CSE 421/521 - Operating Systems
Spring 2018

Lecture - XII
Midterm - I Review

Tevfik Koşar

University at Buffalo

March 8th, 2018

Midterm-I Exam

March 13th, Tuesday

3:30pm - 4:50pm

@NSC 201

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

3. Processes

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

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 Prevention
- Deadlock Recovery
- Deadlock Avoidance

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 2-d

What approach to dealing with deadlock does the “Wait-for Graphs” implement (**prevention/avoidance/detection/recovery**)?

Solution 2-d

What approach to dealing with deadlock does the “Wait-for Graphs” implement (**prevention/avoidance/detection/recovery**)?

Deadlock detection

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.

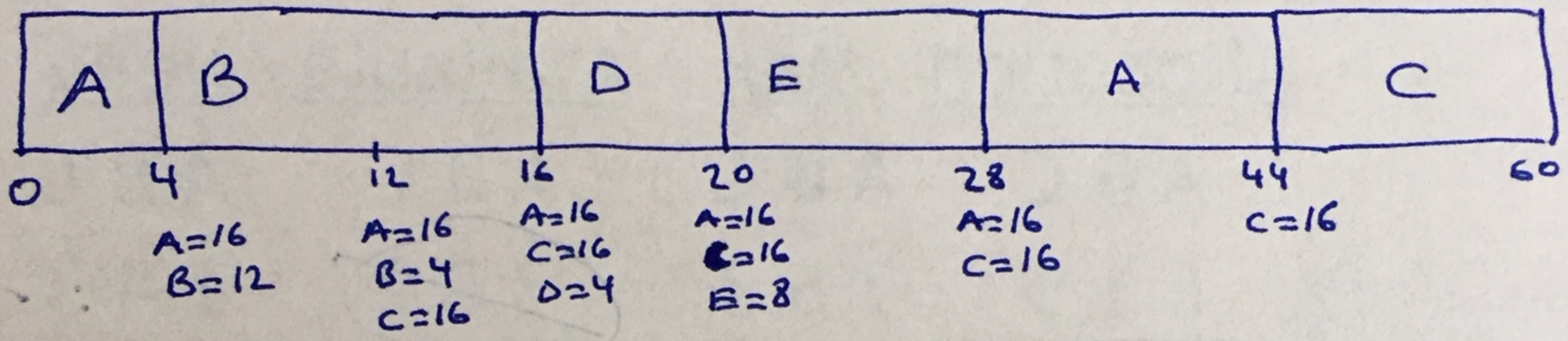
a) Draw Gantt chart illustrating the execution of these processes using Shortest Job First (Preemptive) algorithm.

b) What is the waiting time of each process

c) What is the turnaround time of each process

Solution 3

a)



b)

$W_A = 28-4 = 24$

$W_B = 0$

$W_C = 44-12 = 32$

$W_D = 0$

$W_E = 0$

c)

