CSC 4103 - Operating Systems
Fall 2009

LECTURE - XIV
# MIDTERM REVIEW

Tevfik Koşar

Louisiana State University
October 13th, 2009

## Midterm Exam

October 15th, Thursday
3:10pm-4:30pm
@1112 Patrick Taylor Hall

## Chapters included in the Midterm Exam

- Ch. 1.1-1.10 (Introduction)
- Ch. 2.1-2.8 (OS Structures)
- Ch. 3.1-3.4 (Processes)
- Ch. 4.1-4.4 (Threads)
- Ch. 5.1-5.3 (CPU Scheduling)
- Ch. 6.1-6.7 (Synchronization)
- Ch. 7-1-7.7 (Deadlocks)

## 1 & 2: Overview

- Basic OS Components
- OS Design Goals & Responsibilities
- OS Design Approaches
- Kernel Mode vs User Mode
- System Calls
- Virtual Machines



## 3. Processes

- Process Creation & Termination
- Context Switching
- Process Control Block (PCB)
- Process States
- Process Queues & Scheduling
- Interprocess Communication



## 4. Threads

- Concurrent Programming
- Threads vs Processes
- Threading Implementation & Multi-threading Models
- Other Threading Issues
  - Thread creation & cancellation
  - Signal handling
  - Thread pools
  - Thread specific data

## 5. CPU Scheduling

- Scheduling Criteria & Metrics
- Scheduling Algorithms
  - FCFS, SJF, Priority, Round Robing
  - Preemptive vs Non-preemptive
  - Gantt charts & measurement of different metrics
- Multilevel Feedback Queues
- Estimating CPU bursts



## 6. Synchronization

- Race Conditions
- Critical Section Problem
- Mutual Exclusion
- Semaphores
- Monitors
- Classic Problems of Synchronization
  - Bounded Buffer
  - Readers-Writers
  - Dining Philosophers
  - Sleeping Barber



## 7. Deadlocks

- Deadlock Characterization
- Deadlock Detection
  - Resource Allocation Graphs
  - Wait-for Graphs
  - Deadlock detection algorithm
- Deadlock Avoidance
- Deadlock Recovery

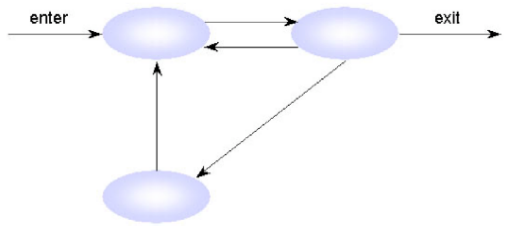


## Exercise Questions



## Question 1

Complete this three-state process transition diagram: label the three states and the four arrows, then give one example of each of the four transitions you have just labeled.






## Question 2

What are the three types of process scheduling performed by the operating system? Briefly describe each type of scheduling

## Question 3

In the code below, assume that *(i)* all `fork` and `execvp` statements execute successfully, *(ii)* the program arguments of `execvp` do not spawn more processes or print out more characters, and *(iii)* all `pid` variables are initialized to `0`.

a. What is the total number of processes that will be created by the execution of this code?
b. How many of each character 'A' to 'G' will be printed out?



## Question 3 *(cont)*

```
void main()
{
      ...
      pid1 = fork();
      pid2 = fork();
      if (pid1 != 0) {
          pid3 = fork();
          printf("A\n");
      } else {
          printf("B\n");
          execvp(...);
      }
      if (pid2 == 0 && pid3 != 0) {
          execvp(...);
          printf("C\n");
      }
      pid4 = fork();
      printf("D\n");
      if (pid3 != 0) {
          printf("E\n");
          pid5 = fork();
          execvp(...);
      }
      printf("F\n");
      execvp(...);
      pid6 = fork();
      printf("G\n");
      if (pid6 == 0)
          pid7 = fork();
}
```



## Question 4

Which processor scheduling algorithm results in the shortest average waiting time. Justify your answer.



## Question 5

Explain why Round-Robin scheduling tends to favor CPU bound processes over I/O bound ones.



## Question 6

CPU scheduling quanta have remained about the same over the past 20 years, but processors are now about 1,000 times faster. Why haven't CPU scheduling quanta changed?



## Question 7

List 4 events that might occur to cause a user process to be context switched off the processor.

## Question 8

Assume S and T are binary semaphores, and X, Y, Z are processes. X and Y are identical processes and consist of the following four statements:

*P(S); P(T); V(T); V(S)*

And, process Z consists of the following statements:

*P(T); P(S); V(S); V(T)*

Would it be safer to run X and Y together or to run X and Z together? Please justify your answer.



## Question 9

Remember that if the semaphore operations *Wait* and *Signal* are not executed atomically, then mutual exclusion may be violated. Assume that *Wait* and *Signal* are implemented as below:

```
void Wait (Semaphore S) {
      while (S.count <= 0) {}
      S.count = S.count - 1;
}
```

```
void Signal (Semaphore S) {
      S.count = S.count + 1;
}
```

Describe a scenario of context switches where two threads, T1 and T2, can both enter a critical section guarded by a single mutex semaphore as a result of a lack of atomicity.



## Question 10

- Given a system that provides binary semaphores (semaphores whose values is either 0 or 1). Show the code to implement counting semaphores using binary semaphores.



## Question 11

Consider the following set of processes:

| Process ID | Arrival Time | Priority | Burst Time |
|---|---|---|---|
| A | 0 | 5 | 20 |
| B | 4 | 1 | 12 |
| C | 12 | 2 | 16 |
| D | 16 | 4 | 8 |
| E | 20 | 3 | 4 |

PS: Lower priority numbers mean higher priority. (eg. 1-highest, 5-lowest)

**(a)** Draw Gantt charts illustrating the execution of these processes using the following algorithms:

**1.** Priority (Non-preemptive)
2. Priority (Preemptive):



## Question 11 *(cont.)*

b) What is the waiting time of each process for each of the scheduling algorithms in Part a?

c) What is the turnaround time of each process for each of the scheduling algorithms in Part a?



## Question 12

Consider the exponential average formula used to predict the length of the next CPU burst. What are the implications of assigning the following values to the parameters used by the algorithm?

a. $\alpha = 0$ and $\tau_0 = 100 milliseconds$

b. $\alpha = 0.99$ and $\tau_0 = 10 milliseconds$

## Question 13

```
boolean lamp[2];
int book = 0;
```

```
void do_thread0()
{
    while (true) {
        lamp[0] = true;
        while (lamp[1]) {
            if (book == 1) {
                lamp[0] = false;
                while (book == 1);
                    /* nothing */
                lamp[0] = true;
            }
        }
        /*** CRITICAL REGION 0 ***/
        book = 0;
        lamp[0] = false;
        ...
    }
}
```

```
void do_thread1()
{
    while (true) {
        lamp[1] = true;
        while (lamp[0]) {
            if (book == 0) {
                lamp[1] = false;
                while (book == 0);
                    /* nothing */
                lamp[1] = true;
            }
        }
        /*** CRITICAL REGION 1 ***/
        book = 1;
        lamp[1] = false;
        ...
    }
}
```

## Question 13 *(cont)*

Does this code guarantee mutual exclusion of the two threads from their respective critical regions?



## Question 13-b

Does this code guarantee "progress", i.e., if one thread is currently executing outside its critical region, the other thread will always have the opportunity to enter its own critical region?