Handout 9
CS242: Autumn 2004
24 November

Study Questions

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. .... Concurrent ML events

The Concurrent ML `recv` function can be defined from `recvEvt` by

```
fun recv(ch) = sync (recvEvt (ch));
```

as shown in the text. Give a similar definition of `send` using `sendEvt` and explain your definition in a few sentences.

5. .... 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.

6. .... Java ConcurrentHashMap

ConcurrentHashMap allows concurrent access to a hash table with minimal locking. Without going into all of the details, this problem asks you to think about some aspects of the design. The relatively tricky design of ConcurrentHashMap is intended to allow read access to parts of the data structure that may be written concurrently, in a way that causes the read to try again if the state seems inconsistent or incomplete.

The hash table implementation uses a resizable array of hash buckets, each consisting of a linked list of Map.Entry elements. As with other hash table implementations you may be familiar with, the hashcode of an entry determines its bucket; when several entries hash to the same bucket, they are placed in a linked list. Here is part of the definition of a Map.Entry class.

```
    protected static class Entry implements Map.Entry {
    protected final Object key;
    protected volatile Object value;
    protected final int hash;
    protected final Entry next;
    ...
}
```

The volatile modifier asks the Java Virtual Machine to order accesses to the shared copy of the variable so that its most current value is always read.

Instead of a single lock governing access to the entire collection, ConcurrentHashMap uses a lock over each segment of buckets. The linked list used by ConcurrentHashMap is designed so that the implementation can detect that its view of the list is inconsistent or stale. If it detects that its view is inconsistent or stale, or simply does not find the entry it is looking for, it then synchronizes on the appropriate bucket lock and searches the chain again.

(a) What does the use of `final` in the `Map.Enty` class tell you about the way a linked list of Map.Entry elements may change when a ConcurrentHashMap is updated? (Don't think too deeply - the purpose of this question is to point out something that is important for later questions.)

(b) There is one straightforward way to remove an item from a linked list of `Map.Enty` objects. Describe the steps involved in removing the second item from a list. Assume the pointer to the beginning of the list is mutable. (*Hint:* Pure Lisp.)

(c) A problem with the Java Memory Model (now being updated) is that a thread may access part of an object before the thread running the constructor completes. The implementation of ConcurrentHashMap uses default values for final fields that are zero or null; these values are overwritten by the constructor. This lets a concurrent thread test whether the constructor has set meaningful values in final fields. Assume that a thread searching the list for a data value does not initially synchronize on the lock for this list. (The reason not to lock the list is to allow greater concurrency.) What should the thread do if it sees the default values?

(d) The ConcurrentHashMap retrieval operations first find the head pointer for the desired bucket. This is done without locking, so the value of the head pointer could be stale. The operation then traverses the linked list representing the bucket starting from the head pointer, without acquiring the lock for that bucket. If the operation does not find the value it is looking for, it acquires the lock for the bucket and tries again. Here is the code, in case you want to look at it; you may be able to answer the question without reading the code.

```
int hash = hash(key);  // throws null pointer exception if key is null

  // Try first without locking...
  Entry[] tab = table;
  int index = hash & (tab.length - 1);
  Entry first = tab[index];
```

```
    Entry e;

    for (e = first; e != null; e = e.next) {
      if (e.hash == hash && eq(key, e.key)) {
        Object value = e.value;
        // null values means that the element has been removed
        if (value != null)
          return value;
        else
          break;
      }
    }

    // Recheck under synch if key apparently not there or interference
    Segment seg = segments[hash & SEGMENT_MASK];
    synchronized(seg) {
      tab = table;
      index = hash & (tab.length - 1);
      Entry newFirst = tab[index];
      if (e != null || first != newFirst) {
        for (e = newFirst; e != null; e = e.next) {
          if (e.hash == hash && eq(key, e.key))
            return e.value;
        }
      }
      return null;
    }
  }
```

Why do the second traversal? What advantage does this two-pass algorithm have over simply locking the linked list the first time and doing only one traversal?

(e) Removing an element from a ConcurrentHashmap poses several problems. First, because a thread could see stale values for the link pointers in a hash chain, simply removing an element from the chain would not be sufficient to ensure that other threads will not continue to see the removed value when performing a lookup. However, there's a clue to how the implementation works in the `get` code above – the appropriate Entry object is found and its value field is set to null. Then the algorithm you may have discovered in part (b) is used. Explain why declaring the `value` field as `volatile` is useful here.