$T_A = 44-0 = 44$

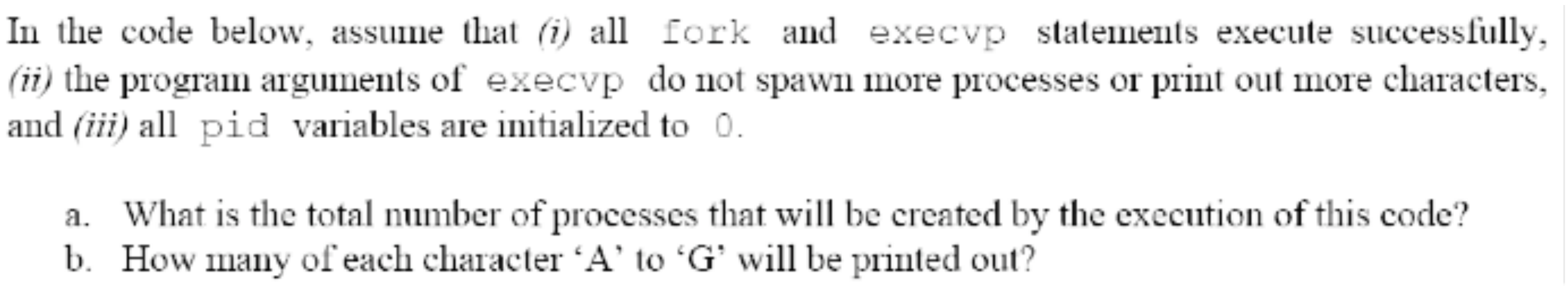
$T_B = 16-4 = 12$

$T_C = 60-12 = 48$

$T_D = 20-16 = 4$

$T_E = 28-20 = 8$

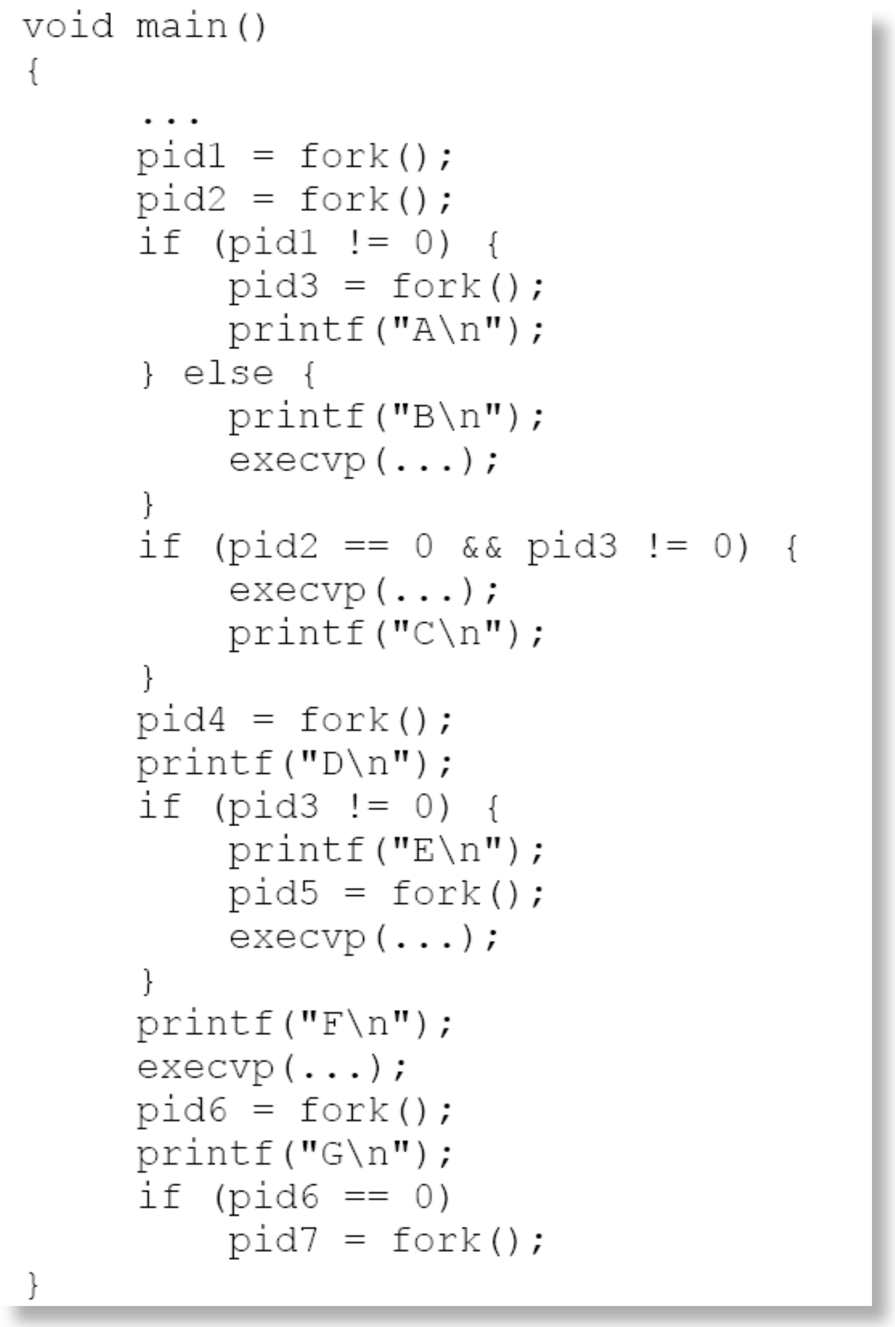
Question 4



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`.

