INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

Stamp / Signature of the Invigilator

EXAMINATION (Mid Semester / ~~End Semester~~)	SEMESTER (~~Autumn~~ / 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 of 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 Operating Systems

Mid-Semester Examination

24-Feb-2025, 02:00pm–04:00pm

Maximum Marks: 80

Instructions to students

- Please write in the spaces provided in the question paper itself. Be brief and precise.
- For rough work, you can use the extra blank pages provided at the end. If you need additional space for rough work, please ask for supplementary sheets from the invigilators.
- Do not write anything on this page. Questions start from the next page (Page 3).

1. **[*Process-synchronization tools*]**

(a) A **barrier** (not the same as a memory barrier taught in the class) is often used as a high-level primitive for process synchronization. Suppose that $n \geqslant 2$ processes want to synchronize at certain points. For example, they may be sharing the work of a multiplication $P = MN$ of very large shared matrices. If they write the entries of P in mutually distinct locations, mutual exclusion is not necessary. However, until all of the processes finish their parts in the multiplication task, work on the product P cannot start (shared again by the same processes). A barrier is initialized to n. After finishing the assigned part in the multiplication task, each process joins the barrier. The first, second, . . . , $(n-1)$-th processes wait on the barrier. As soon as the n-th process joins the barrier, all of the n processes leave the barrier, and start their respective post-multiplication works.

In this exercise, we make a **hardware-level implementation** of a barrier B. It consists of two shared atomic integer-valued variables `join` and `leave` (standing for the number of processes yet to join or yet to leave the barrier). Neither of these integers should ever get a value < 0. When B is not in use, both `join` and `leave` should be 0. A barrier B supports the following two functions.

`barrier_init(&B,n)` is used to initialize a barrier B by a positive integer value n. If at the time of this initialization, either `join` or `leave` holds a value > 0, then the barrier is in use, and the initialization must fail. If not, both `join` and `leave` will get the value n. If multiple processes want to initialize an unused barrier simultaneously, only one of these processes should succeed; all others will fail.

`barrier_wait(&B)` is to be invoked by a process after a successful initialization of B, say, by the value n. Exactly n processes must call `barrier_wait(&B)` for lifting the barrier. If at the time of calling the function, `join` does not store a positive value, the attempt fails (to prevent more than n processes from joining the barrier, or to prevent a process from joining an uninitialized barrier). Otherwise, the value of `join` is decremented atomically by 1. All of the n processes that join the barrier must wait until `join` becomes 0. When `join` becomes 0, all of these waiting processes can proceed forward simultaneously. But before doing so, the processes must leave the barrier one by one. This is done by each process atomically decrementing `leave` by 1. Every leaving process must see `leave` > 0 before the decrement, otherwise the operation fails. Strictly after all the joined processes leave, the barrier is ready again for (re-)initialization.

Implement below the primitives `barrier_init()` and `barrier_wait()` using hardware-level ***compare-and-swap*** instructions and busy waits. Your implementation must handle all error conditions mentioned above. **No credit** for an implementation based on any other software/hardware primitive. Each function should return a boolean status (SUCCESS or FAILURE). **[4 + 6]**

```
typedef struct {
    atomic int join;   /* number of processes yet to join the barrier */
    atomic int leave;  /* number of processes yet to leave the barrier */
} barrier;

shared barrier B;      /* Assume that this declaration sets both join and leave in B to 0 */

boolean barrier_init ( barrier *B, int n )  /* Assume that n is not a shared variable */
{




        if (n <= 0) return FAILURE;

        if (compare_and_swap(&(B → leave), 0, n) != 0)
            return FAILURE;

        if (compare_and_swap(&(B → join), 0, n) != 0)
            return FAILURE;

        return SUCCESS;




}
```

```
boolean barrier_wait ( barrier *B )
{
        int temp;

        do {
            temp = B → join;
            if (temp <= 0) return FAILURE;
        } while (compare_and_swap(&(B → join), temp, temp - 1) != temp);

        while (B → join != 0) ;  /* No need to protect this read */

        do {
            temp = B → leave;
            if (temp <= 0) return FAILURE;
        } while (compare_and_swap(&(B → leave), temp, temp - 1) != temp);

        return SUCCESS;

}
```

