CPS 110 Midterm

Out: Wednesday, October 8, 14:50
Due via Hardcopy: 16:05

This is an in-class exam. Read the entire exam through before you begin working. Read each question carefully, and note all that is required of you. Keep your answers *clear and concise*. Often when you write more than you need you end up saying things that are incorrect. I don't expect you to need more than a page for any sub-question.

If you need to make any assumptions that are not clear from the question, then please state them explicitly. For code, any kind of pseudocode is fine as long as its meaning is clear. You may assume standard routines like lists, queues, hash tables, etc.

You are allowed one sheet of notes; any non-printed sources such as other students, laptops, PDAs, and phones are prohibited. Do not discuss the exam with anyone until the exam period is over. Please sign this page to indicate that you have completed your exam within these rules. Good luck!

Name:

Signature:

Problem	Score
1	
2	
3	
4	
5	
total	

1 Scheduling (40 pts)

Consider the following jobs, with arrivals times and times to completion:

- A: arrives at time 0, takes 6 time units to run
- B: arrives at time 0, takes 2 time units to run
- C: arrives at time 3, takes 2 time units to run
- D: arrives at time 8, takes 5 time units to run

Trace through the resulting schedules under the FCFS and STCF (also called SJF) scheduling policies, and their preemptive variants: FCFS-P (also called RR) and STCF-P. Compute the average response times.

Note: the run times refer to time that the job requires on the CPU. The jobs use CPU time only with no I/O. The system has one CPU (core).

2 Semaphores (40 points)

Your Project 1 thread library provides a monitor abstraction (mutex locks and condition variables) for user threads to synchronize. Semaphores offer an alternative abstraction. Outline how to extend your thread library to implement semaphores. In particular, show how to implement the semaphore `Up/V()` and `Down/P()` operations. You may rely on your thread library's internal data structures and routines for thread contexts, thread queues, context switch, and internal synchronization (disable/enable interrupts).

3 Mailboxes (40 pts)

Show to extend your Project 1 thread library with a mailbox abstraction to enable threads to send and receive messages. Here's how it works. Each mailbox is identified by a number, like locks in Project 1. Each mailbox has a message queue. There are two operations on a mailbox: `Send()` and `Receive()`. `Send()` takes a pointer to a message object and posts it on the message queue. `Receive()` takes the next message off the queue and returns a pointer to the message object. Each mailbox can hold at most *N* messages. `Send()` blocks until there is room on the queue to post a message. `Receive()` blocks until there is a message on the queue to deliver. Messages are delivered in order, and no message is delivered more than once. Synchronize with monitors (mutex/condition variable) or semaphores if you prefer.

```
Send(Message_t *m)
{

}

Message_t *Receive()
{
    (on back)
}
```

4 Deadlocked (40 pts)

Is it possible for a program to use these mailboxes (from Problem 3) in a way that results in deadlock? If no, explain why deadlock cannot occur: which of the preconditions for deadlock is/are not met, and why? If yes, construct an example, and briefly discuss how the system should deal with deadlock. Note: an "example" is defined as a program for which some schedules lead to deadlock and others do not, together with a legal schedule that leads to deadlock.

5 Coroutines (40 pts)

Various elders trace the origin of threads to a 1963 paper by Melvin Conway introducing a programming construct called *coroutines*. Like a subroutine/procedure/method, a coroutine is a block of code with a named entry point and a sequence of statements in some programming language, which may include calls to subroutines. What's new is that coroutines introduce a `Yield()` primitive: `Yield(target)` passes control to another coroutine by name (given by the string argument `target`. A coroutine begins execution at its entry point when it first receives control. Each subsequent yield to a coroutine resumes its execution at the point of its last yield.

Wikipedia provides the following example:

```
var q := new queue

coroutine produce
   loop
      while q is not full
         create some new items
         add the items to q
      yield to consume

coroutine consume
   loop
      while q is not empty
         remove some items from q
         use the items
      yield to produce
```

Compare/contrast the implementation of coroutines to threads with respect to stacks, scheduling, and synchronization.