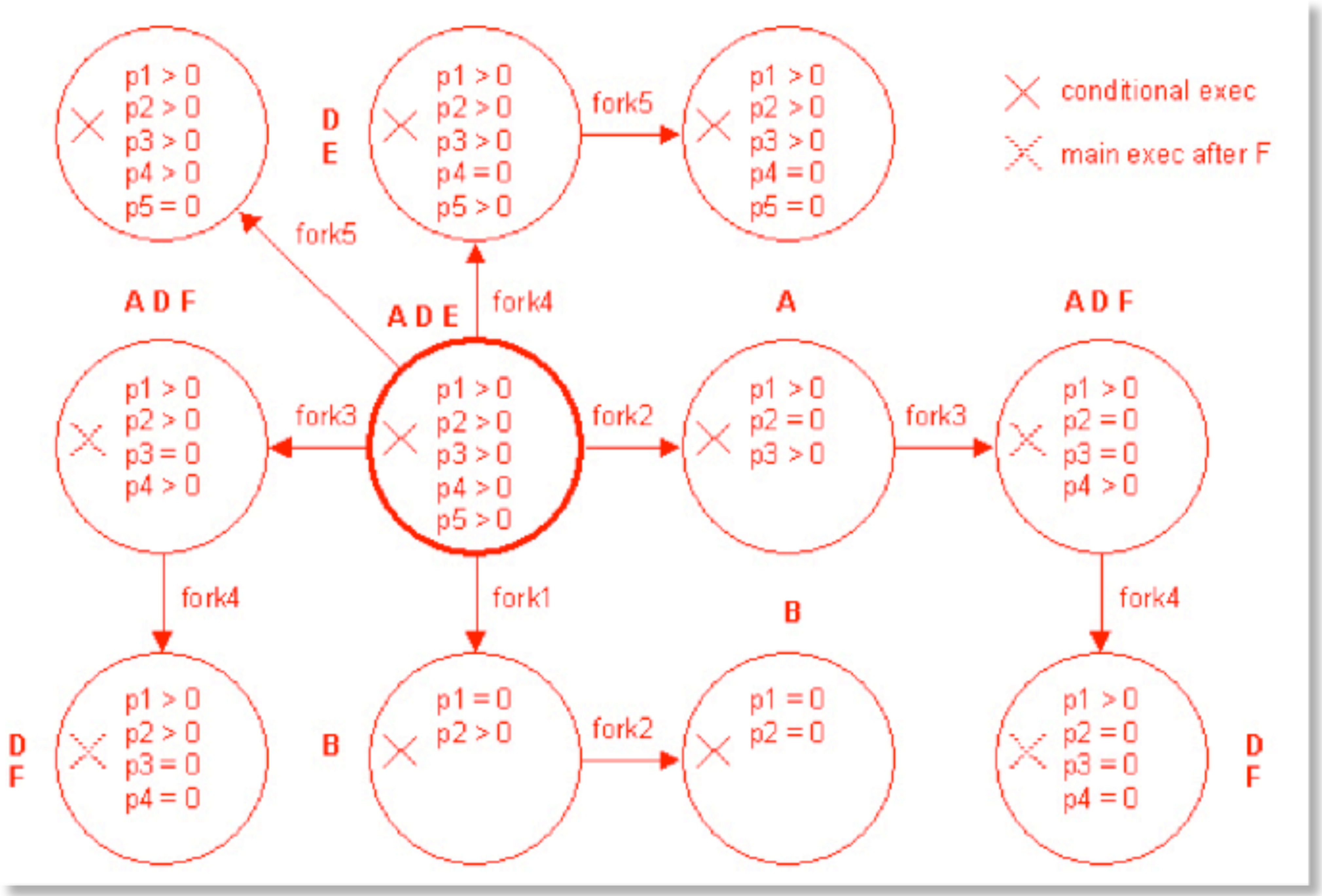
a. What is the total number of processes that will be created by the execution of this code?
b. How many of each character 'A' to 'G' will be printed out?

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


p1 > 0
p2 > 0
p3 > 0
p4 > 0
p5 = 0
D
E
p1 > 0
p2 > 0
p3 > 0
p4 = 0
p5 > 0
fork5
p1 > 0
p2 > 0
p3 > 0
p4 = 0
p5 = 0
conditional exec
main exec after F
fork5
fork4
A D F
A D E
A
A D F
p1 > 0
p2 > 0
p3 = 0
p4 > 0
fork3
p1 > 0
p2 > 0
p3 > 0
p4 > 0
p5 > 0
fork2
p1 > 0
p2 = 0
p3 > 0
fork3
p1 > 0
p2 = 0
p3 = 0
p4 > 0
fork4
fork1
B
fork4
D
F
p1 > 0
p2 > 0
p3 = 0
p4 = 0
B
p1 = 0
p2 > 0
fork2
p1 = 0
p2 = 0
p1 > 0
p2 = 0
p3 = 0
p4 = 0
D
F

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.