(b) Consider the following two **software solutions** to solve the critical-section problem for two processes P_0 and P_1. The code for P_i is given below, where $i \in \{0, 1\}$. The other process is called P_j, where $j = 1 - i$.

```
shared boolean flag[2] = {0, 0};
shared int turn = 0;

while (1) {
    /* Entry section */

    flag[i] = true;
    if (flag[j]) {
       if (turn == j) {
          flag[i] = false;
          while (turn == j) ;
          flag[i] = true;
       }
    }

    /* critical section */

    /* Exit section */

    turn = j;
    flag[i] = false;

    /* remainder section */
}
```

Solution 1

```
shared boolean flag[2] = {0, 0};
shared int turn = 0;

while (1) {
    /* Entry section */

    flag[i] = true;
    while (flag[j]) {
       if (turn == j) {
          flag[i] = false;
          while (turn == j) ;
          flag[i] = true;
       }
    }

    /* critical section */

    /* Exit section */

    turn = j;
    flag[i] = false;

    /* remainder section */
}
```

Solution 2

Assume that the compiler (or the hardware) does not modify the structure of the above codes. Prove or disprove with justification which of these software solutions (if any) support(s) mutual exclusion. **[5 + 5]**

Solution 1: *False*.

Suppose that at some point of time (like at the beginning) `turn` stores the value 0. At that point, both P_0 and P_1 plan to enter their respective critical sections, and the following sequence of events happens.

1. P_1 sets `flag[1] = true`.
2. Since `flag[0]` is still false, the condition in the outer if statement is false for P_1, so P_1 enters its critical section.
3. P_0 sets `flag[0] = true`.
4. Since `turn` is still 0, the condition in the inner if statement is false for P_0, so P_0 too enters its critical section.

Solution 2: *True*

Suppose that both P_0 and P_1 plan to enter their critical sections (almost) at the same time. Consider two cases.

Case 1: One of the processes (say, P_0) has set `flag[0] = true`, but P_1 is yet to set `flag[1] = true`. P_0 does not enter the outer-loop body, and goes to its critical section irrespective of the value of `turn`. Then, so long as P_0 is not in its exit section, P_1 sees `flag[0] = true`, and enters its outer loop. Now, if `turn` is 0, P_1 is stuck in its inner while loop. If turn is 1, it is stuck in its outer while loop.

Case 2: The processes have both set their `flags` to 1, and check the condition of their outer while loops. They both enter the loop, because both `flags` are true. But `turn` is not modified in the entry section, and it can be either 0 or 1 (cannot be both). But then, it is impossible for both the processes to bypass their inner while loops together. Indeed, the process whose `turn` it is will enter. The other one will be stuck in the inner while loop.

2. [*Process scheduling*]

Consider a dual-core symmetric multiprocessor system with a common ready queue for both the cores (call them Core0 and Core1). On both the cores, Round-Robin Scheduling is used, each with a time quantum of $q = 5$ (all times are in ms in this exercise). Suppose that a process P is at the front of the common ready queue. The scheduler uses the following policy.

Case 1: Both Core0 and Core1 are free, and P is a new arrival. In this case, P is scheduled to Core0.

Case 2: Both Core0 and Core1 are free, and P joined the ready queue after a CPU timeout or completion of IO. In this case, P will be scheduled to the core where it ran last. The reason for this is that if P migrates to the other core, then there is a cache re-population penalty (see Assumption 1 below for more on this).

Case 3: Only one of the cores is free. Then, the scheduler is forced to schedule P to that core. That may result in a process migration from one core to the other, and if so, will incur cache re-population penalty.

Case 4: Both Core0 and Core1 are busy. It is not time for scheduling.

In addition to these scheduling rules, make the following two assumptions.

