Faculty of Arts & Sciences
Department of Computer Science
CMPS 272—Operating Systems
Fall 2005–2005
Friday, December 2, 2005

Midterm Exam
Version A

Name:________________________________ **Student Id:**__________________________

Signature:________________________________ **Section:**

Lect I	**10–11**
Lect II	**1–2**

Duration: 100 minutes

Instructions

- The exam is made of two handouts: *Questions* and *Appendices*. Make sure you have both of them.
- The exam is closed book, closed notes, and closed neighbor.
- Your handwriting should be readable so it can be graded. Include all work or justification for partial credit.

Problem 1	**15**		
Problem 2	**10**		
Problem 3	**10**		
3.1		5	
3.2		5	
Problem 4	**15**		
Problem 5	**15**		
5.1		5	
5.2		10	
Problem 6	**15**		
6.1		6	
6.2		9	
Problem 7	**25**		
7.1		10	
7.2		15	
Total	**105**		

Problem 1. True/False Questions (15 = 15 × 1)

Mark your answer in the table below. Note that there is a **–1 penalty** associated with each wrong answer!

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T															
F															

1. A thread is a lightweight process.
2. The priority inversion problem is caused by a high priority process running inside its critical section.
3. A system call is an interface between the OS and the user.
4. Interrupt handlers can be modified by user programs.
5. A shell is a command interpreter but not a process.
6. An unsafe state is a deadlocked state.
7. Interrupt handlers execute in user space.
8. Atomic operations mean concurrent operations.
9. All signals can be disabled.
10. Signals cannot be used in interprocess communication.
11. "`fork()`" is a Unix system call that creates two new child processes.
12. Process scheduling has to do with moving processes back and forth between the running and blocked states.
13. Swapping has to do with process scheduling.
14. The critical section of a process is a private segment of code of that process.
15. One of the solutions for the race condition problem is to have one process run faster than the other.

Problem 2. Deadlocks (10)

Consider a resource allocation problem that consists of 5 processes (P_0, P_1, P_2, P_3, & P_4) and 4 resource types (R_0, R_1, R_2, & R_3). There are 6 units of resource R_0, 3 units of R_1, 4 units of R_2, and 2 units of R_3. This system uses the Banker's algorithm for deadlock avoidance. The table below gives the currently allocated resources and the maximum number of resources each process can have.

	Allocation				**Maximum**			
Process	R_0	R_1	R_2	R_3	R_0	R_1	R_2	R_3
P_0	3	0	1	1	4	1	1	1
P_1	0	1	0	0	0	2	1	2
P_2	1	1	1	0	4	2	1	0
P_3	1	1	0	1	1	1	1	1
P_4	0	0	0	0	2	1	1	0

The system receives the following two requests in order, P_1 (0 0 1 0) and P_2 (0 0 1 0). How will the system handle these requests? Justify your answer.

Problem 3. Scheduling (10 = 5 + 5)

Consider the scheduling problem as described in the table below.

Process	**Arrival**	**Duration**	**Priority**
P_0	0	5	2
P_1	1	4	4
P_2	1	1	3
P_3	2	3	1
P_4	2	6	5

1. Using the *priority with prevention* scheduling algorithm (1 = *highest*, 5 = *lowest*), find the average waiting time for all processes (show your work).

2. Using SJF with preemption, find the average waiting time for all processes (show your work).

Problem 4. Processes (15)

What does the following piece of code do? Draw a graph showing the parent-child relations between processes. Assign process ids start at 100.

```
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>

#define N 4

int i;
pid_t pid;

for (i = 1; i < N; i++) {
  if (pid = fork())
    break;
  }
fprintf(stderr, "This is process %ld with parent %ld\n",
        (long) getpid(), (long) getppid());
```

Problem 5. Synchronization I (15 = 5 + 10)

Consider a system with three asynchronous concurrent processes. Each of these processes generates an event as shown below. Define semaphores and use P & V operations so the events always occur in the order *event*-1, *event*-2, *event*-3, *event*-1, *event*-2, *event*-3…

1. Declare and initialize the needed semaphores:

2. Fill in the blanks in the table below.

***P*$_1$**	***P*$_2$**	***P*$_3$**
…	…	…
repeat	**repeat**	**repeat**
…	…	…
event-1	*event*-2	*event*-3
…	…	…
until *false*	**until** *false*	**until** *false*

Problem 6. Synchronization II (15 = 6 + 9)

The following program is an attempt to solve the mutual exclusion program for two processes.

```
#define FALSE 0
#define TRUE  1
#define N     2

void enter_region (int process) {
  int other;

  other = 1 - process;
  interested[process] = TRUE;
  turn = process;
  while (interested[other] == TRUE || turn == process) {
    /* do nothing */
  }
}

void leave_region (int process) {
  interested[process] = TRUE;
}
```

1. Give a sequence of events that will cause a race condition.

2. Correct the program so as to avoid the race condition.

Problem 7. Synchronization III (25 = 10 + 15)

With the rainy season upon us, local roads flood on a regular basis. During or immediately after a storm, a four-lane road may be reduced to a single road. In a stroke of genius and/or wisdom, the local transportation authority hires you to program the microcontrollers that control access to single lane roads.

1. Write a semaphore-based protocol that allows drivers to avoid gridlock. Do not worry about starvation in either direction of the one-lane road.

2. Modify your protocol to maximize throughput while ensuring fairness. In this solution, a car traveling in a particular direction can enter the one-lane road provided (1) there are other cars traveling in the same direction in the one lane road and (2) there are no cars waiting at the other end of the one-lane road to enter it in the opposite direction.