CpSc 418 **Homework 4** Due: Nov. 30, 2012, 11:59pm

5% extra credit if solution submitted by 11:59pm on Nov. 27.

Please submit your solution using the handin program as:
cs418 hw4
Your submission should consist of the following files:

`hw4.erl` – Erlang source (ASCII text). All functions requested in this assignment must be exported by this module.

`hw4.c` – C source (ASCII text). All functions requested in this assignment must be exported by this module.

`hw4.txt` – plain, ASCII text, or `hw4.pdf` – PDF.

1. **Reduce and Scan (75 points)**
   Implement each of the operations below using Erlang (with the `wtree` module) **and** MPI (using `MPI_Reduce`, `MPI_Scan`, and `MPI_Op_create`).

   (a) Find element. **(25 points)**

   i. Draw a picture. **(5 points)**
   Given an array, A, of N elements, and a special value, q, define

   $$\{first, last\} \quad = \quad index(q, A)$$

   Where *first* is the smallest integer, $i \in 1, \ldots, N$ such that $A_i = q$, and *last* is the largest integer in $i \in 1, \ldots, N$ such that $A_i = q$. If no element of A is equal to q, then *first* is $+\infty$, and *last* is $-\infty$. For example, if

   $$A \quad = \quad [1, 2, 2, 4, 2, 6, 2, 4, 6, 2, 6, 4, 2, 4, 6, 6]$$

   and $\{first, last\} = index(4, A)$, then $first = 4$ and then $last = 14$. Draw a diagram that shows how this computation can be performed using a reduce operation with four processes where each process initially holds four consecutive elements of A. You can draw your diagram neatly by hand, scan it, and include it in `hw4.pdf`, or you can draw it using a drawing program of your choice, export it as a PDF file, and include it in `hw4.pdf`.

   ii. Erlang version: (**10 points**)
   `hw4:index(W, KeyA, Q) -> {First, Last}`

   - `W` is a worker pool.
   - `KeyA` is the key for the source list.
   - `Q` is value to search for in the list.

   `First` and `Last` are set to the indices of the first and last occurrences of `Q` in the distributed list associated with `KeyA`. If `Q` does not occur in this list, then the atom `undefined` is returned.

   iii. MPI version: (**10 points**)
   `void first_last(int *src, int count, int q, int *dst, int root, MPI_comm comm)`

   - `src` is a pointer to an array of `count` elements.
   - `q` is the special value to search for.
   - `dst` is a pointer to an array of 2 elements.
   - `comm` is an MPI communicator

   `dst[0]` gets the index of the first occurence of `q` in `src`, and `dst[1]` gets the index of the last occurence of `q` in `src`. If `q` does not occur in `src`, both `dst[0]` and `dst[1]` are set to -1.

(b) Rolling average. **(25 points)**

i. Draw a picture. **(5 points)**
Given an array, A, of N elements, the M-element rolling average of A is the array B where

$$B_k \;=\; \frac{1}{M} \sum_{i=\max(1,k-(M-1))}^{k} A_i$$

For example, if

$$A \;=\; [1, 4, 9, 16, 25, 36, 49, 64],$$

and B is the 3-element rolling average of A, then

$$B \;=\; [1/3, 5/3, 14/3, 29/3, 50/3, 77/3, 110/3, 149/3].$$

Draw a diagram that shows how this computation can be performed using a scan operation with four processes where each process initially holds two consecutive elements of A, and each process will hold two elements of B at the end of the reduce. You can draw your diagram neatly by hand, scan it, and include it in `hw4.pdf`, or you can draw it using a drawing program of your choice, export it as a PDF file, and include it in `hw4.pdf`.

ii. Erlang version: (**10 points**)
`hw4:rolling_average(W, KeyA, KeyB, M)`

- `W` is a worker pool.
- `KeyA` is the key for the source list.
- `KeyB` is the key for the result list.
- `M` a positive integer.

