National University of Computer and Emerging Sciences, Lahore Campus
Course: Operating Systems CS2006, Assignment:2

Weight: 3.3 **Total Marks:20**
Submission deadline: 22/21-04-2025

Instruction/Notes:

1. Understanding of the problems is part of the assignments. So, no query please.
2. You will get Zero marks if found any type of cheating.
3. 25 % deduction of over marks on the one-day late submission after due date.
4. 50 % deduction of over marks on the two-day late submission after due date. No submission after two days.
5. **VIVA will be taken, IN-CLASS SUBMISSION.**

Question: 1 **[2+2+2]**

Process Synchronization - Critical-Section Problem with **TestAndSet**.
Suppose we have an atomic operation **TestAndSet**(), which works as if it were implemented by pseudo code such as:

```
Boolean test-and-set (boolean &lock)
{
        temp=lock;
        lock=TRUE;
        return temp;
}
```

Here is the function named: **Function1** which claims to satisfy the critical section problem:

```
1:  void Function1(int i, int j, int n) {
2:      boolean key;
3:      while (TRUE) {
4:          waiting[i] = TRUE;
5:          key = TRUE;
6:          while (waiting[i] && key) { key = test-and-set (&lock); }
7:          waiting[i] = FALSE;
8:          {
9:              / CRITICAL SECTION
            }
            j = (i + 1) % n;
10:         while ( (j != i) && !waiting[j] ) { j = (j + 1) %
11:         n; } if (j == i) { lock = FALSE; } else {waiting[j] =
12:         FALSE; }
            {
13:             // REMAINDER SECTION
            }
14:     }
15: }
```

Here are two processes **PA & PB** which used to call function named **Function1**, and with some shared regions:

<table>
<tr><td>Process A</td><td>Process B</td></tr>
<tr><td colspan="2">Memory region shared by both processes:
#define N 2
boolean waiting[N]; // Assume initialized all FALSE
boolean lock = FALSE;</td></tr>
<tr><td>1. #define ME 0
2. int j = 0;
3. Function1(ME, j, N);</td><td>1. #define ME 1
2. int j = 1;
3. Function1(ME, j, N);</td></tr>
</table>

A- The above solution satisfies which necessary or optional requirement of critical section problem Conditions? Justify your answer.

B- What is the purpose of line6 (While Loop) in **Function1() ?**

C- What is the purpose of line10 (While Loop) in **Function1() ?**

Question: 2-

Multi-Account Banking System Simulation with User-Defined Transactions: [5]

You are tasked with creating a multi-account banking system simulation using POSIX threads and semaphores in C. The program should simulate multiple accounts and allow a user to perform **deposit** and **withdrawal** transactions on any of these accounts by providing input through the command line.

Requirements:

1- Define Accounts:

- Initialize an array of **NUM_ACCOUNTS** accounts, each with a starting balance of 1000.

2- Transaction Functions:

- Write two functions, **deposit**() and **withdraw**(), that:
  i. Accept an account number and an amount as arguments.
  ii. Deposit or withdraw the specified amount from the specified account.
  iii. Use semaphores to ensure that only one transaction can modify an account's balance at a time.

3- Synchronization:

i. Use an array of semaphores, one for each account, to manage thread-safe access to each account.4

4- Main Loop and Exit:

i. Continue prompting the user for transactions until they choose to exit.
ii. Display the final balances for each account before the program terminates.

Expected Output:

```
Enter transaction (account_number deposit/withdraw amount) or 'exit' to stop:
deposit
Account Number (0-4): 2
Amount: 500
Deposited 500 to account 2, New Balance: 1500

withdraw
Account Number (0-4): 2
Amount: 200
Withdrew 200 from account 2, New Balance: 1300

exit

Final account balances:
Account 0: 1000
Account 1: 1000
Account 2: 1300
Account 3: 1000
Account 4: 1000
```

Question:3 **[5]**

Simulate a **ticket booking system** using threads, processes, and semaphores. The program should manage ticket availability at a counter and ensure that no race conditions occur as customers attempt to book tickets.

