CSE 120 Spring 2002 Midterm Examination

You have 80 minutes to complete this exam. Please answer all four questions. The value of each question is indicated at the beginning of the question; take these scores into account if you find yourself pressed for time.

1. (20 points) Consider the following purported solution to the bounded buffer problem. The function PRODUCE produces an integer and the function CONSUME consumes an integer.

```
int buf[4], in = 0, out = 0;
cobegin
while (1) {
  while (out == (in + 1) mod 4) ;
  buf[in] = PRODUCE();
  in = (in + 1) mod 4;
  }
||
while (1) {
  while (in == out);
  CONSUME(buf[out]);
  out = (out + 1) mod 4;
  }
coend;
```

- Is this a *safe* solution to the bounded buffer problem? If so, then give an informal argument as to why it is; otherwise, give an adversary schedule demonstrating that it is not. If you argue that it is safe, be sure to state how many values can be produced but not yet consumed.

It is safe. Values are placed and removed from the buffer in the same sequence. The guard of the loop in the consumer does not allow `out` to lap `in`, and the guard of the loop in the producer does not allow `in` to overwrite a value that has not yet been consumed. Indeed, this solution only allows 3 values to be produced and not consumed.

- Is this a *live* solution to the bounded buffer problem? If so, then give an informal argument as to why it is; otherwise, give an adversary schedule demonstrating that it is not.

It is live assuming that `PRODUCE` and `CONSUME` always complete. The producer and consumer increment variables in the guards thereby making a true guard false. Hence, liveness could be lost only if both guards were simultaneously true. There are no values of `in` and `out` that can make the guards simultaneously true.

2. (20 points) The following is the semaphore implementation of Hoare-style monitors using semaphores that we covered in class (I've included it here for your convenience).

- Declare the following structure:

```
struct {
  semaphore lock = 1, csem = 0, usem = 0;
  int ccount = 0, ucount = 0;
  } m;
```

- Wrap each invocation of an entry procedure as follows:

```
P(m.lock);
invoke entry procedure
if (m.ucount > 0) V(m.usem);
else V(m.lock);
```

- Replace `c.signal()` with the following:

```
m.ucount++;
if (m.ccount > 0) { V(m.csem); P(m.usem); }
m.ucount--;
```

- Replace `c.wait()` with the following:

```
m.ccount++;
if (m.ucount > 0) V(m.usem);
else V(m.lock);
P(m.csem);
m.ccount--;
```

a) Explain how it could be possible for more than one process to be on the urgent queue at the same time.

Consider an entry procedure that has the code:

```
c.wait();
c.signal();
```

Suppose two processes *a* and *b* are blocked on `c`. Let a third process *c* execute `c.signal()`. This will (eventually) result with *c* and *b* on the urgent queue and *a* in the monitor.

b) Suppose one instead implemented `signal` and `wait` as follows:

- Replace `c.signal()` with the following:

```
m.ucount++;
if (m.ccount > 0) { V(m.csem); m.ccount--; P(m.usem); }
m.ucount--;
```

- Replace `c.wait()` with the following:

```
m.ccount++;
if (m.ucount > 0) V(m.usem);
else V(m.lock);
P(m.csem);
```

Show that this implementation suffers a liveness problem: it is possible for a process to be waiting on the condition variable `c`, yet no subsequent `c.signal()` will unblock the process.

This question was incorrect. The problem that results is also a *safety* issue as well as a *liveness* issue. We struck this part off of the test and gave extra credit to whoever worked on this problem in a fruitful direction.

The problem arises because the instruction I've highlighted in red is not executed with mutual exclusion. Notice that the process that executes `c.signal()` unblocks a process from the `c` semaphore and then is ready to execute the highlighted code. That unblocked process will eventually, without blocking, start executing monitored code; it has the monitor lock. If that process also calls `c.signal()` then it too can unblock a process from the c semaphore and be at the code highlighted in red.

