CSE 421/521 - Operating Systems
Fall 2012

Lecture - XII
Midterm Review

Tevfik Koşar

University at Buffalo
October 11th, 2012

Midterm Exam

October 16th, Tuesday
9:30am-10:50am
@101 Davis

Chapters included in the Midterm Exam

- Ch. 1 (Introduction)
- Ch. 2 (OS Structures)
- Ch. 3 (Processes)
- Ch. 4 (Threads)
- Ch. 5 (CPU Scheduling)
- Ch. 6 (Synchronization)
- Ch. 7 (Deadlocks)

1 & 2: Overview

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

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 *(Bankers alg. excluded)*
- Deadlock Recovery

Exercise Questions

Question 1

Are each of the following statements True or False? Circle the correct answer.

(a) In multiprogramming, it is safe to have an arbitrary number of threads/ processes reading a piece of data at once. (True / False)
(b) Kernel mode can directly access hardware devices, user mode cannot. (True / False)
(c) Deadlocks cannot arise without semaphores. (True / False)
(d) Semaphores are destroyed by the OS when your process exits. (True / False)
(e) A process that is blocked is not given any processor time by the scheduler until the condition that caused the blocking no longer applies. (True / False)

Solution 1

Are each of the following statements True or False? Circle the correct answer.

(a) In multiprogramming, it is safe to have an arbitrary number of threads/ processes reading a piece of data at once. (True / False) --True
(b) Kernel mode can directly access hardware devices, user mode cannot. (True / False) -- True
(c) Deadlocks cannot arise without semaphores. (True / False) -- False
(d) Semaphores are destroyed by the OS when your process exits. (True / False) -- False
(e) A process that is blocked is not given any processor time by the scheduler until the condition that caused the blocking no longer applies. (True / False) -- True

Question 2-a

A system that meets the four deadlock conditions will **always/ sometimes/never** result in deadlock?

Solution 2-a

A system that meets the four deadlock conditions will **always/ sometimes/never** result in deadlock?

Sometimes – meeting four deadlock conditions is necessary for a deadlock to occur, but not sufficient.

Question 2-b

Round-robin scheduling **always/sometimes/never** results in more context switches than FCFS?

Solution 2-b

Round-robin scheduling **always/sometimes/never** results in more context switches than FCFS?

Sometimes – if every job has an execution time less than the quantum, then it has the same number as FCFS.

Question 2-c

Which of the following scheduling algorithms can lead to starvation (**FIFO/Shortest Job First/Priority/Round Robin**)?

Solution 2-c

Which of the following scheduling algorithms can lead to starvation (**FIFO/Shortest Job First/Priority/Round Robin**)?

SJF, Priority – in either approach, the jobs with lower priority or the long jobs may never get executed depending on the arrival pattern of the jobs.

Question 3

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

Consider the above set of processes.

a) Draw Gantt chart illustrating the execution of these processes using Shortest Job First (Preemptive) algorithm.

b) What is the waiting time of each process

c) What is the turnaround time of each process

Question 4

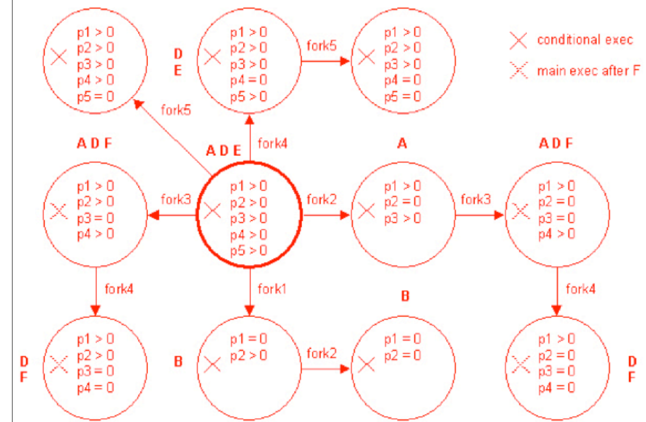
In the code below, assume that *(i)* all `fork` and `execvp` statements execute successfully, *(ii)* the program arguments of `execvp` do not spawn more processes or print out more characters, and *(iii)* all `pid` variables are initialized to `0`.

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 4 *(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();
}
```


Solution 4



Question 5

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.

Solution 5

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.

Answer: It is safer to run X and Y together since they request resources in the same order, which eliminates the circular wait condition needed for deadlock.

Question 6

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.

Solution 6

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.

Answer: Assume that the semaphore is initialized with count = 1. T1 calls Wait, executes the while loop, and breaks out because count is positive. Then a context switch occurs to T2 before T1 can decrement count. T2 also calls Wait, executes the while loop, decrements count, and returns and enters the critical section. Another context switch occurs, T1 decrements count, and also enters the critical section. Mutual exclusion is therefore violated as a result of a lack of atomicity.

Question 7

```
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 7 *(cont)*

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

Solution 7

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

YES, it does. Once thread0 has set lamp[0] to true, thread1 will be busy waiting in the while(lamp[0]) loop and cannot access CR1 (and vice-versa swapping 0 and 1). If thread1 was already in CR1 when thread0 set lamp[0] to true, then necessarily it is thread0 that will be busy waiting in the while(lamp[1]) loop since lamp[1] must be already true by the time thread1 reaches CR1. This is because lamp[1] = true is the always the last statement executed by thread1 before reaching CR1, wherever it came from (vice-versa swapping 0 and 1). Finally, if for any reason both lamp flags are already true upon starting line 1, OR if lines 1 and 2 get interleaved (t0(1)-t1(1)-t0(2)-t1(2)…), then both threads will enter their while(lamp[x]) loops together: at this point, the current book value decides that only one of them will go into the if structure and reset its own lamp flag to 0 (then become trapped in the inner book-controlled loop), thereby allowing the other to escape the while(lamp[x]) loop and enter its CR.

Question 7-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?

Solution 7-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?

NO, it does not. Here is one counter-example:

- **schedule line 1 in thread0 –> lamp[0] becomes true**
- **then execute thread1's lines 1, 2, 3, 4, 5-6-5-6-5-6…**
- **thread1 is trapped into the small loop on lines 5-6 (it entered the big loop because lamp[0] was true and the small loop because book was 0)**
- **now, resume thread0, which is going to execute lines 2, 10, 11 and 12**
- **<u>at this point, thread0 can take all the time it wants to execute inside the noncritical area 13 while thread1 is still trapped in the tight loop of lines 5-6</u>**
- **nothing can free thread1 anymore precisely because the value of book did NOT change in that erroneous code: it remained 0**