Compute the `M`-element rolling average of the distributed list associated with `KeyA` and store it as a distributed list associated with `KeyB`.

iii. MPI version: (**10 points**)
`void rolling_average(double *src, double *dst, int count, int m, MPI_comm comm)`

- `src` is a pointer to an array of `count` elements.
- `dst` is a pointer to an array of `count` elements.
- `m` is a positive integer.
- `comm` is an MPI communicator.

Compute the `m`-element rolling average of the elements of `src` and store the result in `dst`.

(c) Credit Card balance (**25 points**) Consider a credit-card account that is opened on day 0 with a balance of \$0.00. Let T be a list of transactions, where each transaction is a tuple (d, v); d is an integer, the date on which the transaction took place; and v is the amount of the transaction. If v is positive, it is a *purchase*, which increases the balance owed on the account. If v is negative, it is a *payment*, which decreases the balance owed. For any positive integer, n, we compute the balance on day n in two steps:

$$\begin{aligned}
\text{balance}(0) &= 0 \\
\text{balance}(n) &= (1+r) * \text{balance}(n-1) + \sum_{(n,a)\in T} a, && \text{the "true" balance} \\
\text{acctbal}(n) &= \text{round}(\text{balance}(d), 0.01), && \text{rounded to the nearest penny}
\end{aligned}$$

where r is the daily interest rate, and $\text{round}(x, p)$ rounds x to the nearest multiple of p. Note that this credit card *pays* interest if you've got a negative balance – don't expect this for a reall credit card.

As an example, let

$$T = [(1, 17.42), (2, 5.00), (3, -20.00), (4, 1.00), (4, 12.34), (6, -20.00), (7, 10.00), (10, 9.99)]$$

and $r = 0.02$ (a usurious rate, even for a credit card!). Letting B be the true balance following each transaction, and A be the account balance. We get:

$$B = [17.42,\ 22.7684,\ 3.223768,\ 4.28824336,\ 16.62824336,\ -2.69997561,\ 7.24602488,\ 17.67953957]$$
$$A = [17.42,\ 22.77,\ 3.22,\ 4.29,\ 16.63,\ -2.70,\ 7.25,\ 17.68]$$

i. Draw a picture. **(5 points)**
Draw a diagram that shows how the computation of the account balance after each transaction can be performed using a scan operation with four processes where each process initially holds two consecutive elements of T, and each process will hold two elements of B at the end of the scan. You can draw your diagram neatly by hand, scan it, and include it in `hw4.pdf`, or you can draw it using a drawing program of your choice, export it as a PDF file, and include it in `hw4.pdf`.

ii. Erlang version: (**10 points**)
`hw4:balance(W, KeyT, KeyB, Rate)`

- `W` is a worker pool.
- `KeyT` is the key for the distributed list of transactions. This list is sorted in ascending order of transaction date, and that each transaction is of the form `{Date, Amount}`.
- `KeyB` is the key for the result list.
- `Rate` is the daily interest rate (i.e. r in the problem statement).

Compute the after transaction balances for the transactions stored as the distributed list associated with `KeyT` and store the resulta as a distributed list associated with `KeyB`. Of course, you can use erlang's floating-point arithmetic and will get a bit of floating-point round-off when computing the "true" balance.

iii. MPI version: (**10 points**)
`void balance(struct Transaction *tr, double *dst, int count, int m, MPI_comm comm)`

- `tr` is a pointer to an array of `count` transactions where

```
struct Transaction {
    int date;
    double amount;
}
```

- `dst` is a pointer to an array of `count` elements.
- `comm` is an MPI communicator

Compute after-transaction balances for the transactions stored as `src` and store the result in `dst`. Of course, you can use double-precision arithmetic for your calculations and incur a bit of round-off error when computing the "true" balance.

