Final Examination
Concurrent Programming

18th December 2002

You may use up to one page of notes for this exam, but otherwise it is closed book. Each subproblem is worth the number of points indicated. Graduate students answer all the questions (130 points). Undergraduates may skip up to thirty points worth of problems or subproblems – or answer all of them and I will count the best scores.

You must explain each answer or show how you arrived at it. This is required for full credit and is helpful for getting partial credit. Do your work on these sheets, using additional sheets if necessary.

The exam will end promptly at 12:10.

Good Luck!

1. Consider the following code fragment, which sums the elements of array a[0:n-1}:

```
co
   *
   i=0;
   sum1 = 0;
   **
   while (i<n) {
   ***
      sum1 = sum1+a[i];
      i = i+2;
   }
   ****
//
   j=1;
   sum2 = 0;
   while (j<n) {
      sum2 = sum2 + a[j];
      j = j + 2;
   }
oc
sum = sum1 + sum2;
```

(10)Write assertions corresponding to points *, **, ***, and **** that could be used in programming logic as part of a proof that when the program completes, sum indeed contains the sum of the array elements.

(10) Show that the assignment j=j+2 in the second loop does not interfere with the assertion corresponding to ****

2(a). (10) Give an implementation of a binary semaphore using conditional atomic actions. If the semaphore value is already 1 when a V operation is performed on the semaphore, the operation returns without blocking and without changing the value of the semaphore.

2(b). (10) Give an implementation of a binary semaphore using monitors. Assume signal-and-continue semantics for the monitors.

3. (10) Consider the following program:

```
typedef struct node {
   int val;
   struct node *next;
   struct node *prev;
} Node;
Node a = {3, (struct node *) 0, (struct node *) 0};
Node b = {4, &a, (struct node *) 0};
Node c = {5, (struct node *) 0, (struct node *) 0};
Node d = {6, (struct node *) 0, (struct node *) 0};
a.prev = &b;
co
   c.next = b.next;
   c.prev = &b;
   b.next->prev = &c;
   b.next = &c;
//
   d.next = b.next;
   d.prev = &b;
   b.next->prev = &d;
   b.next = &d;
oc
```

If the arms of the co statement are executed sequentially, the result is a doubly linked list with the nodes in the order b, d, c, a. Give an example of an execution order for the statements of the arms that leads to node c becoming unreachable from b.

4. (10) Consider a uniprocessor system with a strict priority scheduler (the system is always running the highest-priority ready thread) that uses semaphores for synchronization. Describe how an application program (multithreaded) could encounter an unbounded priority inversion.

(10) Describe two different changes that could be made to the system that would prevent unbounded priority inversions. (I am looking for changes to the (strict priority scheduler+semaphores) system, not changes to the application.)

5. (10) Using the communication primitives developed in Project 2, give an example of a program that deadlocks if the primitives are synchronous but which does not deadlock if the primitives are asynchronous.

6. (20) Recall the readers and writers problem: multiple readers may concurrently read a database, but a writer may only execute if there are no readers and no other writers. Using monitors, create entry and exit code for writers and readers. That is, provide implementations for the procedures StartWriter, StartReader, EndWriter, EndReader in the code sequences below. Your implementation should give a waiting writer priority over subsequently arriving readers.

```
Writer:
   StartWriter();
   ... write ...
   EndWriter();
```

```
Reader:
   StartReader();
   ... read ...
   EndReader();
```

7. (10) We talked about the double-checked locking pattern in the context of optionally initializing a variable, but it can be applied any time that some property needs to be established once and thereafter it remains true. Suppose that P is such a property and MakeP is a procedure that can be run exactly once to make P true. In a sequential program we could write

```
if (!P) {
   MakeP()
}
```

at any point where we needed P to be true. In a concurrent program the above code runs the risk of calling MakeP() more than once. Show how to use double-checked locking to ensure that MakeP() is called exactly once in a concurrent program, while not requiring locking every time the truth of P is tested.

8. In client-server computing it is common for a server to be working on requests from many clients at the same time and there are many ways to accomplish such concurrency. Consider the following approaches; event-loop (EL), heavyweight process (HP) per client request, lightweight thread (LT) per client request, leaders and followers using threads (LF). For each of the following criteria, order the approaches using > to mean "is better than" and = to mean "is approximately the same as" in terms of their desirability for that criteria. Justify your answer.

Example: startup overhead

Example answer: EL > LF > LT > HP

Example reason: the event loop (EL) requires the minimum allocation of a data structure to represent the client request; LF requires waiting for an existing thread to not be busy; LT requires allocation and initialization of a new thread in an existing address space; HP requires allocation and initialization of a new process including allocating a new address space.

a. (5) CPU efficiency (efficiency is (useful work accomplished)/(total work expended):

b: (5) Total memory required

c: (5) Ease of programming to exploit a multi-processor

d: (5) What overall measure of the different approaches would you suggest to guide the choice of which to use in a particular situation? (There is no one right answer here, but you should be able to synthesize a reasonable answer based on what you've learned in this class.)