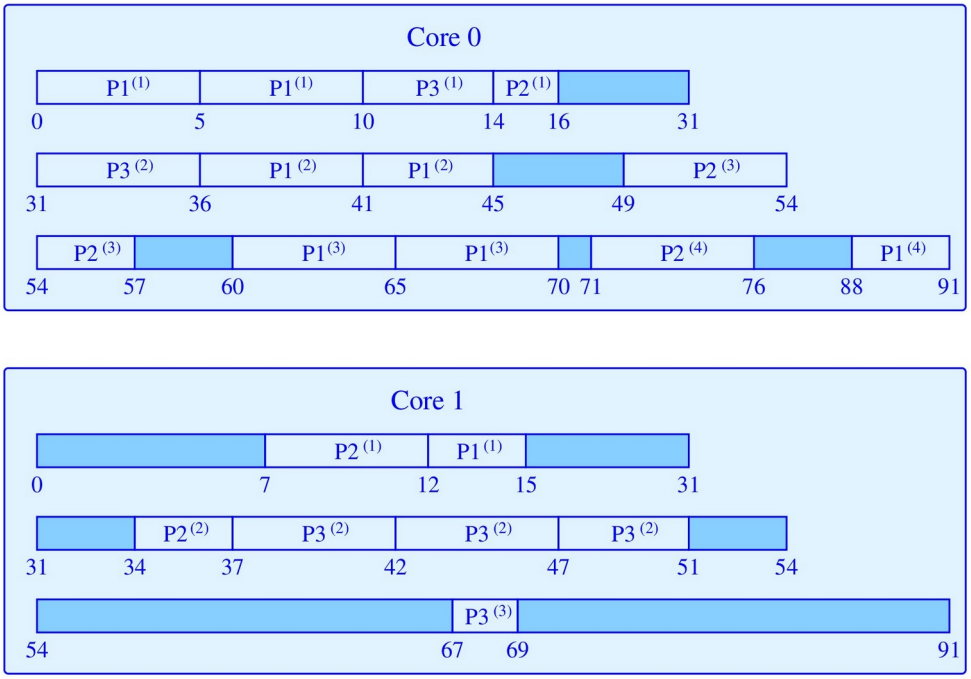
Assumption 1: Each process migration from one core to the other incurs a cache re-population time of 1 ms. Every time this happens, the current (remaining) CPU-burst time of the process increases by 1. Assume also that the initial cache-population time needed when the process is scheduled for the very first time is already included in the first CPU burst time of that process, that is, you do not need to add 1 ms at the beginning. Later on, each migration of each process will call for an additional CPU overhead of 1 ms.

Assumption 2: When two or more processes want to join the ready queue at the same time, they will do so in the increasing sequence of their PID's.

Suppose that the system deals with three processes P1, P2, and P3 with the following details. P1 and P2 exit after their fourth CPU bursts, whereas P3 exits after its third CPU burst. All times are in ms.

Process	PID	Arrival Time	CPU Burst 1	IO Burst 1	CPU Burst 2	IO Burst 2	CPU Burst 3	IO Burst 3	CPU Burst 4
P1	89	0	12	20	8	15	10	18	3
P2	123	7	6	18	2	12	7	14	5
P3	345	9	4	17	18	16	2	–	–

(a) Draw the Gantt chart for both Core0 and Core1. Shade the CPU-idle durations. In the Gantt chart, show the different CPU bursts of the process P_i as $P_i^{(1)}$, $P_i^{(2)}$, $P_i^{(3)}$, . . . (for $i = 1, 2, 3$). **[14]**



(b) From the Gantt chart, compute the turnaround times and wait times of the three processes P1, P2, and P3. Include cache re-population times in the wait time (but not in the running time). Show your calculations and answers in the table below. **[6]**

Process	Running time	Turnaround Time	Wait time
P1	12 + 20 + 8 + 15 + 10 + 18 + 3 = 86	91 – 0 = 91	91 – 86 = 5
P2	6 + 18 + 2 + 12 + 7 + 14 + 5 = 64	76 – 7 = 69	69 – 64 = 5
P3	4 + 17 + 18 + 16 + 2 = 57	69 – 9 = 60	60 – 57 = 3

3. [*Synchronization examples*]

(a) The deadlock-free solution of the Dining Philosophers Problem, taught in the class, uses a semaphore vector `s[]`, initialized to 0, and an integer vector `state[]` storing `THINKING`, `HUNGRY`, or `EATING`. Prof. X proposes alternate implementations of `take_forks()` or `take_chopsticks()` (see Fig. 3(a)) and `put_forks()` or `put_chopsticks()` (see Fig. 3(b)), to solve the Dining Philosophers Problem. Assume that `mutex` is initialized to 1.

```
void take_forks_X (int i)
{
    wait(mutex);
    state[i]=HUNGRY;
    test(i);
    wait(&s[i]);
    signal(mutex);
}
```

Fig. 3(a)

```
void put_forks_X (int i)
{
    wait(mutex);
    test(LEFT);
    test(RIGHT);
    state[i]=THINKING;
    signal(mutex);
}
```

Fig. 3(b)

(i) If Prof. X only replaces the `take_forks_X()` in the original Dining-Philosophers-Problem solution (he uses the original `put_forks()`), explain the effect of the revised solution. **[3]**

Suppose that Philosopher 0 is eating. Philosopher 1 will be blocked without releasing `mutex`. Hence all other philosophers will be blocked too.

