CMPSCI 377 Midterm (Sample)

You will have 90 minutes to work on this exam, which is closed book. There are 4 problems on 9 pages.

Please write your name on EVERY page.

You are to abide by the University of Massachusetts Academic Honesty Policy.

Name:_____

Problem 1	_____ out of 25
Problem 2	_____ out of 25
Problem 3	_____ out of 25
Problem 4	_____ out of 25
Total	**_____ out of 100**

Name _____

1. Analyzing a Concurrent Program (approx. 20 minutes)

The following is an implementation of an atomic transfer function. `transfer` should atomically dequeue an item from one queue and enqueue it on another. By atomically, we mean that there must be no interval of time during which an external thread can determine that an item has been removed from one queue but not yet placed on another (assuming the external thread locks a queue before examining it). `transfer` must complete in a finite amount of time, and must allow multiple transfers between unrelated queues to happen in parallel. You may assume the following:

- `queue1` and `queue2` never refer to the same queue.
- `queue1` always has an item to dequeue (so `dequeue(queue1)` succeeds).
- `dequeue`, `enqueue`, `thread_lock`, and `thread_unlock` are all written correctly.
- `thread_lock` is fair. That is, lock requests are granted in the order of the calls.

State whether the implementation (i) works, (ii) doesn't work, or (iii) sometimes works and sometimes doesn't work.

- If you claim it works, present your reasoning about how you came to this conclusion.
- If you claim it doesn't work (or only works sometimes), describe the circumstances that cause it not to work, **and** re-write `transfer` so it always works (and still meets all the requirements).

```
struct queue {
      int lockNum; /* a unique lock number per queue */
      item *headPtr; /* pointer to the head of the queue */
};

void transfer(struct queue *queue1, struct queue *queue2) {
      item *thing; /* the item being transferred */
      thread_lock(queue1->lockNum);
      thread_lock(queue2->lockNum);
      thing = dequeue(queue1);
      enqueue(queue2, thing);
      thread_unlock(queue2->lockNum);
      thread_unlock(queue1->lockNum);
}
```

Name _____

Name _____

2. Writing a Concurrent Program (approx. 20 minutes)

You have joined a software company that is writing an implementation of the Banker's algorithm for the UMass Credit Union. They have defined the global variables below. They have also written the function `isDangerous(int customer, int amount)`, which returns 1 if letting the specified customer borrow the specified amount may allow deadlock (otherwise `isDangerous` returns 0). `isDangerous` does not modify any global variables.

```
int cash; /* amount of cash currently at the Credit Union */

int creditLimit[CUSTOMERS]; /* each customer's credit limit (the maximum
                                    each customer can borrow) */

int currentBorrowed[CUSTOMERS]; /* the current amount borrowed by each
                                    customer */
```

Recall the general structure of a thread that uses the Banker's algorithm:

```
declareCreditLimit /* sets creditLimit[customer] */
while (not done) {
      getCash(customer, amount)/* borrow money against credit limit */
      do work
}

returnAllCash(customer)
```

Your job is to implement the `getCash` and `returnAllCash` functions. Assume a customer never calls `getCash` with an amount that would cause him/her to exceed his credit limit. Use monitors (`thread_lock`, `thread_unlock`, `threadWait`, `threadSignal`, and `threadBroadcast`) to handle synchronization. Keep your solution as simple as possible.

Name _____

```
getCash(int customer, int amount)
{









}
returnAllCash(int customer)
{









}
```

Name _____

3. Modified Monitors (approx. 20 minutes)

Standard condition variables have no memory of past signals. That is, `threadSignal` has no effect on threads that call `threadWait` in the future.

Your task is to implement a modified condition variable that remembers past signals. A signal should be delivered to a waiting thread if there are any threads waiting on this condition variable; otherwise, a signal should be saved and delivered to a thread that later waits on this condition variable.
Each signal should be delivered to exactly one thread. Assume there is only 1 lock and 1 condition variable.

Write pseudo-code for `thread_wait_new` and `thread_signal_new`. Here are library functions you may use:

- Use the "`test&set`" instruction (**not** `interrupt_disable`) to ensure atomicity in your thread library code. You do not need to worry about interrupts occurring.
- Use `enqueue` and `dequeue` to manipulate thread queues. `enqueue` adds a thread onto the tail of a queue. `dequeue` returns the thread at the head of a queue and deletes that thread from the queue. State clearly which queue and thread is being manipulated.
- Use `swapcontext` to switch to a new thread. State clearly which thread you are switching to.
- Use `thread_lock_internal` and `thread_unlock_internal` to acquire and release the lock. These are internal versions of `thread_lock` and `thread_unlock` that assume atomicity is ensured at the time they are called.

State any assumptions you make about how other functions (particularly those that call `swapcontext`) use `test&set`.

Name _____

Name _____

4. Test case design (approx. 20 minutes)

Consider the following pseudo-code of thread_lock() and thread_unlock(), which allows a lock to be acquired in non-FIFO order (i.e. not in the order of calls to thread_lock()).

```
thread_lock(int lockid)
{
      while (lock[lockid] == BUSY) {
            add self to this lock's queue;
            switchToNextReadyThread(); // current thread goes to sleep
      }
      lock[lockid] = BUSY;
}

thread_unlock(int lockid)
{
      lock[lockid] = FREE;
      if (lock's queue is not empty) {
            remove first thread off this lock's queue,
            and put that thread on the ready queue;
      }
}
```

Your task is to design a test case that will illustrate that the locks are not always acquired in FIFO order by the above design. The next page has a partially complete test case using three threads T1, T2, and T3. Assume that initially T1, T2, and T3 are added to the ready queue so that T1 will run first, followed by T2, and then T3. The thread library is non-preemptive.

Your task is to modify the code for one or more threads below by inserting **at most three** thread_yield() operations so that threads request locks in the order T1, T2, T3 but end up acquiring locks in the non-FIFO order of T1, T3, T2. No other code besides thread_yield() should be inserted.

Name _____

```
// Function called by thread T1
void T1_testLocks()
{


      thread_lock(1);


      ... print out the lock owner ...;


      thread_unlock(1);

}


// Function called by thread T2
void T2_testLocks()
{


      thread_lock(1);


      ... print out the lock owner ...;


      thread_unlock(1);
}


// Function called by thread T3
void T3_testLocks()
{


      thread_lock(1);


      ... print out the lock owner ...;


      thread_unlock(1);
}
```