CSE 451 Homework Assignment #3:
Monitors and Deadlock
Due Date: Thursday, October 28, in section

1. Does the following implement Mesa-style condition variables? Why or why not? (Hint: would this work for your solution to #2?)

```
class Condition {
     Semaphore *semaphore;
     int numWaiting;
}

Condition::Condition() {
     semaphore = new Semaphore(0);
     numWaiting = 0;
}

void Condition::Wait (Lock *lock) {
     numWaiting++;
     lock->Release();
     semaphore->P();
     lock->Acquire();
}

void Condition::Signal(Lock *lock) {
     if (numWaiting > 0) {
          numWaiting--;
          semaphore->V();
     }
}
```

2. You have been hired by Caltrans to synchronize traffic over a narrow light-duty bridge on a public highway. Traffic may only cross the bridge in one direction at a time, and if there are ever more than 3 vehicles on the bridge at one time, it will collapse under their weight. In this system, each car is represented by one thread, which executes the procedure *OneVehicle* when it arrives at the bridge:

```
enum Direction (East, West);
OneVehicle(Direction direc)
{
  ArriveBridge(direc);
  CrossBridge(direc);
  ExitBridge(direc);
}
```

In the code above, *direc* gives the direction in which the vehicle will cross the bridge.

(a) Write the procedures *ArriveBridge* and *ExitBridge* (the *CrossBridge* procedure should just print out a debug message), using locks and Mesa-style condition variables.

ArriveBridge must not return until it safe for the car to cross the bridge in the given direction (it must guarantee that there will be no head-on collisions or bridge collapses). *ExitBridge* is called to indicate that the caller has finished crossing the bridge; *ExitBridge* should take steps to let additional cars cross the bridge. This is a lightly-travelled rural bridge, so you do not need to guarantee fairness or freedom from starvation.

(b) In your solution, if a car arrives while traffic is currently moving in its direction of travel across the bridge, but there is another car already waiting to cross in the opposite direction, will the new arrival cross *before* the car waiting on the other side, *after* the car on the other side, or is it impossible to say? Explain briefly.

3. Suppose the too much milk solution presented in class is run on a multiprocessor implementing weak ordering with respect to memory references. Recall that with weak ordering, the effects of operations performed by one CPU can be seen in a different order by other CPUs.

(a) Will the solution still work? If so, explain why. If not, give a scenario where the code presented in class will fail.
(b) Every multiprocessor supporting weak ordering implements a "fence" instruction. When a CPU executes a fence, the instruction doesn't complete until all operations performed by a CPU before the fence are visible to every other CPU. Can fence instructions be added to the too much milk solution to make it work even on a weakly ordered system? If so, explain how. If not, explain why not.

4. Below are listed several ways of handling deadlock.

(a) One criteria to use in evaluating different approaches to deadlock is which permits the greatest concurrency -- in other words, which allows the most threads to make progress without waiting when there isn't deadlock. Give a rank order from 1 to 6 for each of the ways of handling deadlock listed below, where 1 allows the greatest degree of concurrency, and 6 allows the least concurrency; if two approaches offer roughly equal concurrency, indicate that.

(b) Another criteria to use in evaluating deadlock algorithms is efficiency -- in other words, which requires the least CPU overhead. Rank order the approaches according to efficiency, with 1 being the most efficient, assuming that encountering deadlock is a very rare event. Again, indicate if two approaches are equally efficient. Does your ordering change if deadlocks happen frequently?

(i) banker's algorithm
(ii) detect deadlock and kill thread, releasing all its resources
(iii) reserve all resources in advance
(iv) restart thread and release all resources if thread needs to wait
(v) resource ordering
(vi) detect deadlock and roll back thread's actions