(ii) If Prof. X only replaces the `put_forks_X()` in the original Dining-Philosopher-Problem solution (he uses the original `take_forks()`), explain the effect of the revised solution. **[3]**

Even if one philosopher *i* returns back the fork and executes `put_fork_x(i)`, `test(LEFT)` and `test(RIGHT)` will fail to wake up the neighbors.

(b) Consider a set of *n* processes, each of which executes the following piece of code (here we show the code of process P_i). We wish to ensure that all the *n* processes print `"Hello OS"` (in any order), before any process is allowed to print `"Bye OS"`. This solution uses two semaphore variables `x` and `y`, initialized to 1 and 0, respectively. Moreover, the code utilizes a shared integer variable `count`, which is initialized to 0. Complete the code by filling up the (six) blanks. **[6]**

Process P_i

```
printf("Hello OS") ;
wait( ___________x___________ ) ;
count++ ;
if ( ____________count == n____________ )
    ____________signal(y)_____________ ;
signal( ___________x___________ ) ;
wait( ___________y___________ ) ;
_________signal(y)_________ ;
printf("Bye OS") ;
```

(c) The *Cigarette Smokers Problem* is a classic synchronization problem. In this problem, three ingredients are required to construct and smoke a cigarette: tobacco, paper, and matches. There are four actors, each represented by a process. One of the actors is the agent, and the other three are smokers. The agent has an infinite supply of all of the three ingredients (tobacco, paper, and matches). Each of the three smokers has an infinite supply of only one ingredient and nothing else. That is, the first smoker possesses only tobacco, the second smoker only paper, and the third smoker only matches.

The three smoker processes run a loop in an attempt to smoke, which requires that they obtain one unit of both of the ingredients from the table, that they do not possess. For instance, the smoker with tobacco requires to obtain paper and matches from the table. The agent loops repeatedly, randomly choosing a pair of ingredients (say, paper and matches), puts them on the table to make available to the smokers. Each time the agent does this, one of the three smokers should be able to acquire all the three ingredients, and smoke. For example, if the agent chooses paper and matches, then the tobacco-possessing smoker can acquire these two items. Combined with its own supply of tobacco, it can then smoke. Until one smoker starts smoking, the agent waits.

There is a possibility of deadlock, as the agent puts paper and matches on the table. The smoker with tobacco acquires the paper, and the smoker with paper acquires the matches. Both are blocked for the third ingredient, and wait. The agent also waits, since none of the smokers is able to smoke.

In the following, we supply an incomplete solution for the agent, the smoker with tobacco, the smoker with paper, and the smoker with matches. Complete the solution in order to ensure that there should not be any deadlock. **[12]**

The proposed solution uses the following five semaphores.

```
agent = 0;          // Ensures only one set of ingredients is placed at a time by the agent.
tobacco_sem = 0;    // Wakes up the smoker to smoke, who has tobacco (when the other two ingredients are
                       supplied by the agent).
paper_sem = 0;      // Wakes up the smoker to smoke, who has paper (when the other two ingredients are
                       supplied by the agent).
match_sem = 0;      // Wakes up the smoker to smoke, who has matches (when the other two ingredients
                       are supplied by the agent).
mutex = 1;          // Ensures mutual exclusion while accessing the table (to put or get the ingredients).
```

Process agent

```
{
    while (true) {

        wait( ____________mutex____________ ) ;

        choice = randomly choose one pair of ingredients (tobacco and paper), (tobacco and matches), or
                 (paper and matches);

        if (choice == (tobacco and paper)) {

            ____________signal(match_sem)____________ ;   // Wake up the smoker with matches
        }
        else if (choice == (tobacco and matches)) {

            ____________signal(paper_sem)____________ ;   // Wake up the smoker with paper
        }
        else if (choice == (paper and matches)) {

            ____________signal(tobacco_sem)____________ ;  // Wake up the smoker with tobacco
        }

        signal( ____________mutex____________ ) ;


        wait( ____________agent____________ ) ;
    }
}
```

Process smoker_with_tobacco

```
{
    while (true) {

        wait( ____tobacco_sem____ ) ;       // Wait until paper and matches are on the table

        wait( ____mutex____ ) ;

        /* take paper and matches */

        ____signal(agent)____ ;       // Notify agent that ingredients are taken

        ____signal(mutex)____ ;

        smoke();
    }
}
```

