Homework 9

Due Never

Handout 24
CS242: Autumn 2002
3 December

Reading

1. Read Chapter 14

Problems

1. 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}.

2. Concurrent access to objects

This question asks about synchronizing methods for stack and queue objects.

(a) Bounded stacks can be defined as objects, each containing an array of up to n items. Here is pseudo-code for one form of stack class.

```
class Stack
   private
      contents : array[1..n] of int
      top : int
   constructor
      stack () =   top := 0
   public
      push (x:int) : unit =
          if top < n then
              top :=  top + 1;
              contents[top] := x
          else raise stack_full;
      pop ( )  : int  =
          if top > 0 then
              top :=  top - 1:
              return contents[top+1]
```

```
          else raise stack_empty;
end Stack
```

If stacks are going to be used in a concurrent programming language, what problem might occur if two threads invoke push and pop simultaneously? Explain.

(b) How would you solve this problem using Java concurrency concepts? Explain.

(c) Suppose that instead of stacks, we have queues:

```
class Queue
   private
      contents : array[1..n] of int
      front, back : int
   constructor
      queue() =  front := back := 1
   public
      insert (x:int) : unit =
          if back+1 mod n !=  front then
             back :=  back+1 mod n;
             contents[back] := x
          else raise queue_full;
      remove ( )  : int  =
          if front != back then
             front :=  front+1 mod n;
             return contents[front]
          else raise queue_empty;
end Queue
```

Suppose that five elements have been inserted into a queue object and none of them have been removed. Do we have the same concurrency problem as we did with push and pop when one thread invokes insert and another thread simultaneously invokes remove? Assume that n is 10. Explain.

3. .... Java syncronized objects

This question asks about the following Java implementation of a bounded buffer. A bounded buffer is a FIFO data structure that can be accessed by multiple threads.

```
class BoundedBuffer {
   // designed for multiple producer threads and
   // multiple consumer threads
   protected int numSlots = 0;
   protected int[] buffer = null;
   protected int putIn = 0, takeOut = 0;
   protected int count = 0;

   public BoundedBuffer(int numSlots) {
      if (numSlots <= 0)
         throw new IllegalArgumentException("numSlots <= 0");
      this.numSlots = numSlots;
      buffer = new int[numSlots];
   }
   public synchronized void put(int value)
         throws InterruptedException {
      while (count == numSlots) wait();
      buffer[putIn] = value;
      putIn = (putIn + 1) % numSlots;
```

```
        count++;
        notifyAll();
    }
    public synchronized int get()
          throws InterruptedException {
        int value;
        while (count == 0) wait();
        value = buffer[takeOut];
        takeOut = (takeOut + 1) % numSlots;
        count--;
        notifyAll();
        return value;
    }
}
```

(a) What is the purpose of `while (count == numSlots) wait()` in `put`?

(b) What does `notifyAll()` do in this code?

(c) Describe one way that the buffer would fail to work properly if all synchronization code is removed from `put`.

(d) Suppose a programmer wants to alter this implementation so that one thread can call `put` at the same time as another calls `get`. This causes a problem in some situation but not in others. Assume that some locking may be done at entry to `put` and `get` to make sure the concurrent-execution test is satisfied. You may also assume that increment or decrement of an integer variable is atomic and that only one call to `get` and one call to `put` may be executed at any given time. What test involving `putIn` and `takeOut` can be used to decide whether `put` and `get` can proceed concurrently?

(e) The changes in part (d) will improve performance of the buffer. List one reason that leads to this performance advantage. Despite this win, some programmers may choose to use the original method anyway. List one reason why they might make this choice.

4. .... Resources and Java Garbage Collection

Suppose we are writing an application that uses a video camera which is attached to the computer. Our application, written in Java, has multiple threads, which means that separate parts of the application may run concurrently. The camera is a shared resource that can only be used by one thread at a time and our multithreaded application may try to use the camera concurrently from multiple threads.

The camera library (provided by the camera manufacturer) contains methods that will ensure that only one thread can use the camera at a time. These methods are called:

```
camera.AcquireCamera()
camera.ReleaseCamera()
```

A thread that tries to acquire the camera while another object has acquired it will be blocked until `camera.ReleaseCamera()` has been called. When a thread is blocked, it simply stops without executing any further commands until it becomes unblocked.

You decide to structure your code so that you create a `MyCamera` object whenever a thread wants to use the camera, and you “delete” the object (by leaving the scope that contains a pointer to it) when that thread is done with the camera. The object calls `camera.AcquireCamera()` in the constructor and calls `camera.ReleaseCamera()` in the `finalize` method, as follows:

```
import camera   // imports the camera library
```

```
class MyCamera {
    ...
    MyCamera() {
        ...
        camera.AcquireCamera();
        ...
    }
    ...            // (other methods that use the camera go here)
    finalize() {
        ...
        camera.ReleaseCamera();
        ...
    }
}
```

Here is some sample code that would use the `Mycamera` object:

```
{
...
MyCamera c = new MyCamera();

... // (code that uses the camera)
}   // end of scope so object is no longer reachable
```

In this question, we will say that a *deadlock* occurs if all threads are waiting to acquire the camera, but `camera.ReleaseCamera` is never called.

(a) When does `camera.ReleaseCamera` actually get called?

(b) This code can cause a deadlock situation in some Java implementations. Explain how.

(c) Does calling the garbage collector using `Runtime.getRuntime().gc()` after leaving the scope where the camera is reachable solve this problem?

(d) How can you fix this problem by modifying your program (without trying to force garbage collection or using `synchronized`) so deadlock will not occur?

(e) Suppose you had a multi-threaded Java implementation with the garbage collector running concurrently as a separate thread. Assume the garbage collector is always running, but it may run slowly in the background if the program is active. This will eventually garbage collect every unreachable object, but not necessarily as soon as it becomes unreachable. Does deadlock, as defined above, occur (in the original code above) in this implementation? Why or why not?

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.