2. Test-and-set **(30 points)**
In class and on homework 3, we considered mutual exclusion algorithms for which the only atomic (i.e. indivisible) operations were memory reads and memory writes. Modern machines provide other instructions, where a simple one is `tas` ("test-and-set"). In particular,

```
tas $Rdst, $Rptr
```

reads the memory location at the address given by register `$Rptr`, stores the value read in register `$Rdst`, and sets the content of the memory location to `1`.

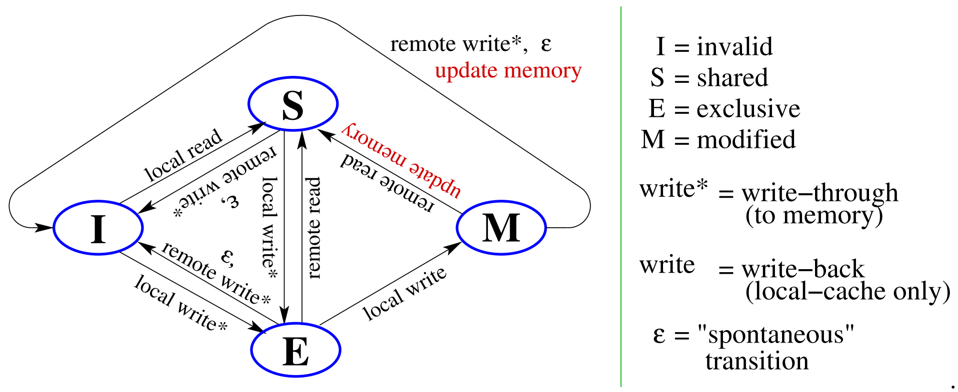

Figure 1: The MESI Cache-Coherence Protocol

(a) Using `tas` for mutual exclusion (**10 points**)
Show that the following code guarantees mutual exclusion for N threads indicated by their respective program counters, $\texttt{PC}_0, \ldots, \texttt{PC}_N$.

```
initially:  flag = 0;
PC_i=0:  while(true) {
PC_i=1:    non-critical code
PC_i=2:    while(tas(&flag));
PC_i=3:    critical section
PC_i=4:    flag= false;
PC_i=5:  }
```

where `tas(&flag)` performs a test-and-set on address of `flag` and returns the value that had been stored in `flag`. Following the Peril-L convention, `flag` is underlined in the code above to indicate that it is a global variable.
To show that this code guarantees mutual exclusion, let

$$ncrit \quad = \quad |\{i \mid \texttt{PC}_i \in \{3, 4\}\}|$$

Now show that I_N is an invariant of the program where:

$$I_N \quad = \quad (\underline{\texttt{flag}} = (ncrit = 1)) \wedge (ncrit \leq 1)$$

Finally, write a *short* explanation of why I_N implies that at most one thread is in its critical section at any given time.

(b) Test-and-Set with MESI (**10 points**)
Figure 1 shows the MESI protocol from the October 4 lecture. Show that this protocol is insufficient for implementing the `tas` instruction. In particular, consider two threads that try to perform a test-and-set at the same time. Assume that thread 0 performs the first read. Show that there are no states that their caches can be in after this read that guarantees that thread 0 performs its write before thread 1 performs its read.

(c) Extending MESI (**10 points**)
Now, add a fifth state to the MESI protocol that we will label **T** in diagrams in honour of the test-and-set instruction. We will add a new operation called "read-with-intent-to-write" that is used for the read operation of a test-and set, and brings the cache into the **T state**. Draw the state-diagram for the five-state protocol that supports test-and-set. Your diagram should have states **M**, **E**, **S**, **I**, and **T**. Show the transitions for local-read, local write, remote-read, remote-write, local-read-with-intent-to-write, remote-read-with-intent-to-write, and ϵ. To make the transition labels legible, you may use the following abbreviations:

lr: read by the local processor
lw: write by the local processor
lx: read-with-intent-to-write by the local processor
rr: read by another (i.e. remote) processor
rw: write by another (i.e. remote) processor
rx: read-with-intent-to-write by another (i.e. remote) processor
ϵ: Spontanous transition (always allowed)