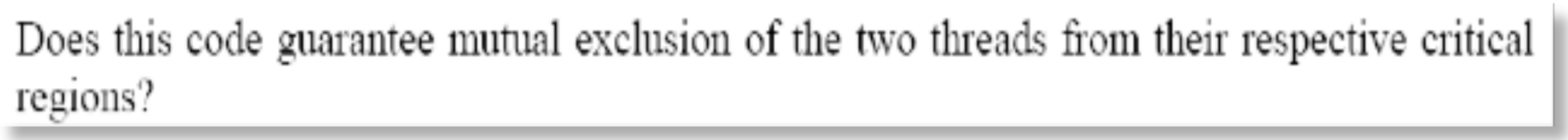
Question 7

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

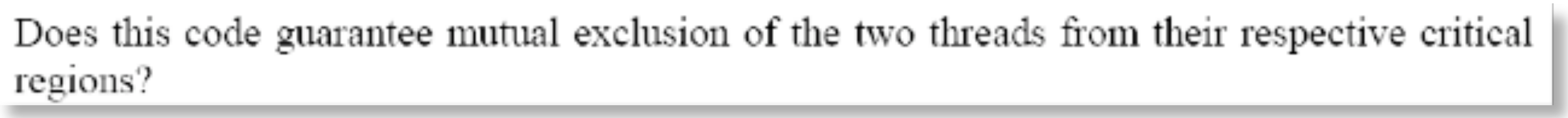
```
void do_thread0()
{
    while (true) {
        lamp[0] = true;
        while (lamp[1]) {
            if (book == 1) {
                lamp[0] = false;
                while (book == 1);
                    /* nothing */
                lamp[0] = true;
            }
        }
        /*** CRITICAL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

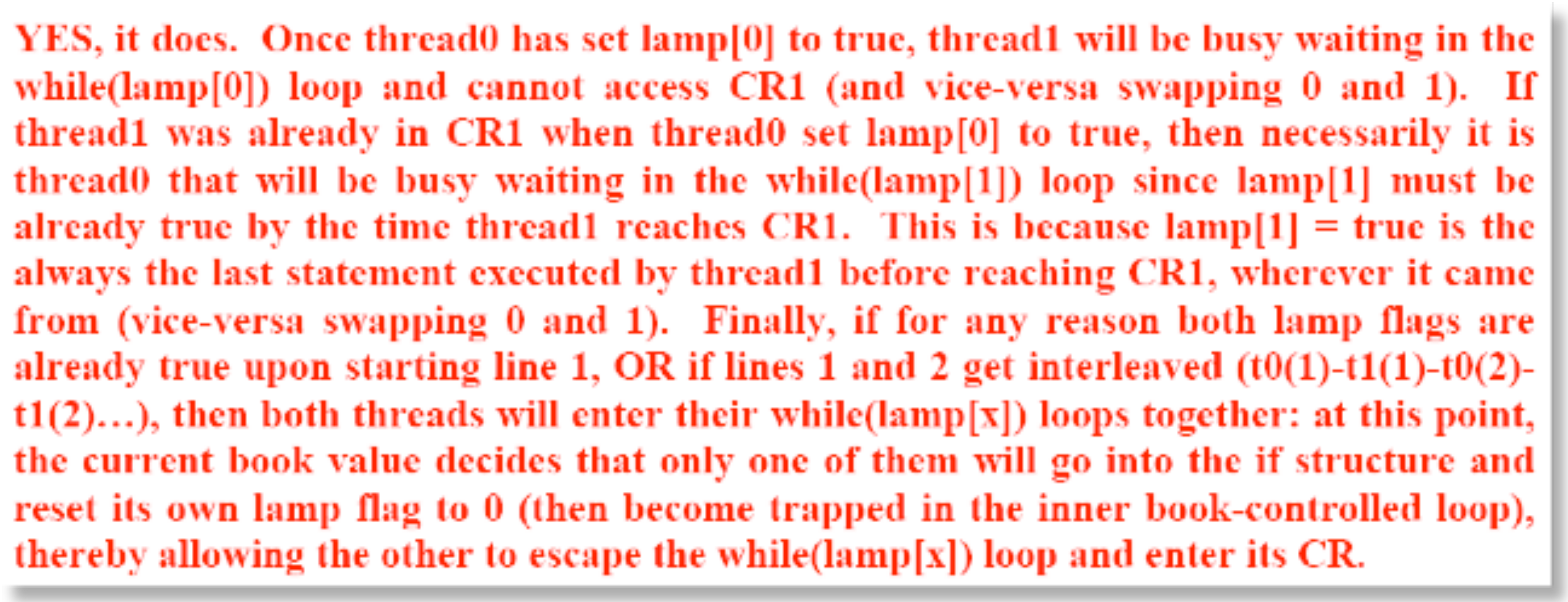
```
void do_thread1()
{
    while (true) {
        lamp[1] = true;
        while (lamp[0]) {
            if (book == 0) {
                lamp[1] = false;
                while (book == 0);
                    /* nothing */
                lamp[1] = true;
            }
        }
        /*** CRITICAL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