Requirements:

1. Initial Setup:
   i. Set an initial number of tickets available (e.g., 10 tickets).
   ii. Use a semaphore to control access to the ticket counter.
2. Process Structure:
   i. The main() function should create a parent and a child process.
   ii. Both the parent and child processes will represent ticket agents at different counters.
3. Thread Structure:
   i. Each process will spawn multiple threads (representing customers) that try to book a ticket.
   ii. Each customer thread will check if tickets are available, book one ticket if possible, and then decrement the ticket count.
4. Synchronization:
   i. Use a semaphore to ensure that only one thread accesses the ticket counter at a time to prevent overselling.
5. Program Execution:
   i. The parent process should display the remaining tickets after all threads finish.

Expected Output:

```
Parent process: Ticket counter started.
Ticket booked by customer in thread 0. Tickets left: 9
Ticket booked by customer in thread 1. Tickets left: 8
Ticket booked by customer in thread 2. Tickets left: 7
Ticket booked by customer in thread 3. Tickets left: 6
Ticket booked by customer in thread 4. Tickets left: 5
Child process: Ticket counter started.
Ticket booked by customer in thread 0. Tickets left: 4
Ticket booked by customer in thread 1. Tickets left: 3
Ticket booked by customer in thread 2. Tickets left: 2
Ticket booked by customer in thread 3. Tickets left: 1
Ticket booked by customer in thread 4. Tickets left: 0
Parent process finished. Tickets remaining: 0
```

Question: 4 **[2+1+1]**

Consider a **Multilevel Feedback Queue Scheduler** having three queues numbered from 1 to 3 (see **Fig. A**). The processes are scheduled as follows:

1- A new process enters queue 1 which is served using Round Robin (RR). When it gains CPU, process receives 8 milliseconds. If it does not finish in 8 milliseconds, process is moved to the end of queue 2.

2- If queue 1 is empty, the processes at queue 2 are served using RR and receives 16 milliseconds. If it does not complete, it is preempted and moved to queue 3.

3- Processes in queue 3 are run on a First Come First Serve (FCFS) basis, but are run only when queues 1 and 2 are empty.

4- A process that arrives for queue 2 will preempt a process in queue 3. A process in queue 2 will in turn be preempted by a process arriving for queue 1.

5- If a process does not use up its quantum in queue 2 due to preemption by queue 1, it will keep its current queuing level and be put into the end of the queue. Then, it can still get the same amount of quantum (not remaining quantum) next time when it is picked.

The following set of processes, with the arrival times and the length of the CPU-burst times given in milliseconds, have to be scheduled using this Multilevel Feedback Queue Scheduler:

Process	Arrival Time	Burst Time
P1	0	17
P2	12	25
P3	28	8
P4	36	32
P5	46	18

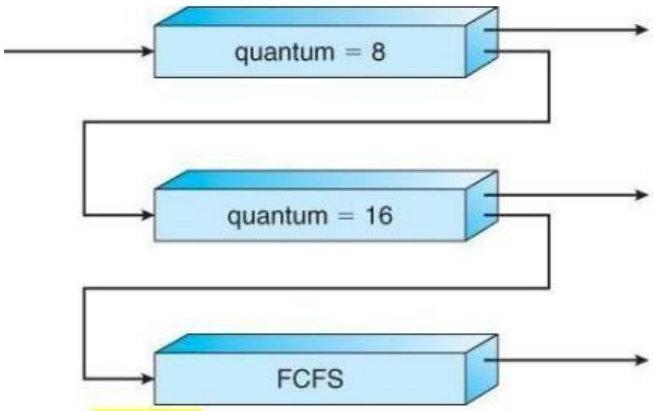

Figure A: Multilevel feedback queue

(a) Draw a Gantt chart illustrating the execution of these processes.
(b) Calculate the number of context switches for the processes.
(c) Calculate the average waiting time and the average turnaround time for the scheduling.