INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR	
	Stamp / Signature of the Invigilator

EXAMINATION (Mid Semester)	SEMESTER (Spring)

Roll Number										Section		Name	
Subject Number	C	S	3	1	2	0	2	Subject Name	*Operating Systems*				

Department / Center of the Student		Additional sheets	

Important Instructions and Guidelines for Students

1. You must occupy your seat as per the Examination Schedule/Sitting Plan.
2. Do not keep mobile phones or any similar electronic gadgets with you even in the switched off mode.
3. Loose papers, class notes, books or any such materials must not be in your possession, even if they are irrelevant to the subject you are taking examination.
4. Data book, codes, graph papers, relevant standard tables/charts or any other materials are allowed only when instructed by the paper-setter.
5. Use of instrument box, pencil box and non-programmable calculator is allowed during the examination. However, exchange of these items or any other papers (including question papers) is not permitted.
6. Write on both sides of the answer script and do not tear off any page. **Use last page(s) of the answer script for rough work.** Report to the invigilator if the answer script has torn or distorted page(s).
7. It is your responsibility to ensure that you have signed the Attendance Sheet. Keep your Admit Card/Identity Card on the desk for checking by the invigilator.
8. You may leave the examination hall for wash room or for drinking water for a very short period. Record your absence from the Examination Hall in the register provided. Smoking and the consumption of any kind of beverages are strictly prohibited inside the Examination Hall.
9. Do not leave the Examination Hall without submitting your answer script to the invigilator. **In any case, you are not allowed to take away the answer script with you.** After the completion of the examination, do not leave the seat until the invigilators collect all the answer scripts.
10. During the examination, either inside or outside the Examination Hall, gathering information from any kind of sources or exchanging information with others or any such attempt will be treated as **'unfair means'**. Do not adopt unfair means and do not indulge in unseemly behavior.

Violation of any of the above instructions may lead to severe punishment.

Signature of the Student

To be filled in by the examiner

Question Number	1	2	3	4	5	6	7	8	9	10	Total
Marks Obtained											

Marks obtained (in words)	Signature of the Examiner	Signature of the Scrutineer

CS31202/CS30002 Operating Systems, Spring 2023–2024

Mid-Semester Test

21–February–2024 | 02:00pm–04:00pm | Maximum marks: 40

[*Write your answers in the question paper itself. Be brief and precise. Answer all questions.*]

Do not write anything on this page.

