# CPS 110 Midterm Exam

March 9, 2000

There are four questions worth 50 points each. Answer all questions. Please sign your name and staple your answers together. Allocate your time carefully. Your answers will be graded on content, not style. Any kind of pseudocode is fine as long as its meaning is clear. You have 75 minutes.

1. Tweedledum and Tweedledee are separate threads executing their respective procedures. The code below is intended to cause them to forever take turns exchanging insults through the shared variable X in strict alternation. Tweedledum goes first. The *Sleep()* and *Wakeup()* routines operate as discussed in class: *Sleep* blocks the calling thread, and *Wakeup* unblocks a specific thread if that thread is blocked, otherwise its behavior is unpredictable (like Nachos *Scheduler::ReadyToRun*).

```
void
Tweedledum()
{
    while(1) {
        Sleep();
        x = Quarrel(x);
        Wakeup(Tweedledee thread);
    }
}
```

```
void
Tweedledee()
{
    while(1) {
        x = Quarrel(x);
        Wakeup(Tweedledum thread);
        Sleep();
    }
}
```

   a) The code shown above exhibits a well-known synchronization flaw. Explain the flaw. Briefly outline a scenario in which this code would fail (after the first iteration), and the outcome of that scenario.

   b) Show how to fix the problem using semaphore *P* (down) and *V* (up) operations. No, you may not disable interrupts or use *Yield*.

   c) Implement Tweedledum and Tweedledee correctly using a mutex and condition variable.

2. *Unisex restroom*. The attorneys and staff at Ally McBeal's firm have determined that concurrent sharing of the restroom by males and females is creating unnecessary stress in the workplace. Since there is only one restroom, they propose to coordinate by the following policy. The restroom may be visited concurrently by any number of individuals provided they are of the same gender. A sign on the door indicates the status of the restroom (*open for males*, *open for females*, or *empty*). Each individual politely waits until the restroom is available to serve that individual's gender. Implement procedures that allow threads to *Enter* and *Exit* the restroom in accordance with this policy, using mutexes and condition variables for synchronization. Be sure your solution avoids starvation and deadlock.

3. Implement general (counting) semaphores using binary semaphores.

4. *Process manager.* You are to implement a *Process* class to serve as the core of a process manager for a multiprogrammed kernel. The *Process* class maintains parent/child/sibling relationships among processes, and coordinates the *Exec*, *Exit*, and *Join* system calls. Your solution will implement three key *Process* methods to operate on *Process* object.

   *p->Birth(Process* parent)* registers this newly created process *p* as a child of its parent.
   *p->Death(int status)* indicates that this process *p* has exited with the specified status.
   *int status = child->Join()* waits for this *child* process to exit (i.e., to call *Death*), and returns the child's exit status.

   Each *Process* object *P* has an associated thread. The thread bound to *P* calls *Birth* and *Death* on *P* as part of the implementation of the *Exec* and *Exit* system calls respectively. *Join* on *P* may be called only by the thread bound to the parent of *P*. You do not need to create these threads or enforce these restrictions; they are intended to simplify the problem.

   *Death* on *P* waits until (1) the children of *P* have exited, and (2) the parent of *P* has exited or has executed a *Join* on *P*. The intent is that the *Process* object for *P* may be safely deleted after *Death* returns.

   Your solution should represent the process tree with the following state for each *Process* object *P*: (1) a list of the children of *P*, (2) a pointer to the parent of *P*, and (3) *P*'s exit status if *P* has exited. Use a single mutex and condition variable for synchronization. You may assume any reasonable set of list primitives if their meaning is clear, and you may add other methods for *Process* as needed to support *Birth*, *Death*, and *Join*. Be sure that your solution is free of deadlock and dangling references.