sqrt{N}/P$ and i be the smallest index of a dirty row. Because the band of dirty rows has height at most $P/2$, the largest index of a dirty row is at most $i + \frac{P}{2} - 1$. Each block that is sorted in phases 5 or 7 consists of M rows. Let $i' = i \bmod M$. Consider two cases:

case $0 \leq i' < M - (P/2)$: This means that rows i through $i + (P/2) - 1$ are all in block $H_{i \,\mathsf{div}\, M}$ and will be sorted in phase 7.

case $M - (P/2) \leq i' < M$: Let $i" = (i \;\mathsf{div}\; M) + 1$. I will show that rows i through $i + (P/2) - 1$ are in blocks $F_{i"}$ and $G_{i"}$. Because $i' \geq M - (P/2)$,

$$\begin{array}{rcl} (i + (P/2)) \;\mathsf{div}\; M & \geq & (i \;\mathsf{div}\; M) + 1 \\ & = & i" \end{array}$$

Thus, row i is in block $F_{i"}$ or later. The block for the shifted location of the last dirty row is $(i + ((P/2) - 1) + (P/2)) \;\mathsf{div}\; M$ We get:

$$\begin{array}{rcll} (i + ((P/2) - 1) + (P/2)) \;\mathsf{div}\; M & = & (i + P - 1) \;\mathsf{div}\; M & \\ & \leq & (i + M - 1) \;\mathsf{div}\; M & , P \leq M \\ & \leq & (i \;\mathsf{div}\; M) + 1 & \\ & = & i" & \end{array}$$

Thus, row $i + (P/2) - 1$ is in block $F_{i"}$ or earlier. However, row i must be in a block that is no later than the block for row $i + (P/2) - 1$. Therefore, rows i through $i + (P/2) - 1$ are all in block $F_{i"}$ and will be be sorted in phase 5.

4. Mutual Exclusion

   (a) (**10 points**) Show that the Bakery algorithm is deadlock free.
   This means that if one or more threads are waiting to get the lock, some thread will eventually get it.

In all of my answers, I'll write Idle(i) to indicate that thread i is in state Idle, and likewise for Ticketing, Spinning, and Critical.
First, I'll show a handy invariant:

$$I_1 \quad = \quad \forall 1 \leq i \leq n.\ \texttt{flag[i]} = \neg\mathsf{Idle}(i)$$

This follows immediately by considering the actions that change the variables that appear in the invariant. If a thread is in state Idle, the only way for it to leave is by entering the Ticketing state which it does by setting `flag[i]` to true. Conversely, the only point at which a thread sets `flag[i]` to false is when it transitions from state Critical to state Idle. The I_1 holds initially when all threads are in state Idle and `flag[i]` is false for all $1 \leq i \leq n$. Thus, I_1 is an invariant.
As stated in the homework problem, every thread repeatedly cycles between states Idle, Ticketing, Spinning, and Critical.
Now, assume that there is at most one thread in the Spinning state. The precedes *relation defined in the homework problem is a total order: for any pair of threads, θ_i, and θ_j, exactly one of $\theta_i \prec \theta_j$, $\theta_j \prec \theta_i$, or $\theta_i = \theta_j$ holds; and if $\theta_i \prec \theta_j$ and $\theta_j \prec \theta_k$ then $\theta_i \prec \theta_k$. A correct solution may assume that precedes is a total order without saying anything about that.*
If there is more than one thread spinning, exactly one is the least by the precedes ordering. Let i be the index of this thread. The next time thread i executes the do-while loop at lines 11. . . 21, either it will exit the loop, or there must be some thread j such that `flag[j]` and $\theta_j \prec \theta_i$. By the choice of i, thread j is not spinning. By invariant I_1, thread j is either Ticketing or Critical. If thread j is Critical, then we've established that a thread has reached its critical section. If thread j is Ticketing then it must eventually reach the Spinning state (it cannot block waiting for other threads). Thus, eventually, either some thread is in its critical region or every thread

with `flag[j]` true must be spinning. In the latter case, the least thread by the precedes relation will enter its critical region the next time it executes the do-while loop.
This shows if at least one thread is spinning, then eventually some thread will be in its critical section.

(b) (**10 points**) Show that the Bakery algorithm issues grants in order.
This means that if thread `i1` has the lock with ticket value `t1`, and the next thread to acquire the lock is thread `i2` with ticket value `t2`, then either `t1` $<$ `t2`, or `t1` $=$ `t2` and `i1` $<$ `i2`.

Once again, I'll use an invariant:

$$I_{2.a} \quad = \quad \forall 1 \leq i,j \leq n.\ (\mathsf{Spin}(i) \wedge \neg\mathsf{Spin}(j)) \;\Rightarrow\; (\theta_j \prec \theta_i)$$

One more critical observations is:

$I_{2.b}$: *Furthermore, if thread j is ticketing when thread i starts its for-loop at lines 13-19, then thread i will continue to spin as long as thread j is ticketing.*

These claims could be written with logic formulas as well, but that would involve formulas with `v`, `ok_to_enter`, and the sets that had been "considered" so far by the two for-loops. CpSc 418 isn't that formal; so, I'll go with the English version. My claim is that $I_1 \wedge I_{2.a} \wedge I_{2.b}$ s an invariant. Here's the proof.

Observation 1: for each $1 \leq i \leq n$, `ticket[i]` increases monotonically over time. This is because when thread i assigns a new value to `ticket[i]` at line 10, the value of `v` must be greater than or equal to the old value of the ticket.