Question 7 *(cont)*

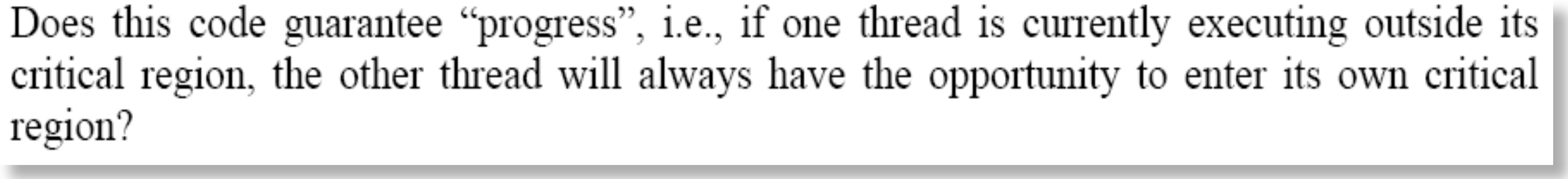

Does this code guarantee mutual exclusion of the two threads from their respective critical regions?

Solution 7

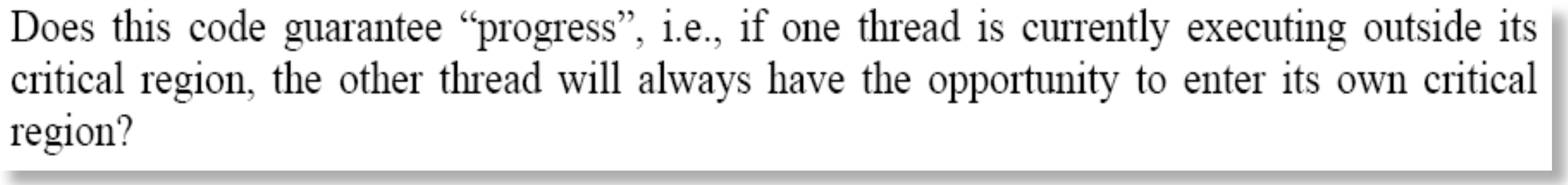

Does this code guarantee mutual exclusion of the two threads from their respective critical regions?

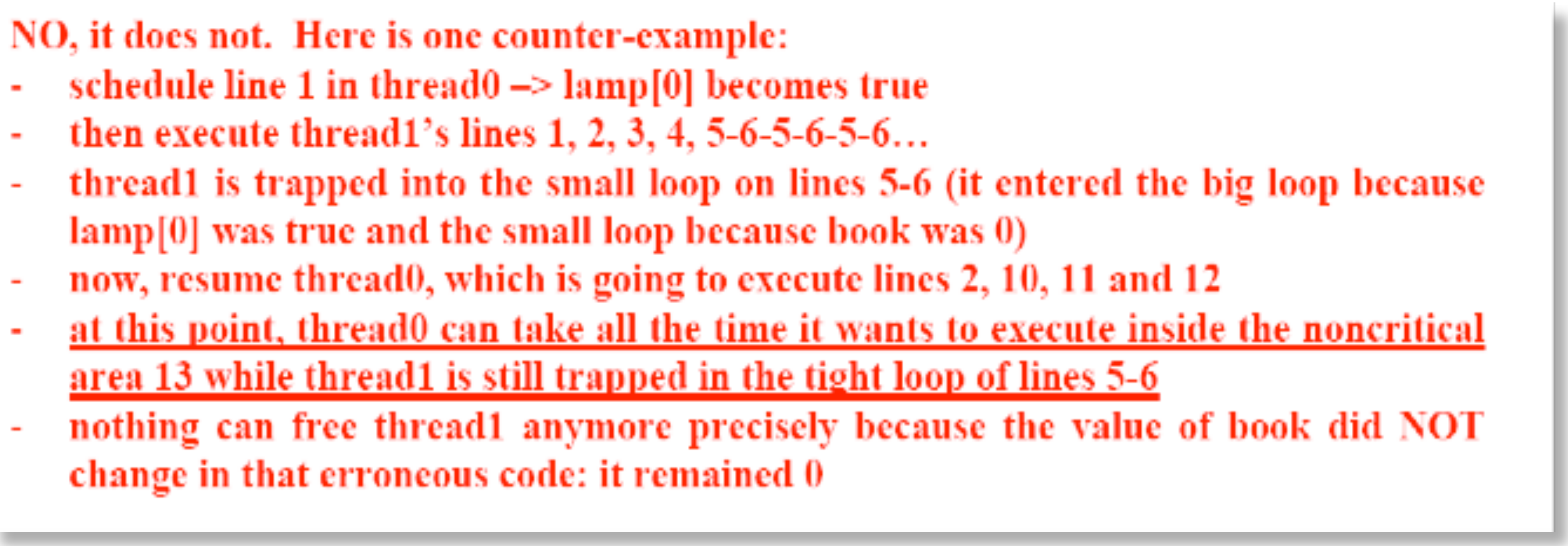

YES, it does. Once thread0 has set lamp[0] to true, thread1 will be busy waiting in the while(lamp[0]) loop and cannot access CR1 (and vice-versa swapping 0 and 1). If thread1 was already in CR1 when thread0 set lamp[0] to true, then necessarily it is thread0 that will be busy waiting in the while(lamp[1]) 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**
- **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**