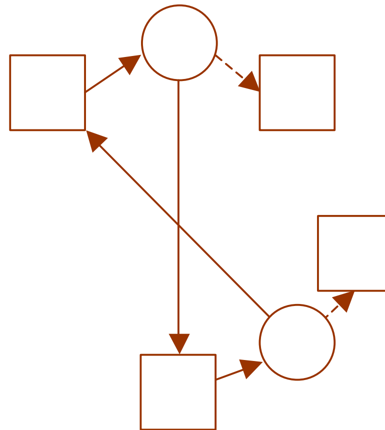
If two instances of the highlighted code is executed concurrently, then the value of `m.ccount` may only be decremented by one. Thus, `m.ccount` may be 1 when there are no processes blocked on condition `c`.

When this happens, the signalling process will block on `m.usem`. If no process enters or exits an entry procedure, this process is stuck here. If a process does, and signals `c`, then `m.csem` will be set to 1. A subsequent process that waits on `c` will not block..

3. (20 points) Consider the following variant of the Dining Philosopher's problem in which each philosopher has three arms (I'll let you decide what to call this problem). Assume that the number n of philosophers is odd and at least 3. Philosopher i needs forks i, $i \oplus 1$ and $i \oplus \lceil n/2 \rceil$ to eat (where $\oplus$ indicates addition modulo n). For example, if n is 5, then $\lceil n/2 \rceil$ is 3, and so Philosopher 2 needs forks, 2, 3 and 0 to eat.

a) Assuming that philosophers allocate a fork at a time, for any value of n that is odd and at least 3, what is the smallest number of philosophers that can deadlock? Illustrate such a smallest deadlock with a resource allocation graph.

Two can deadlock for any odd value of $n > 3$:



b) Using semaphores and hierarchical locking, give a deadlock-free solution to this problem. You can give the code for a single (generic) philosopher rather than writing out the complete **cobegin/coend** statement.

Have philosopher i execute the following loop:

```
while (1) {
  f1 = min(i, i + 1 % n, (i + 1)/2 % n);
  f2 = max(i, i + 1 % n, (i + 1)/2 % n);
  f3 = mid(i, i + 1 % n, (i + 1)/2 % n);
  P(f1);
  P(f2);
  P(f3);
  eat;
  V(f1);
  V(f2);
  V(f3);
  }
```

4. (20 points) Some short questions, each worth five points.

   a) Recall that a simple solution to the mutual exclusion problem uses a single general semaphore `mutex` that is initialized to 1. Then, when a process wishes to enter the critical section, it executes `P(mutex)`, and when the process leaves the critical section it executes `V(mutex)`.

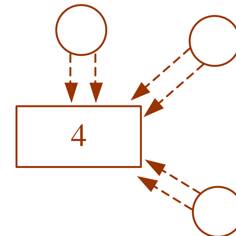
      Consider $n > 2$ processes that are using this simple solution to the mutual exclusion problem. What queuing properties does the semaphore need to implement to ensure that this simple solution satisfies the liveness property of the mutual exclusion problem?

      If an element is added to a queue and an unbounded number of removals occur, then the element will be one of those that is removed. A FIFO queue implements this property.

   b) Consider a fast-food restaurant as a concurrent system. Give five properties of the system: two safety, two liveness, and one of your own choice. Be sure to identify each property you give as a safety or liveness property.

      (S) There is always at least one person who is taking orders.
      (S) The restaurant is open every day from 6AM to 10PM.
      (S) Each small hamburger weighs between 5.5 and 6 oz.
      (L) After you pay for your order, your food will eventually come.
      (L) Each person who enters the restaurant eventually leaves.

   c) Suppose three processes share four resource units which can be reserved and released only one at a time. Each process needs a maximum of two units. Show that deadlock cannot occur.

      The maximum claims graph at the right illustrates the initial state of the system. Since all requests are for single units, for this system to be deadlocked, all four units need to have been allocated and all processes holding resources have only one unit (since otherwise they will request no further resources before releasing). If all four units are allocated, though, then at least one of the three processes has allocated two units (this is an instance of the commonly used pigeonhole principle that says if you distribute more than n objects among n individuals, then at least one individual has at least two objects). That process is not blocked, and so the system cannot deadlock.

   d) True/False:

T	**F**	A cycle in a resource allocation graph is sufficient for the system to be deadlocked.
T	**F**	A condition variable is a kind of semaphore.
T	**F**	Busy waiting is never a good idea.
T	**F**	All schedulers implement the finite progress axiom.
T	F	A single process can be deadlocked by itself.