EECS 482 Midterm (Fall 1998)

You will have 80 minutes to work on this exam, which is closed book. There are 4 problems on 9 pages.

Read the entire exam through before you begin working. Work on those problems you find easiest first. Read each question carefully, and note all that is required of you. Keep your answers clear and concise, and carefully state all of your assumptions.

You are to abide by the University of Michigan/Engineering honor code. Please sign below to indicate that you have abided by the honor code on this exam.

Honor code pledge: I have neither given nor received aid on this exam.

Signature:_____

Name:_____

Uniqname: _____

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

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`, `threadLock`, and `threadUnlock` are all written correctly.
- `threadLock` 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 */
    threadLock(queue1->lockNum);
    threadLock(queue2->lockNum);
    thing = dequeue(queue1);
    enqueue(queue2, thing);
    threadUnlock(queue2->lockNum);
    threadUnlock(queue1->lockNum);
}
```

Uniqname:_____

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 Michigan 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 (`threadLock`, `threadUnlock`, `threadWait`, `threadSignal`, and `threadBroadcast`) to handle synchronization. Keep your solution as simple as possible.

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









}

returnAllCash(int customer)
{









}
```

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 `threadWaitNew` and `threadSignalNew`. Here are library functions you may use:

- Use the "`test&set`" instruction (**not** `interruptDisable`) 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 `threadLockInternal` and `threadUnlockInternal` to acquire and release the lock. These are internal versions of `threadLock` and `threadUnlock` 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`.

Uniqname:_____

4. Lazy Fork (approx. 20 minutes)

As discussed in class, Unix `fork` initializes the contents of the child's address space by aggressively copying the contents of the parent's address space. Your job is to speed up `fork` by deferring or avoiding the work of copying data between the two address spaces. Your solution should reduce the number of bytes copied as much as possible for the two common ways of using `fork`: running two copies of the same program, and starting a new program by using `exec` after `fork`. Other than increased speed, user processes should not notice any difference between the old and new `fork`. Assume that address spaces are represented by a simple page table, and that the MMU respects read/write protection bits and sets reference and dirty bits appropriately. Hint: think about sharing data between the parent and child.

Describe how to modify fork to defer or avoid copying. Be specific and concise in your answers. Note that there are 4 parts to answer (two are on the following page).

a. What is the unit of copying, and why should/can the copy not be done on a larger or smaller unit?

b. What events in the parent will trigger a copy? What events in the child will trigger a copy?

c. Describe how the page tables and MMU data are manipulated and used when `fork` is called.

d. Describe how the page tables and MMU data are manipulated and used when a copy must be made.