CSE 421/521 - Operating Systems
Fall 2011

LECTURE - XIV
MIDTERM REVIEW

Tevfik Koşar

University at Buffalo
October 18th, 2011

Midterm Exam

October 20th, Thursday
9:30am-10:50am
@215 NSC

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

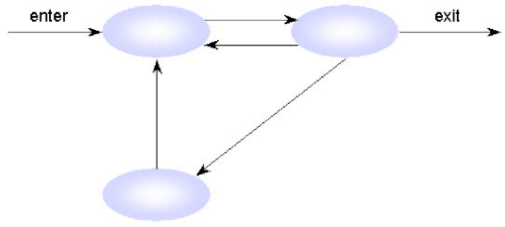
7. Deadlocks

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

Exercise Questions

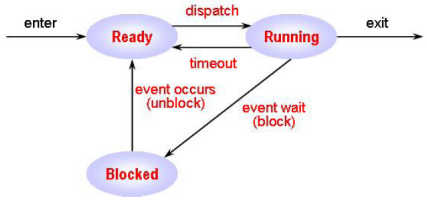
Question 1

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.



Solution 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.



Solution 1 *(cont)*

Timeout, such as preemptive timeout: the process receives a timer interrupt and relinquishes control back to the O/S dispatcher: the O/S puts the process in "Ready" mode and dispatches another process to the CPU

Dispatch: a "Ready" process has been chosen to be the next running process, for example a previously timed-out process or a process that was just created, like a spawned process or a batch job

Event wait (block), such as I/O wait: a process invokes an I/O system call that blocks waiting for the I/O device: the O/S puts the process in "Blocked" mode and dispatches another process to the CPU

Event occurs (unblock), such as end of I/O wait: an I/O system sends an interrupt to the CPU to signal it has finished a task; the O/S may then decide to unblock the waiting ("Blocked") process and put it in the "Ready" queue for upcoming scheduling

Question 2

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

Solution 2

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

Long-term scheduling = process creation: which programs (stored on disk) will be considered for execution (an empty process structure can already be created by the O/S, before actually loading the job).

Medium-term scheduling = loading and swapping in & out: programs jobs are actually loaded into or out of memory (swapped between "Suspended" and "Ready" / "Blocked" states).

Short-term (CPU) scheduling = dispatching: which job available in memory is run next → this is the "dispatch" arrow in b.

Question 3

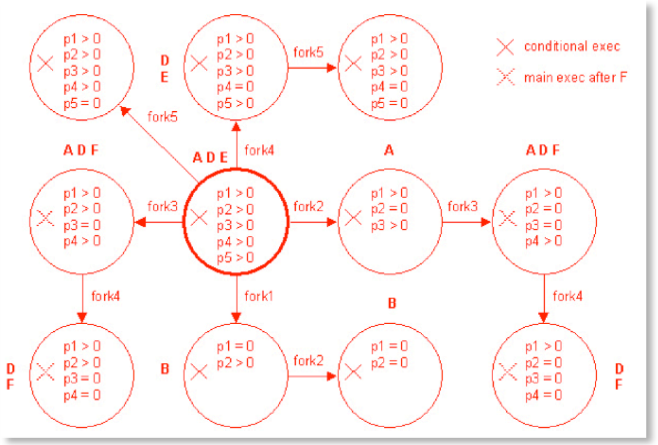
In the code below, assume that *(i)* all `fork` and `execvp` statements execute successfully, *(ii)* the program arguments of `execvp` do not spawn more processes or print out more characters, and *(iii)* all `pid` variables are initialized to `0`.

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();
}
```

Solution 3



Question 4

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

Solution 4

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

Answer: Shortest Job First (SJF), since moving a short process before a long one decreases the waiting time of the short process more than it increases the waiting time of the long process. Consequently, the average waiting time decreases.

Question 5

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

Solution 5

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

Answer: Each process gets put back at the end of the queue no matter how much or how little of the quantum was used. I/O bound processes tend to run for a short period of time and then block which means they might have to wait in the queue a long time.

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?

Solution 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?

Answer: The length of the scheduling quanta is based on the overhead of context switching a processor and the need to move between processes within the time of human perception. The overhead of context switching due to the need to invalidate caches has remained relatively constant, and the time of human perception has also not evolved much in the past 20 years.

Question 7

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

Solution 7

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

Answer:

1. Timer Interrupt (time for another process to run)
2. Blocking to wait for another event (e.g. wait system call)
3. Process termination
4. I/O Interrupt

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.

Solution 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.

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 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.

Solution 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.

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 10

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 $\tau_0 = 10 milliseconds$

Solution 10

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 $\tau_0 = 10 milliseconds$

Answer: When $\alpha = 0$ and $\tau_0 = 100 milliseconds$, the formula always makes a prediction of 100 milliseconds for the next CPU burst. When $\alpha = 0.99$ and $\tau_0 = 10 milliseconds$, the most recent behavior of the process is given much higher weight than the past history associated with the process. Consequently, the scheduling algorithm is almost memory-less, and simply predicts the length of the previous burst for the next quantum of CPU execution.

$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n.$$

Question 11

```
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 11 *(cont)*

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

Solution 11

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 11-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 11-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**
- **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**
- **nothing can free thread1 anymore precisely because the value of book did NOT change in that erroneous code: it remained 0**