Homework 8

Due 03 December

Handout 22
CS242: Autumn 2003
26 November

____
(Print your name)

Prob	#1	# 2	# 3	# 4	# 5	# 6	Total
✓+							
✓							
✓-							
0							

Note: Please either write your answers legibly in the space provided or attach a typewritten solution. Make sure that your answers are stapled together.

Reading

1. Read chapter 14 on concurrency.

Problems

1. .... Fairness

The guarded-command looping construct

```
do
    Condition ⇒ Command
    ...
    Condition ⇒ Command
od
```

involves nondeterministic choice, as explained in the text. An important theoretical concept related to potentially nonterminating nondeterministic computation is *fairness*. If a loop repeats indefinitely, then fair nondeterministic choice must eventually select each command whose guard is true. For example, in the loop

```
do
    true ⇒ x := x+1
    true ⇒ x := x-1
od
```

both commands have guards that are always true. It would be *unfair* to execute `x := x+1` repeatedly without ever executing `x := x-1`. Most language implementations are designed to provide fairness, usually by providing a bounded form. For example, if there are n guarded commands, then the implementation may guarantee that each enabled command will be executed at least once in every $2n$ or $3n$ times through the loop. Since the number $2n$ or $3n$ is implementation dependent, though, programmers should only assume that each command with a true guard will eventually be executed.

(a) Suppose that an integer variable x can only contain an integer value with absolute value less than INTMAX. Will the `do ... od` loop above cause overflow or underflow under a fair implementation? What about an implementation that is not fair?

(b) What property of the following loop is true under a fair implementation but false under an unfair implementation?

```
go  := true;
n := 0;
do
    go => n := n+1
    go => g := false
od
```

(c) Is fairness easier to provide on a single-processor language implementation or a multiprocessor? Discuss briefly.

2. .... Actor computing

The Actor mail system provides asynchronous buffered communication and does not guarantee that messages (*tasks* in Actor terminology) are delivered in the order they are sent. Suppose actor A sends tasks $t_1, t_2, t_3, \ldots$ to actor B and we want actor B to process tasks in the order A sends them.

(a) What extra information could be added to each task so that B can tell whether it receives a task out of order? What should B do with a task when it first receives it, before actually performing the computation associated with the task?

(b) Since the Actor model does not impose any constraints on how soon a task must be delivered, a task could be delayed an arbitrary amount of time. For example, suppose actor A sends tasks $t_1, t_2, t_3, \ldots, t_{100}$ and actor B receives the tasks $t_1, t_3, \ldots, t_{50}$ without receiving task t_2. Since B would like to proceed with some of these tasks, it makes sense for B to ask

A to resend task t_2. Describe a protocol for A and B that will add resend requests to the approach you described in part (a) of this problem.

(c) Suppose B wants to do a final action when A has finished sending tasks to B. How can A notify B when A is done? Be sure to consider the fact that if A sends *I'm done* to B after sending task t_{100}, the *I'm done* message may arrive before t_{100}.

3. .... Message Passing

There are eight message-passing combinations involving synchronization, buffering, and message order, as shown in the following table.

	Synchronous		Asynchronous	
	Ordered	Unordered	Ordered	Unordered
Buffered				
Unbuffered				

For each combination, give a programming language that uses this combination and explain why the combination makes some sense, or explain why you think the combination is either meaningless or too weak for useful concurrent programming.

4. .... CML Thread Implementation

The single-processor implementation of Concurrent ML uses coroutines that are implemented using continuations. You may recall from earlier homework (Problem 8 in Homework 4) that a coroutine is a function that can return ("yield") a value and still retain its execution state so that it can later pick up where it left off later.

The code below, written by Andrew Appel, is also available in the homework directory of the CS242 web site.

```
    structure Q = Queue    (* Use standard Queue package         *)
    open SMLofNJ.Cont      (* Use module providing continuations *)

(* The "ready queue" is the queue of threads ready to run       *)
    val rdyQ : unit cont Q.queue = Q.mkQueue()

    fun exit () = throw (Q.dequeue rdyQ) ()
```

```
fun fork f =
  callcc (fn parent =>
   (Q.enqueue (rdyQ, parent) ;
    (f ()) handle _ => () ;
    exit ()))

fun yield () =
  callcc (fn k =>
    (Q.enqueue (rdyQ, k) ; exit ()))
```

The thread package provides three functions: `fork`, `yield`, and `exit`. The `fork` function forks off a new thread and suspends the current thread. In doing so, the function puts the current thread in the *ready queue* and transfers control to the new thread. The `yield` function suspends the current thread and yields control to the next thread in the ready queue. The `exit` function exits the current thread. As you'll notice in the code, we use `callcc` to save the state of the current thread and use `throw` to transfer control.

The SML code file in the handouts directory also contains a test program. The `go` function forks two processes, each defined using the `skipby` function.

```
fun rand i = (i * 109) mod 7 > 3
fun spaces 0 = () | spaces i = (print " "; spaces(i-1))

fun skipby(n,k) =
let fun loop i = if i>100 then ()
               else
               (spaces(10*n);
                print (Int.toString i ^ "\n");
        if rand i then yield() else ();
        loop(i+k))
 in loop n
end

fun wait 0 = ()
|   wait n = (yield(); wait(n-1))

fun go() = (fork (fn () => skipby(0,2));
            fork (fn () => skipby(1,2));
            wait 200)
```

Your tasks are the following:

(a) Run `go()` and look at the output. Explain the output and explain how the program produces it.

(b) Modify `go` by adding a third thread that prints out multiples of three (0, 3, 6, 9...) in a third column. Print your program and the output and submit it with your homework.

(c) Is the output of your program deterministic? Why?

(d) "Preemptive scheduling" refers to a situation when a thread can be stopped before it calls `yield`; preemption does not occur with the thread implementation used in this problem. Why does nonpreemptive scheduling produce fewer race conditions? Explain.

5. .... Separate read and write synchronization

For many data structures, it is possible to allow multiple reads to occur in parallel, but reads cannot be safely performed while a write in progress and it is not safe to allow multiple writes simultaneously. Rewrite the Java `LinkedCell` class given in this chapter to allow multiple simultaneous reads, but prevent reads and writes while a write is in progress. You may want to use more than one lock. For example, you could assume objects called `ReadLock` and `WriteLock` and use `synchronized` statements involving these two objects. Explain your approach and why it works.

Attach a **typewritten** solution.

6. .... Java memory model

This program with two threads is discussed in the text.

```
x = 0; y = 0;
Thread 1:  a = x; y = 1;
Thread 2:  b = y; x = 1;
```

Draw a box-and-arrow illustration showing the order constraints on the memory actions (*read, load, use, assign, store, write*) associated with the four assignments that appear in the two threads. (You do not need to show these actions for the two assginments setting `x` and `y` to 0.)

(a) Without prescient stores.

(b) With prescient stores.