Observation 2: for each $1 \leq i \leq n$, the value assigned to `ticket[i]` is greater than or equal to the value of the largest ticket when thread i entered the Ticketing *state (at line 5), and less than or equal to the value of the largest ticket when thread i updates `ticket[i]` (at line 10). This is a corollary of observation 1.*

I_1 *is an invariant as shown for question 4a.*

$I_{2.a}$: *because tickets only change when entering the spinning state, there are only two actions that we need to consider: thread i transitioning from* Ticketing *to* Spinning*; and thread j transitioning from* Spinning *to* Critical*.*

Thread i transitions from Ticketing *to* Spinning*:*
The value assigned to `ticket[i]` is greater than or equal to the value that `ticket[j]` held when thread i entered the Ticketing *state (Observation 2).*
If `ticket[j]` did not change while thread i was Ticketing*, then `ticket[i]` $\geq$ `ticket[j]`, and $I_{2.a}$ holds.*
If `ticket[j]` changed while thread i was Ticketing*, then thread j entered the* Spinning *state while thread i was* Ticketing*, and clause $I_{2.b}$ shows that thread j must still be spinning. Therefore, $I_{2.a}$ holds.*

Thread j transitions from Spinning *to* Critical*:*
We need to show that for any thread i in the Spinning *state, $\theta_j \prec \theta_i$.*
If thread i was Spinning *for entire time that thread j was executing its for-loop at lines 13…19, then $\theta_j \prec \theta_i$ because `ok_to_enter` would have been set to false for thread j otherwise.*
If thread i entered Spinning *while thread j was executing its for-loop, then we consider the value of `flag[j]` when thread i inspected it at line 15.*
If `flag[i]` was true, then $\theta_j \prec \theta_i$ must have held. This means thread i had already reached the Spinning *state, and it's still there.*
If `flag[i]` was false, thread i was Idle *when thread j checked `flag[j]`. By the assumption that thread i is now* Spinning*, it must have transitioned from* Idle *to* Ticketing *to* Spinning*. By Observation 2, `ticket[i]` $>$ `ticket[j]`. Therefore, $\theta_j \prec \theta_i$, and $I_{2.a}$ holds.*

We've now shown that clause $I_{2.a}$ is maintained by all program actions.

$I_{2.b}$: if thread j is Ticketing *when thread i starts its for-loop, then* `flag[j]` *is true (by I_1), and*

$$\mathsf{Ticketing}(j) \quad \Rightarrow \quad \theta_j \prec \theta_i \quad , I_{2.a}$$

Therefore, thread i will set its `ok_to_enter` *variable to false at lines 15…17 unless thread j leaves* Ticketing*.*

This shows that all clauses of the invariant are maintained by all actions of all threads. Thus, $I_1 \wedge I_{2.a} \wedge I_{2.b}$ is an invariant as claimed.

All three clauses hold in the initial state as well. I showed this for I_1 already. For $I_{2.a}$ and $I_{2.b}$, this follows immediately from all threads starting in the Idle *state.* $\square$

Whew – that invariant was a bit of work! We'll give full credit to any solution that recognizes that no thread can enter the critical region while another thread is ticketing and that spinning threads enter the critical region in order. Most of the complexity of my argument was handling the case where one thread enters the Spinning *state while another* Spinning *thread is checking the flags and tickets of the other threads. A less detailed, more intuitive argument would be fine.*

Back to the original problem. Consider the case where thread i is in the critical region with ticket t_i and thread j is the next to enter with ticket t_j. If thread j is Spinning *before thread j leaves its critical section, then $\theta_i \prec \theta_j$ by $I_{2.a}$. Otherwise, thread j must have been* Idle *when thread i entered its critical section (by $I_{2.b}$). This means that* `ticket[j]` *will be larger than the current value of* `ticket[i]` *when thread j reaches* Spinning *(by Observation 2). In either case, $\theta_i \prec \theta_j$ as required.*

(c) (**10 points**) Show that the Bakery algorithm guarantees mutual exclusion.

I'll prove it by contradiction. Assume that threads i and j (with $i \neq j$) are both in state Critical*. One of them must have been the last to enter state* Critical *– I'll assume that thread j was the last to enter state* Critical*. Because threads enter the critical region in order, we have $\theta_i \prec \theta_j$. This means that* `flag[i]` *must have been false when thread j checked it at line 15. From I_1, thread i_1 was in state* Idle *when thread j was at line 15. From Observation 2,* `ticket[i]` *was set to a value greater than* `ticket[j]` *at line 10. Therefore $\theta_j \prec \theta_i$. A contradiction.*

Because a violation of mutual exclusion implies a contradiction, mutual exclusion cannot be violated.

(d) (**10 points**) Show that if the statement

```
5:        flag[i] = true;
```

is moved until after the for loop at lines 7–9, then mutual exclusion is not guaranteed.

Observation 2 from the invariant proof hints at the counter-example. When the assignment to `flag` *is moved, it is no longer the case that a spinning thread must wait for a ticketing thread to finish before entering the critical region. Because two threads can compute their ticket values concurrently and get the same ticket value, this leads to the mutual exclusion violoation.*

Consider the following execution.

1. *From the intial state, threads 1 and 2 both calculate values for* `v`*. Because no threads have updated their tickets yet, they both get a value of 1.*
2. *Thread 2 continues and sets its flag, executes the for-loop at lines 13…19. Because* `flag[1]` *is false, thread 2's* `ok_to_enter` *variable remains true at the end of the loop. Thread 2 enters its critical region.*
3. *Thread 1 continues and sets its flag. When thread 1 executes the for-loop at lines 13…19 it finds that* `flag[2]` *is true and* `ticket[2]` $=$ `ticket[1]` $= 1$*. Because $1 < 2$, thread 1's* `ok_to_enter` *variable remains true at the end of the loop. Thread 1 enters its critical region.*
4. *Mutual exclusion is now violated.*