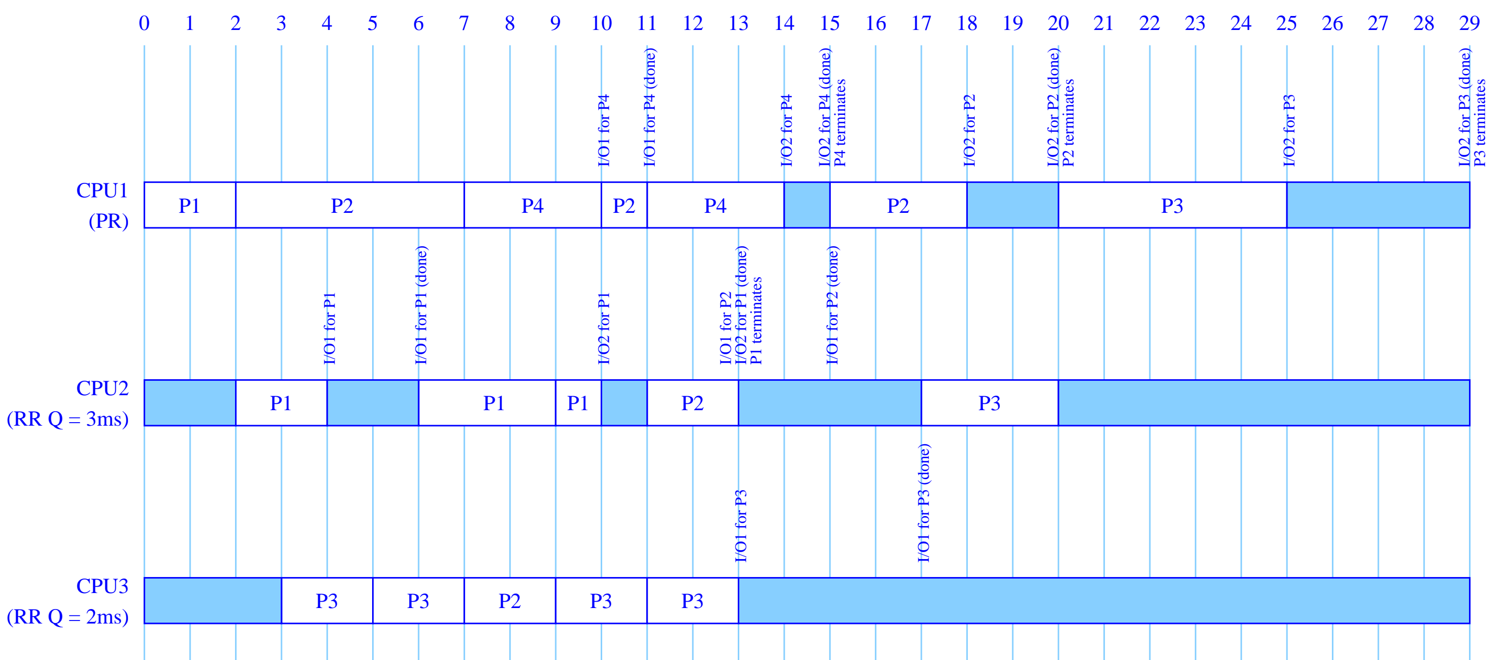
---

1. (a) Consider a multi-processor system with three processors CPU1, CPU2, and CPU3. All the three processors share a common ready queue, and each processor internally runs its own scheduling algorithm. Whenever appropriate (like a processor is free, or it is running a low-priority process), the internal scheduler runs to schedule the next process from the common ready queue. As the internal scheduling algorithm, CPU2 and CPU3 use round-robin scheduling with time quanta 3ms and 2ms, respectively, whereas CPU1 uses the preemptive priority scheduling algorithm. CPU1 considers the common ready queue as a priority queue, whereas CPU2 and CPU3 considers this queue as a FIFO queue. If CPU1 runs a low-priority process, and a high-priority process arrives, the priority scheduler of CPU1 suspends the currently running process, and schedules the high-priority process (even if the other processors are free). If multiple processors are free, the free processor with the lowest index gets the opportunity to run its internal scheduler, and schedules the next process. Assume that the round-robin scheduler of CPU2 or CPU3 always inserts a preempted process (when its time quantum is over) to the ready queue (even if the queue was empty before the insertion). Note that one process may run on multiple processors during its lifetime.

Consider four processes arriving to the system at the arrival times and with priorities as specified in the following table (lower priority number means higher priority, so P4 is the highest-priority process, and P1 is the lowest-priority process). Assume that each process requires two CPU bursts and two I/O bursts for completion (with their respective durations as specified in the table). All times are in ms.

Process	Arrival Time	Priority	First CPU burst	First I/O burst	Second CPU burst	Second I/O burst
P1	0	4	4	2	4	3
P2	2	2	10	2	3	2
P3	3	3	8	4	8	4
P4	7	1	3	1	3	1

Draw the Gantt charts for the three processors. In each Gantt chart, clearly mark (alongside the CPU usage by the processes) the time instances showing (i) when a process starts and finishes its first I/O, (ii) when a process starts and finishes its second I/O, (iii) when a process terminates (after its second I/O), and (iv) the durations when the CPU remains idle. **(4)**



For each process, compute the waiting time and the turnaround time. **(2)**

Compute the percentage of idle time for each processor. The percentage calculation for each processor should use (in the denominator) the total duration starting at the time of the first arrival of a job (to the system, not to that processor) and ending at the time of the last removal of a job (from the system, not from that processor). **(2)**

(b) Consider N processes sharing a single CPU in a round-robin fashion ($N \geqslant 2$). Assume that each context switch takes S ms and that each time quantum is Q ms. For simplicity, assume that processes never block on any event. Find the maximum value of Q (as a function of N, S, and T) such that no process will ever wait in the ready queue for more than T ms. **(2)**

Solution After a process is preempted at the end of its time quantum, there will be N context switches and $N-1$ time quanta for the remaining $N-1$ processes before that process is scheduled again. So the total wait time of the process in the ready queue is $NS + (N-1)Q$. We want $NS + (N-1)Q \leqslant T$, that is, $Q \leqslant \frac{T}{N-1} - \left(\frac{N}{N-1}\right) S$, that is, $Q_{max} = \frac{T}{N-1} - \left(\frac{N}{N-1}\right) S$.

2. (a) Consider the following solution for the critical section problem between two processes P_0 and P_1 that share the following variables.

```
boolean flag[2];    /* initialized to false */
int turn;           /* initialized to 0 or 1 */
```

The structure of the Process P_i ($i = 0$ or 1) is given below. The other process is called P_j, where $j = 1 - i$.

```
do {
   /* ENTRY SECTION */
   flag[i] = true;
   while (turn == j) {
      while (flag[j]) { }   /* busy wait */
      turn = i;
   }

   /* CRITICAL SECTION */
   . . .

   /* EXIT SECTION */
   flag[i] = false;

   /* REMAINDER SECTION */
   . . .
} while (1);
```

Prove/Disprove with proper justification: "The above solution guarantees mutual exclusion for the critical section problem." Assume that there is no swap of instructions by the compiler or the hardware. **(8)**

Solution False. Assume that we are in a timeshared (round robin) single-CPU situation. We have the initialization **`flag[0] = flag[1] = false`**. Suppose that we have the initialization **`turn = 0`**. Now, think of the following sequence of events.

