# CIS 415: Operating Systems

Prof. Allen D. Malony

Midterm – November 3, 2015

**NAME:** ______________________________

| **Section** | **Total Points** | **Points Scored** |
|---|---|---|
| 1. Processes and Threads | 5 + 5 + 5 + 5 + 5 + 5 | / 30 |
| 2. Scheduling | 15 + 15 | / 30 |
| 3. Concurrency and Synchronization | 5 + 5 + 5 + 5 + 20 | / 40 |
| 4. Deadlocks | 5 + 5 + 15 + 15 | / 40 |
| Total | 60 + 80 = 140 | |

Comments:

1. Take a deep breath.
2. Write your name on the front page now and initial all other pages.
3. There are 4 sections. Do all sections. Each section is designed to take 20 minutes, more or less.
4. Concept questions in total are worth 60 points. Concept questions are intended to be short answer. If you are spending more than 5 minutes on ANY concept questions, you are writing too much! Move on!
5. If you have a question, raise your hand and we will come to you, if we can. Otherwise, we will acknowledge you and you can come to us.

Exams should be challenging learning experiences. Enjoy it!!!

## 1 Processes and Threads (40)

There are only concept questions in this section.

### 1.1 Concepts (40)

**What do the terms *multiprogramming*, *multiprocessing*, and *multithreading* mean? Is it possible to do multiprogramming if the computer system only has 1 processor?**

**What is the purpose of system calls and how do they work? Give two examples of common system calls.**

**What is a process control block and what is its purpose?**

**What is the process address space? What is in it and where are things located? Draw a picture to show this.**

**What is the difference between the 1:1 and M:1 threading models? Why might we prefer a 1:1 threading model?**

**What is a process? What is thread? How do they differ? What parts of the process address space are shared when using threads versus forking processes?**

## 2 Scheduling (30)

There are no concept questions in this section.

### 2.1 Just Take it Easy, We Will Get to You Eventually (15)

Per Brinch Hansen was a famous computer scientist working in the field of concurrent programming and operating systems. He developed an interesting scheduling strategy called *Highest-Response-ratio-Next (HRN)* that corrects some of the weaknesses in the shortest-job-first (SJF) strategy. HRN is a nonpreemptive scheduling discipline in which the priority of each job is a function not only of the job's service (run) time, but also of the amount of time the job has been waiting for service. Priorities in HRN are calculated dynamically according to the formula

$$priority = \frac{timewaiting + servicetime}{servicetime}$$

**What is a possible weakness of the SJF strategy that HRN is trying to correct?**

**Is indefinite postponement a problem with SJF? How does HRN prevent it?**

**How does HRN decrease the favoritism shown by other strategies to short new jobs?**

**Suppose two jobs have been waiting for about the same time. Are their priorities about the same? Explain your answer.**

## 2.2 Scheduler Cage Match (15)

Five batch jobs A through E, arrive at a computer center at almost the same time. They have estimated running times of 10, 6, 2, 4, and 8 minutes. Their (externally determined) priorities are 3, 5, 2, 1, and 4, respectively, with 5 being the highest priority. For each of the following scheduling algorithms, determine the mean process turnaround time. Ignore process switching overhead. For Round Robin, assume that the system is multiprogrammed, and that each job gets its fair share of the CPU. For the others, assume that only one job at a time runs, until if finishes. All jobs are completely CPU bound. Provide your answer in minutes.

*Round Robin*

*Priority Scheduling*

*FCFS (assumed arrival order 10, 6, 2, 4, 8)*

SJF

## 3 Concurrency and Synchronization (40)

### 3.1 Concepts (20)

**What does mutual exclusion mean?**

**What is a critical section? List the requirements for a solution to the critical section problem.**

**Why is atomic execution important for a synchronization contruct to have?**

**The Facebook "CPU spinning" problem that Prof. Malony presented in class is an interesting case of a process wanting to be responsive while avoiding excessive use of the CPU. One idea that might help is to implement a blocking mutex, but with the following twist. Suppose that the *acquire()* routine accepts a boolean variable *block* which will cause blocking to occur if *block == 1* and the mutex is not available. However, if *block == 0* and the mutex is not available, no blocking will occur, and the *acquire()* will return a 0 to indicate that the mutex was not available. Otherwise, *acquire()* will return a 1. How might you use this "maybe" blocking mutex to address the dual objectives of responsiveness and efficiency?**

## 3.2 Starving Philosophers (20)

Once there were five philosophers whose life consisted of thinking and eating. They would sit around around a round table on which was set a large bowl of spaghetti, five plates (one for each philosopher) and five forks (one between each plate). A philosopher sitting at the table wishing to eat must use the two forks together on either side of the plate to take and eat some spaghetti.