Process smoker_with_paper

```
{
    while (true) {

        wait( ____paper_sem____ ) ;       // Wait until tobacco and matches are on the table

        wait( ____mutex____ ) ;

        /* take tobacco and matches */

        ____signal(agent)____ ;       // Notify agent that ingredients are taken

        ____signal(mutex)____ ;

        smoke();
    }
}
```

Process smoker_with_matches

```
{
    while (true) {

        wait( ____match_sem____ ) ;       // Wait until tobacco and paper are on the table

        wait( ____mutex____ ) ;

        /* take tobacco and paper */

        ____signal(agent)____ ;       // Notify agent that ingredients are taken

        ____signal(mutex)____ ;

        smoke();
    }
}
```

4. **[*Deadlock*]**

(a) Consider a system with two semaphores: S1 initialized to 2, and S2 initialized to 0. Three processes P1, P2, and P3 run as follows.

Process P1

```
wait(S1);
wait(S2);
/* Critical Section */
signal(S2);
signal(S1);
```

Process P2

```
wait(S1);
/* Critical Section */
signal(S2);
```

Process P3

```
wait(S2);
/* Critical Section */
signal(S2);
```

Prove or disprove each of the following two statements with arguments.

(i) Two or three of the processes P1, P2, and P3 may enter their critical sections at the same time. **[3]**

False. Process P2 enters its critical section first. Since S2 is initialized to 0, both P1 and P3 (if available) wait on it. Since S1 is initialized to 2, the first waits of P1 and P2 are non-blocking. Once P2 signals S2 after the completion of its critical section, this semaphore becomes 1, allowing only one of P1 and P3 to enter its critical section. After this process is done with its critical section, the other can enter.

(ii) Suppose that we change the last line of Process P2 from `signal(S2);` to `signal(S1);` (the rest of the codes, and the initialization of S1 and S2 remain the same). Then, two or three of the processes P1, P2, and P3 may end up in a deadlock among themselves. **[3]**

False. For a deadlock to happen, a plural number of processes must hold some resource and wait for one or more other resources (the Hold-and-Wait or Circular-Wait condition). In this example, only P1 can be in such a situation.

(b) A computer system uses the Banker's Algorithm to avoid deadlocks. Its current state is shown in the tables below, where P0, P1, P2, P3, P4 are processes, and R0, R1, R2, R3 are resource types. The existing resource vector *E* shows the total number of resource instances available in the system (*before any allocation*). The need matrix *N* represents the additional resource requirements of different processes.

Current Allocation (*C*)

	R0	R1	R2	R3
P0	3	0	1	1
P1	0	1	1	0
P2	1	1	1	0
P3	1	1	0	1
P4	0	0	0	0

Need Matrix (*N*)

	R0	R1	R2	R3
P0	1	1	0	0
P1	0	1	0	2
P2	3	1	0	0
P3	0	0	1	0
P4	2	1	1	0

Existing Resource Vector (*E*)

R0	R1	R2	R3
6	3	4	2

(i) Show that the system is in a safe state. Show the relevant calculations in detail. **[5]**

Initially, available vector $A = [1, 0, 1, 0]$.

First, P3 can run, since Need[P3] $\leqslant A$.
P3 runs, and returns back the resources.
Updated $A = [2, 1, 1, 1]$.

Now, P0 can run, since Need[P0] is $\leqslant A$.
P0 runs, and returns back the resources.
Updated $A = [5, 1, 2, 2]$.

This continues, and we get the safe sequence <P3, P0, P1, P2, P4>.

(ii) What will the system do if Process P4 requests one more unit of Resource R2? Show all the matrices (if this request is granted), and justify. **[5]**

If P4 makes a request for R2, and we allocate it to P4, the updated available vector is $A = [1, 0, 0, 0]$.

The Current Allocation matrix changes to:

	R0	R1	R2	R3
P0	3	0	1	1
P1	0	1	1	0
P2	3	1	0	0
P3	1	1	0	1
P4	0	0	1	0

The Need matrix changes to:

	R0	R1	R2	R3
P0	1	1	0	0
P1	0	1	0	2
P2	1	1	1	0
P3	0	0	1	0
P4	2	1	0	0

Now, there is no process P_i, for which Need$[P_i] \leqslant A$. That is, there is no safe sequence, and the system goes into an unsafe state. Therefore, the request of P4 cannot be granted.

Space for rough work

Space for rough work

Space for rough work