Question 8

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) $a = 1$ and $T_0 = 100$ ms

b) $a = 0$ and $T_0 = 10$ ms

Solution 8

a) $a = 1$ and $T_0 = 100$ ms

if $a = 1$, $T_{n+1} = a\ t_n = t_n$

Only the actual CPU bursts count (the prediction is always equal to the last actual CPU burst).

b) $a = 0$ and $T_0 = 10$ ms

if $a = 0$, $T_{n+1} = T_n = 10$ ms always

Recent history does not count (the prediction is always equal to $T_0 = 10$ ms)

Question 9

In the code below, three processes are competing for six resources labeled A to F.

a. Using a resource allocation graph (Silberschatz pp.249-251) show the possiblity of a deadlock in this implementation.

b. Modify the order of some of the `get` requests to prevent the possiblity of any deadlock. You cannot move requests across procedures, only change the order inside each procedure. Use a resource allocation graph to justify your answer.

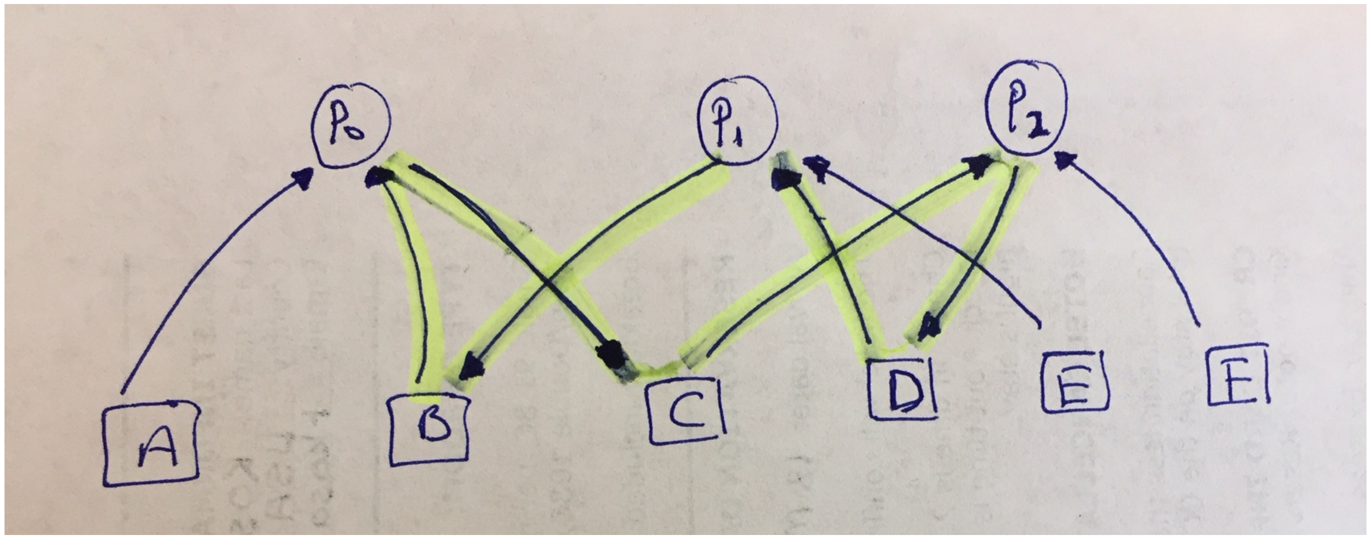
```
void P0()
{
  while (true) {
    get(A);
    get(B);
    get(C);
    // critical region:
    // use A, B, C
    release(A);
    release(B);
    release(C);
  }
}
```

```
void P1()
{
  while (true) {
    get(D);
    get(E);
    get(B);
    // critical region:
    // use D, E, B
    release(D);
    release(E);
    release(B);
  }
}
```

```
void P2()
{
  while (true) {
    get(C);
    get(F);
    get(D);
    // critical region:
    // use C, F, D
    release(C);
    release(F);
    release(D);
  }
}
```

Solution 9

a) According to the Resource Allocation Graph below, there is a cycle between p0->C->p2->D->p1->B->p0. For this reason, we conclude that there is a possibility of deadlock in this system.



b) We can reorder the get requests alphabetically for each process to prevent the deadlock, i.e.:

p0:	**p1:**	**p2:**
get(A)	get(B)	get(C)
get(B)	get(D)	get(D)
get(C)	get(E)	get(F)

Question 10

Consider a system using round-robin scheduling with a fixed quantum *q*. Every context switch takes *s* milliseconds. Any given process runs for an average of *t* milliseconds before it blocks or terminates (burst time).

