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

<table>
<thead>
<tr>
<th>Process ID</th>
<th>Arrival Time</th>
<th>Priority</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>3</td>
<td>20</td>
</tr>
<tr>
<td>B</td>
<td>4</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>C</td>
<td>12</td>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>D</td>
<td>16</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>E</td>
<td>20</td>
<td>3</td>
<td>8</td>
</tr>
</tbody>
</table>

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 all fork and execvp statements execute successfully. (a) The program arguments of execvp do not spawn more processes or print out more characters, and 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?

```c
void main()
{
    pid = fork();
    pid = fork();
    if (pid < 0)
    {
        printf("Error!
    ");
    } else
    {
        printf("Ready!
    ");
    }
    if (pid < 0)
    {
        printf("Ready!
    ");
    }
    pid = fork();
    if (pid < 0)
    {
        printf("Error!
    ");
    }
    printf("Ready!
    ");
    print("Ready!
    ");
    pid = fork();
    if (pid < 0)
    {
        printf("Error!
    ");
    }
}
```

Solution 4

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 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 \texttt{Wait} and \texttt{Signal} are not executed atomically, then mutual exclusion may be violated. Assume that \texttt{Wait} and \texttt{Signal} are implemented as below:

```c
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 \texttt{count} = 1. T1 calls \texttt{Wait}, executes the while loop, and breaks out because \texttt{count} is positive. Then a context switch occurs to T2 before T1 can decrement \texttt{count}. T2 also calls \texttt{Wait}, executes the while loop, decrements \texttt{count}, and returns and enters the critical section. Another context switch occurs, T1 decrements \texttt{count}, and also enters the critical section. Mutual exclusion is therefore violated as a result of a lack of atomicity.

Solution 6

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

```c
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 7

```c
bool lmp[2];
int book = 0;

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

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

Solution 7

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

YES, it does. Once thread0 has set lmp[0] to true, thread1 will be busy waiting in the while(lmp[0]) loop and cannot access CR1 (and vice-versa swapping 0 and 1). If thread1 was already in CR1 when thread0 set lmp[0] to true, then necessarily it is thread1 that will be busy waiting in the while(lmp[1]) loop since lmp[1] must be already true by the time thread1 reaches CR1. This is because lmp[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 lmp flags are already true upon starting line 1, OR if lines 1 and 2 get interleaved (0-1)-1(0)-2(1)-2(1)-

Question 7 (cont)

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

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?

Question 7-b
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 as true
- Then execute thread0's lines 3, 5, 6, 5, 6, 5, 6, 5, 6.
- Thread1 is trapped into the small loop on lines 5-6 (it entered the big loop because Baum[0] was true and the small loop because Baum was 0).
- Now, resume thread0, which is going to execute lines 2, 1, 1, and 12.
- At this point, thread1 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 look did NOT change in that erroneous code: it remained 0.