1. P_1 is scheduled first. It sets **`flag[1] = true`**. Since **`turn == 0`**, P_1 enters the outer while loop. Since P_0 is yet to be scheduled, P_1 sees **`flag[0] = false`**, and comes out of the inner while loop.
2. P_1 is now preempted (before setting **`turn = 1`**).
3. P_0 is scheduled. P_0 still sees **`turn = 0`**, so its outer while loop breaks, and P_0 enters its critical section.
4. Inside the critical section, P_0 is preempted, and P_1 is rescheduled.
5. P_1 now sets **`turn = 1`**, goes to the top the outer while loop.
6. Since **`turn`** is favorable for P_1, the loop is broken, and P_1 too enters its critical section.

(b) Consider a system which implements (i) preemptive priority-based CPU scheduler and (ii) Peterson's solution for mutual exclusion. In such a system, is it possible that a high-priority process gets delayed indefinitely because of the presence of lower-priority processes? Justify. **(2)**

Solution Yes. It is possible. Suppose that P1 is of lower priority than P0. At some point of time P1 is running, and P0 is yet to arrive. P1 is about enter its critical section for the first time, and sets `flag[1] = true`. Precisely at that point of time, P0 arrives, and P1 is preempted. P0 starts running, and some time later it plans to enter its critical section. It sets `flag[0] = true` and `turn = 1`. Now, P0 sees `flag[1] = true` and `turn = 1`, and enters the Peterson loop. P1 will never be scheduled in the presence of P0, so P1 can never set `flag[1] = 0` (after its critical section), and the Peterson loop for P0 becomes an infinite one.

3. You have two accounts in a bank. The amounts you have in these accounts are stored in the shared variables x and y, respectively (in Rupees, assumed integers). At the beginning of a day, you have $x = 10000$ and $y = 5000$. During the day, the following updates take place on your accounts. You go to the bank, manually withdraw Rs. 1000 from the second account, and then manually transfer Rs. 2000 from the second account to the first account. At the same time, an online bank transfer (from an external account) deposits Rs. 4000 to your second account. A process P1 handles the manual updates, whereas a second process P2 handles the online update. In short, the two processes proceed as follows.

```
shared int x = 10000;
shared int y = 5000;
```

P1	P2

`y -= 1000;`
`x += 2000;`
`y -= 2000;`
	`y += 4000;`

The processes P1 and P2 run concurrently on a server with a preemptive round-robin scheduler. Therefore race conditions may happen.

(a) In the evening (that is, strictly after the termination of both the processes P1 and P2), you check your account balances. What are the possible values of x and y that you can see? Justify each possibility and all the ways of arriving at each possibility. **(6)**

Solution Case 1: Everything goes as expected (no race condition): $x = 12000$ and $y = 6000$.

Case 2: Race between P2 and the first assignment of **y** of P1.

(i) P1 wins the race (P2 writes later): $x = 12000$ and $y = 7000$
(ii) P2 wins the race (P1 writes later): $x = 12000$ and $y = 2000$

Case 3: Race between P2 and the second assignment of **y** of P1.

(i) P1 wins the race (P2 writes later): $x = 12000$ and $y = 8000$
(ii) P2 wins the race (P1 writes later): $x = 12000$ and $y = 2000$

Case 4: Race between P2 and all the three lines of P1.

(i) P1 wins the race (P2 writes later): $x = 12000$ and $y = 9000$
(ii) P2 wins the race (P1 writes later): $x = 12000$ and $y = 2000$

(b) Now, suppose that x and y are atomic integers, and the increment (**+=**) and decrement (**-=**) operations are implemented as atomic add and subtraction operations, respectively. Are you guaranteed to see the correct account balances in the evening? If yes, justify. If not, explain how the calculations may go wrong. **(2)**

Solution Yes, we are guaranteed to see the correct balances if atomic operations are used. Atomic operations imply that when P1 or P2 updates **y**, it cannot be preempted. A failed update attempt, on the other hand, does not affect **y**.

(c) Implement the atomic add operation (by a constant amount) on an atomic integer, using the compare-and-swap hardware instruction. (A subtraction is adding the negative of the second operand, so there is no immediate necessity to have an atomic subtraction operation.) **(2)**

Solution

```
atomic_add ( atomic int *x, const int a )
{
    int temp;
    do {
        temp = *x;
    } while (compare_and_swap(x, temp, temp + a) != temp);
}
```

4. Dr. Foosycian (MBBS, MD, FRCP) is a busy doctor. In his chamber, he attends to only 25 patients and 3 medical sales representatives per day. The doctor gives priority to the patients, and never serves a sales representative so long as there are waiting patients. No meeting with the doctor (with a patient or a sales representative) can be preempted. Multiple visitors cannot be in the doctor's chamber at the same time.

