Homework 8

Handout 26
CS242: Autumn 2003
05 December

Solutions

1. .... Fairness

(a) A fair implementation might cause either and underflow or an overflow. Fairness simply requires that the nondeterministic choice eventually selects each action. Since there is no time-bound on how long it might take before the nodeterministic choice selects a particular action, it is possible to either overflow or underflow. The same can be said of an unfair implementation since in that case, there is no requirement to choose a particular alternative eventually.

(b) A fair implementation requires that eventually every alternative is selected. Thus, eventually the `go := false` line will be executed meaning the incrementing of `n` will eventually stop.

Under an unfair implementation there is no requirement to select a particular option and so the code might increment `n` forever.

(c) Simpler on a single-processor since you don't need to synchronize the multiple processors. "Everything is easier on a single-processor language implementation." —Adam Barth

2. .... Actor computing

(a) A sequence number can be added to each task. When B first receives a task, it can check the sequence number. If this is not the first task, or the next one to be processed, then B should store the task and process it later in order.

(b) One protocol, similar to TCP, is for B to acknowledge each received message (by sequence number). To avoid flooding the communication mechanism, A can send a few messages, then wait for acknowledgements to arrive before proceeding with additional messages. If A receives acknowledgements for several messages with sequence numbers greater than n, then A can suspect that message n is delayed and resend it.

(c) The *I'm done* message should have a sequence number too.

3. .... Message Passing

	Synchronous		Asynchronous	
	Ordered	Unordered	Ordered	Unordered
Buffered			TCP	Actor/Java
Unbuffered	CML			UDP

(a) Buffered synchronous makes no sense because you don't need to buffer things if everything is synched up.

(b) Synchronous unordered makes no sense because if you are synched you cannot receive messages out of order.

(c) Asynchronous unbuffered message-passing cannot be ordered so the ordered asynched unbuffered version makes no sense.

4. CML Thread Implementation

(a) The program outputs numbers from 0-100 in two columns, one number in each row. Even numbers are printed in the first column and odd numbers in the second. The number are produced by two threads "skipby(0,2)" and "skipby(1,2)", which yield to each other at psuedo-random intervals.

(b)
```
fun go() = (fork (fn () => skipby(0,2));
            fork (fn () => skipby(1,2));
            fork (fn () => skipby(3,3));
            wait 300)
```

(c) Yes, because we have non-preemptive scheduling and the threads yield to each other at pseudo-random (i.e. deterministic) intervals.

(d) Programmers can now determine exactly when control leaves the current thread, so there is less chance of unexpected interleavings that can lead to race conditions.

5. Separate read and write synchronization

The solution requires only one lock. You keep track by maintaining a count of readers that keeps track of how many people are currently reading the object. This counter is set by the lock on the object. When the reader tries to read it acquires the lock and increments the readers. When done it releases the lock and then can read data without a lock. When the read is done, the value of the counter is decremented using hte lock. Writing can only be done if the readers counter is zero.

6. Java memory model

(a) Since the two threads can occur in any order with respect to each other, you get two different constraint graphs, one for thread one and the other for thread 2.

$$\begin{array}{ccccccccccc}
\text{read x} & \rightarrow & \text{load x} & \rightarrow & \text{use x} & \rightarrow & \text{assign a} & \rightarrow & \text{store a} & \rightarrow & \text{write a} \\
 & & & & & & \downarrow & & & & \\
 & & & & & & \text{assign y} & & & & \\
 & & & & & & \downarrow & & & & \\
 & & & & & & \text{store y} & & & & \\
 & & & & & & \downarrow & & & & \\
 & & & & & & \text{write y} & & & &
\end{array}$$

$$\begin{array}{ccccccccccc}
\text{read y} & \rightarrow & \text{load y} & \rightarrow & \text{use y} & \rightarrow & \text{assign b} & \rightarrow & \text{store b} & \rightarrow & \text{write b} \\
 & & & & & & \downarrow & & & & \\
 & & & & & & \text{assign x} & & & & \\
 & & & & & & \downarrow & & & & \\
 & & & & & & \text{store x} & & & & \\
 & & & & & & \downarrow & & & & \\
 & & & & & & \text{write x} & & & &
\end{array}$$

(b) Here for the prescient stores, the store need not wait for its assign.

$$\begin{array}{ccccccccccc}
\text{read x} & \rightarrow & \text{load x} & \rightarrow & \text{use x} & \rightarrow & \text{assign a} & & \text{store a} & \rightarrow & \text{write a} \\
 & & & & & & \downarrow & & & & \\
 & & & & & & \text{assign y} & & & & \\
 & & & & & & & & & & \\
 & & & & & & \text{store y} & & & & \\
 & & & & & & \downarrow & & & & \\
 & & & & & & \text{write y} & & & &
\end{array}$$

$$
\begin{tikzcd}
\text{read y} \arrow[r] & \text{load y} \arrow[r] & \text{use y} \arrow[r] & \text{assign b} \arrow[d] & \text{store b} \arrow[r] & \text{write b} \\
 & & & \text{assign x} & & \\
 & & & \text{store x} \arrow[d] & & \\
 & & & \text{write x} & &
\end{tikzcd}
$$