Andrew Tanenbaum, a famous computer scientist working in the area of operating systems, presented the following solution to the dining philosophers problem is one of his operating systems books.

```
#define N           5           /* number of philosophers */
#define LEFT        (i-1) mod N  /* number of i's left neighbor */
#define RIGHT       (i+1) mod N  /* number of i's right neighbor */
#define THINKING    0           /* philosopher is thinking */
#define HUNGRY      1           /* philosopher wants forks to eat */
#define EATING      2           /* philosopher is eating */

typedef int semaphore;          /* semaphores are a special type of int */

int state[N];                   /* array to keep philosopher's state */
semaphore mutex = 1;            /* mutual exclusion semaphore */
semaphore s[N];                 /* semaphore per philosopher, initially 0 */

philosopher(i) {
  int i;                        /* philosopher number, 0 to N-1 */
  while (TRUE) {                /* repeat forever */
    think();                    /* philosopher is thinking */
    take_forks(i);              /* acquire two forks or block */
    eat();                      /* yum-yum, spaghetti */
    put_forks(i);               /* put both forks back on table */
  }
}

take_forks(i) {
  int i:                        /* philosopher number, 0 to N-1 */
  wait(mutex);                  /* enter critical region */
  state[i] = HUNGRY;            /* record that philosopher is hungry */
  test(i);                      /* try to acquire 2 forks */
  signal(mutex);                /* exit critical region */
  wait(s[i]);                   /* block if forks were not acquired */
}

put_forks(i) {
  int i:                        /* philosopher number, 0 to N-1 */
  wait(mutex);                  /* enter critical region */
  state[i] = THINKING;          /* philosopher has finished eating */
  test(LEFT);                   /* see if left neighbor can now eat */
  test(RIGHT);                  /* see if right neighbor can now eat */
  signal(mutex);                /* exit critical region */
}

test(i) {
  int i:                        /* philosopher number, 0 to N-1 */
  if ((state[i] == HUNGRY) &&
      (state[LEFT] != EATING) &&
      (state[RIGHT] != EATING)) {
    state[i] = EATING;
    signal(s[i]);
  }
}
```

*a. Describe in words how this solution works.*

*b. Tanenbaum states: "The solution (above) is correct and also allows the maximum parallelism for an arbitrary number of philosophers." Although this solution does prevent deadlocks, it is not entirely correct because starvation is possible. Demonstrate this by counterexample.*
***Hint:*** *Consider the case of five philosophers. Assume that these are gluttonous philosophers – they spend almost no time thinking. As soon as a philosopher has finished one round of eating, he/she is almost immediately hungry. Then consider a configuration in which two of the philosophers are currently eating and the other three are blocked and hungry.*

## 4 Deadlocks (40)

### 4.1 Concepts (10)

**What are the necessary conditions for a deadlock to occur?**

**What is the difference between deadlock prevention and deadlock avoidance?**

## 4.2 Robin Hood Resource Allocation (15)

One day, while reading about the infamous Robin Hood and his Merry Humans, you suddenly got an idea for a new resource allocation policy. After changing into your green tights, you define the rules for the policy. You assume that requests and releases for resources are allowed at any time. If a request for resources cannot be satisfied because the resources are not available, then any processes that are blocked, waiting for resources, are checked. If they have the desired resources, then these resources are taken away from this "rich" process and are given to the "poor" requesting process. The vector of resources for which the waiting process is waiting is increased to include the resources that were taken away.

Ok, good enough. Now you make up an example. Consider a system with three resource types and the vector *Available* initialized to (4,2,2). If process $P_0$ asks for (2,2,1), it gets them. If $P_1$ asks for (1,0,1), it gets them. Then, if $P_0$ asks for (0,0,1), it is blocked (resource not available). If $P_2$ now asks for (2,0,0), it gets the available one (1,0,0) and one that was allocated to $P_0$ (since $P_0$ is blocked). $P_0$'s *Allocation* vector goes down to (1,2,1), and its *Need* vector goes up to (1,0,1).

**Can deadlock occur? If so, given an example. If not, which necessary condition cannot occur?**

**Can indefinite blocking occur?**

## 4.3 “A ship in port is safe, but that’s not what ships are built for.” Grace Hopper (15)

You have been hired by the U.S. Navy to create a system to allocate different types of ships to a group of Navy Admirals who are trying to apply a new “efficiency” protocol of only using what ships are needed at any time in their war games. You created a system and now you are testing it out. Consider the following snapshot of the system. There are no current outstanding unsatisfied requests for ships (resources).

AVAILABLE

| R1 | R2 | R3 | R4 |
|---|---|---|---|
| 2 | 1 | 0 | 0 |

| | ALLOCATION | | | | MAXIMUM | | | | NEED | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| PROCESS | R1 | R2 | R3 | R4 | R1 | R2 | R3 | R4 | R1 | R2 | R3 | R4 |
| P1 | 0 | 0 | 1 | 2 | 0 | 0 | 1 | 2 | | | | |
| P2 | 2 | 0 | 0 | 0 | 2 | 7 | 5 | 0 | | | | |
| P3 | 0 | 0 | 3 | 4 | 6 | 6 | 5 | 6 | | | | |
| P4 | 2 | 3 | 5 | 4 | 4 | 3 | 5 | 6 | | | | |
| P5 | 0 | 3 | 3 | 2 | 0 | 6 | 5 | 2 | | | | |

a. Compute what each Admiral (process) still might request and display in the columns labeled “NEED.”

b. Is this system currently in a safe or unsafe state? Why?

c. Is this system currently deadlocked? Why or why not?

d. Which processes, if any, are or may become deadlocked?

e. If a request from Admiral P3 arrives for (0,1,0,0), can that request be safely granted immediately? In what state (deadlocked, safe, unsafe) would immediately granting that whole request leave the system? Which Admirals (processes), if any, are or may become deadlocked if this whole request is granted immediately?