A new visitor first collects a token (given in the sequence $1, 2, 3, \ldots$) provided that the doctor's daily quota in that visitor's category (patient or sales-rep) is not full. If the quota is already full, the visitor leaves. Otherwise, the visitor informs the doctor about his/her arrival, and then waits outside the doctor's chamber until the doctor calls him/her.

You need to design three processes **`doctor()`**, **`patient()`**, and **`salesrep()`** for simulating the workings of Dr. Foosycian, a patient, and a sales representative, respectively. The processes should be cooperative in the sense that they will follow the doctor's protocol strictly and will not block any resource when not needed. Shared memory and semaphores (and no other IPC primitives) are to be used to synchronize the processes and to provide mutual exclusion. Follow the guidelines given below, and fill up the details of the three processes.

(**Hint:** This problem is similar to the sleeping barber's problem with three differences. First, the barber has only one type of customer, whereas the doctor has two. Second, the barber keeps on sleeping and cutting hair for ever, whereas our doctor stops after his daily quota. Finally, customers in the barber's shop have to leave for unavailability of empty chairs, whereas visitors to our doctor have to leave if they come too late.)

The following shared variables are to be used with the indicated initializations and implications.

```
shared int last_token_no_patient   = 0;  // Count of tokens already given to patients
shared int last_token_no_salesrep  = 0;  // Count of tokens already given to sales-reps
shared int no_of_waiting_patients  = 0;  // Count of patients that are waiting now
shared int no_of_waiting_salesreps = 0;  // Count of sales-reps that are waiting now
```

In addition, four semaphores are to be used and initialized as follows. The first of these four semaphores is to be used as a counting semaphore, whereas the remaining three are to be used as binary semaphores (that is, as mutex locks).

```
semaphore sem_doctor   = 0;  // Only the doctor waits on this semaphore
semaphore mtx_patient  = 0;  // All patients wait on this semaphore
semaphore mtx_salesrep = 0;  // All sales representatives wait on this semaphore
semaphore mtx_counts   = 1;  // For mutual exclusion of accessing shared variables
```

Do not use any variables or semaphores other than those mentioned above.

First, write (on the next page) the pseudocode of the process for the doctor. The doctor stays in his chamber until he attends to 25 patients and 3 sales representatives. Assume that on each day, at least those many visitors (in each category) come to meet the doctor, and that no visitor having a token leaves without meeting the doctor. Your code must contain appropriate synchronization and mutual-exclusion primitives.

```
doctor ( )
{
    while (true) {
        /* If the doctor is done for the day, break the loop */        (2)

        wait(&mtx_counts);
        if ( (last_token_no_patient == 25) && (last_token_no_salesrep == 3) &&
             (no_of_waiting_patients == 0) && (no_of_waiting_salesreps == 0) ) {
            signal(&mtx_counts);
            break;
        }

        /* Wait for the next visitor */                                (1)

                              wait(&sem_doctor);
        ____________________________________________________________

        /* Serve a patient or a sales representative as per doctor's strategy */   (3)

        wait(&mtx_counts);

        if (no_of_waiting_patients > 0) {
            --no_of_waiting_patients;
            signal(&mtx_patient);
            signal(&mtx_counts);
            attend_to_patient();
        } else {
            --no_of_waiting_salesreps;
            signal(&mtx_salesrep);
            signal(&mtx_counts);
            attend_to_salesrep();
        }

    }
}
```

Then, write the pseudocode of the process for a patient. A patient first tries to get the next token. If no tokens are available, he/she leaves. Otherwise, he/she takes the token, notifies the doctor about his/her arrival, and waits until the doctor calls him/her. Include suitable synchronization and mutual-exclusion primitives. **(2)**

```
patient ( )
{
    wait(&mtx_counts);

    if (last_token_no_patient == 25) {
        signal(&mtx_counts);
    } else {
        ++last_token_no_patient;
        ++no_of_waiting_patients;
        signal(&mtx_counts);
        signal(&sem_doctor);
        wait(&mtx_patient);
        get_medical_advice_from_doctor();
    }
}
```

Finally, write the pseudocode of the process for a sales representative (similar to that for a patient). **(2)**

```
salesrep ( )
{
    wait(&mtx_counts);

    if (last_token_no_salesrep == 3) {
        signal(&mtx_counts);
    } else {
        ++last_token_no_salesrep;
        ++no_of_waiting_salesreps;
        signal(&mtx_counts);
        signal(&sem_doctor);
        wait(&mtx_salesrep);
        sales_promo();
    }
}
```