(a) Determine the fraction of CPU time that will be wasted because of context switches for each of the following cases (your answer should be in terms of *q, s,* and *t*).

i. $t <= q$:

ii. $t >> q$ (t is much greater than q) :

iii. q approaches 0 :

(b) Under what conditions will the wasted fraction of CPU time be exactly 50%?

Solution 10

Consider a system using round-robin scheduling with a fixed quantum *q*. Every context switch takes *s* milliseconds. Any given process runs for an average of *t* milliseconds before it blocks or terminates (burst time).

(a) Determine the fraction of CPU time that will be wasted because of context switches for each of the following cases (your answer should be in terms of *q, s,* and *t*).

i. $t <= q$:

If $q >= t$, the process runs for t and wastes s in one context switch. Thus, with (a) you will waste $s/(t + s)$.

Solution 10 *(cont.)*

ii. t >> q (t is much greater than q) :

When the quantum is shorter than the t , each run t will require t/q context switches, resulting on a waste of (s* t/q) / [t+(s * t/q)] .

iii. q approaches 0 :

As q approaches 0 , efficiency goes to 0 and waste to 100%.

(b) **(4pts)** Under what conditions will the wasted fraction of CPU time be exactly 50%?

Waste is 50% when q = s (if t>>q), or when t=s (if t<=q).

Question 11

Consider the Sleeping Barber Problem, in which there is a barbershop which consists of a waiting room with n chairs and a barber room with one barber chair. If there are no customers to be served, the barber goes to sleep. If a customer enters the barbershop and all chairs are occupied, then the customer leaves the shop. If the barber is busy but chairs are available, then the customer sits in one of the free chairs. If the barber is asleep, the customer wakes up the barber.

Consider the solution to this problem which uses three semaphores:

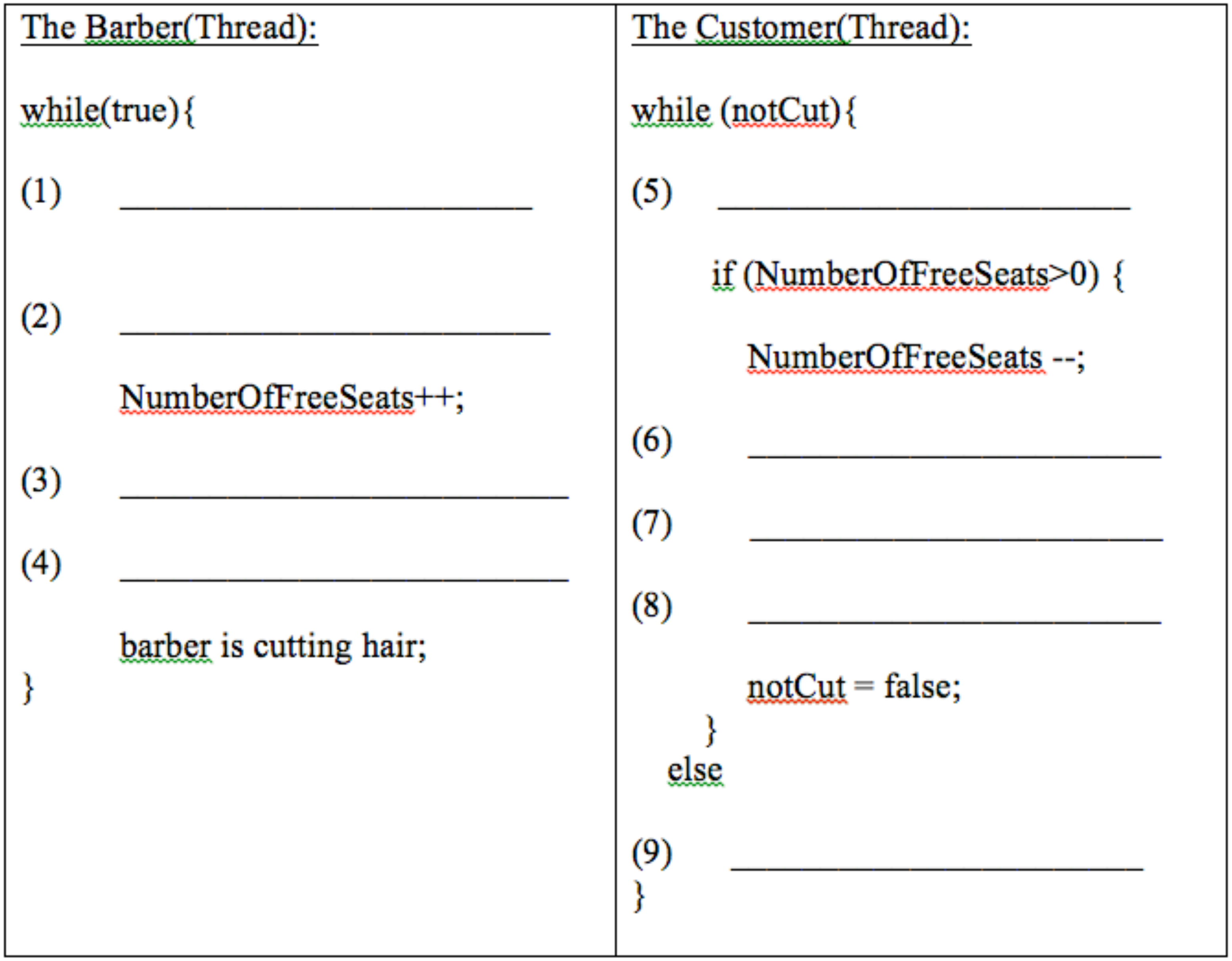
- Semaphore Customers
- Semaphore Barber
- Semaphore accessSeats (mutex)

And an integer to keep track of free seats:

- int NumberOfFreeSeats

Please use the above semaphores correctly in the empty lines of the solution below:

Question 11 (cont.)



The Barber(Thread):

```
while(true){
(1)     ________________________

(2)     ________________________
        NumberOfFreeSeats++;
(3)     ________________________
(4)     ________________________
        barber is cutting hair;
}
```

The Customer(Thread):

```
while (notCut){
(5)     ________________________
    if (NumberOfFreeSeats>0) {
        NumberOfFreeSeats --;
(6)     ________________________
(7)     ________________________
(8)     ________________________
        notCut = false;
    }
  else
(9)     ________________________
}
```

Solution 11

The Barber(Thread):	The Customer(Thread):
while(true){	while (notCut){
(1) P(Customers);	(5) P(accessSeats);
(2) P(accessSeats);	if (NumberOfFreeSeats>0) {
NumberOfFreeSeats++;	NumberOfFreeSeats --;
(3) V(Barber);	(6) V(Customers);
(4) V(accessSeats);	(7) V(accessSeats);
barber is cutting hair;	(8) P(Barber);
}	notCut = false;
	}
	else
	(9) V(accessSeats);
	}

Question 12

Consider a system with three smoker processes and one agent process. Each smoker continuously rolls a cigarette and then smokes it. But to roll and smoke a cigarette, the smoker needs three ingredients: tobacco, paper and matches. One of the smoker processes has paper, another has tobacco and the third has the matches. The agent has an infinite supply of all three materials. The agent places two of the ingredients on the table. The smoker who has the remaining ingredient then makes and smokes a cigarette, signaling the agent on completion. The agent then puts out another two of the three ingredients, and the cycle repeats.

Given below is a solution to the Cigarette-Smokers Problem. Give initial conditions for the semaphores as well as plausible values for the variables j, r & s, such that the agent and smokers are synchronized. Write a couple of sentences on why these initial conditions are necessary and sufficient.

Question 12 *(cont.)*

```
var a: array [0..2] of semaphore {initial condition = ____________ }

    agent: semaphore {initial condition = _____________ }

Agent code :
    repeat
        Set i to a value = rand(3), and j to a value = ____________
        wait(agent);
        signal(a[i]);
        signal(a[j]);
    until false;

Smoker code (for Smoker k):
    repeat
        Set r to a value = ______________, and s to a value = ______________
        wait(a[r]);
        wait(a[s]);
        "smoke"
        signal(agent);
    until false;
```

Solution 12

```
var a: array [0..2] of semaphore {initial condition = ___false_________ }

    agent: semaphore {initial condition =  __true___________ }

 Agent code :
    repeat
        Set i to a value = rand(3), and j to a value = _(i+1) mod 3___________
        wait(agent);
        signal(a[i]);
        signal(a[j]);
    until false;

 Smoker code (for Smoker k):
    repeat
        Set r to a value = _(k+1) mod 3____, and s to a value = __(k+2) mod 3_
        wait(a[r]);
        wait(a[s]);
        "smoke"
        signal(agent);
    until false;
```

Finally...

- Don’t forget to review:

  1. the quiz questions & solutions
  2. the homework questions and solutions