Mohamed Boudiaf University – M'sila
Faculty of Mathematics and Computer Science
Computer Science Department

3nd year degree in Computer Science (IS)
Academic Year 2025-2026

Operating Systems II
Duration: 2h

SOLUTION FOR FINAL EXAM

(Normal Session)

Exercise 1 (6 points)

Consider a system with 4 processes **P1, P2, P3, P4**, and 3 types of resources **A, B, C**, with **2 instances of A, 5 instances of B** and **2 instances of C.** The state of the system is represented as follows:

MAX

	A	B	C
P1	2	2	1
P2	0	3	0
P3	1	2	3
P4	0	1	2

ALLOCATION

	A	B	C
P1	1	2	1
P2	0	1	0
P3	0	0	0
P4	0	1	1

1) Calculate the **AVAILABLE** vector and the **NEED** matrix. **(1 pts)**

- AVAILABLE vector

AVAILABLE=TOTAL INSTANCES−ALLOCATED INSTANCES

	A	B	C
Total instances	2	5	2
Allocated instances	1	4	2
AVAILABLE	1	1	0

- NEED matrix

NEED=MAX−ALLOCATION

	A	B	C
P1	1	0	0
P2	0	2	0
P3	1	2	3
P4	0	0	1

2) Determine whether the system is in a deadlock state using the deadlock detection algorithm. **(1 pts)**

Because we do not have the actual requests we take REQUEST = NEED (all future requests)

WORK

Steps	A	B	C
Step 0	1	1	0
Step 1	2	3	1
Step 2	2	4	1
Step 3	2	5	2

FINISH

P1	P2	P3	P4
F	F	**T**	F
T	F	**T**	F
T	**T**	**T**	F
T	**T**	**T**	**T**

Since for all i, FINISH [i] = true, the system is **not in a deadlock state**.

3) Determine whether the following requests can be satisfied or not using the Banker's algorithm: **(3 pts)**

- **P4** requests **1 instance of C**

Request = (0,0,1)

(0,0,1) ≤ NEED[4] = (0,0,1);

but (0,0,1) > AVAILABLE = (1,1,0), so the request **cannot be satisfied** and **P4 must wait**.

- **P1** requests **1 instance of A**

Request = (1,0,0)

(1,0,0) ≤NEED[1] = (1,0,0)

(1,0,0) ≤ AVAILABLE = (1,1,0)

Assume the request is granted, and update the data:

- AVAILABLE = AVAILABLE – Request
- ALLOCATION[1] = ALLOCATION[1] + Request
- NEED[1] = NEED[1] – Request

NEED

	A	B	C
P1	0	0	0
P2	0	2	0
P3	1	2	3
P4	0	0	1

ALLOCATION

	A	B	C
P1	2	2	1
P2	0	1	0
P3	0	0	0
P4	0	1	1

AVAILBLE

A	B	C
0	**1**	**0**

Next, we apply the **safety algorithm** :

WORK

Steps	A	B	C
Step 0	0	1	0
Step 1	2	3	1
Step 2	2	4	1
Step 3	2	5	2

FINISH

P1	P2	P3	P4
F	F	F	F
T	F	F	F
T	**T**	F	F
T	**T**	F	**T**

- From Step 3, there is **no i such that FINISH [i] = false and NEED[i] <= Work**.
- FINISH [3] = false, so the system is **not in a safe state**. The request **cannot be satisfied**, and **P1 must wait**.
- The **Operating System restores** the previous values of the **ALLOCATION** and **NEED matrices**, and the **AVAILABLE vector**.

4) Give two drawbacks of the Banker's algorithm. **(1 pts)**

- It requires knowing the maximum number of resources required by each process, but this type of information is rarely available in real systems.
- Pessimistic: the algorithm may delay a resource request whenever there is a risk of deadlock, even though in reality the deadlock may never occur.

Exercise 2 (6 points)

We consider the following three concurrent processes. The semaphore **mutex** is initialized to **1**, and the shared variable **x** is initialized to **3**.

Process P1	Process P2	Process P3
(a) P(Mutex);	(d) P(Mutex);	(g) P(Mutex);
(b) x := x + 1;	(e) x := x * 2;	(h) x := x - 4;
(c) V(Mutex);	(f) V(Mutex);	(i) V(Mutex);

1) Consider the following execution sequence: **a d b g c e f h i**. If this scenario is possible, what is the final value of x? justify your answer otherwise **(1 pts)**

Yes, this execution is possible, the final value of x: (Initial value: x = 3)

P1: x := x + 1 → x = 4

P2: x := x * 2 → x = 8

P3: x := x - 4 → x = 4 ➔ final value = 4

2) Answer the same question for this execution sequence: **a d e b c f g h i**. **(1 pts)**

No, this scenario is impossible. P1 executes P(Mutex), so the value of the mutex becomes 0. Then, P2 is blocked when executing P(Mutex) (mutex = -1). Therefore, instruction (**e)** cannot be executed before (c) is completed.

3) Determine all possible final values of **x** with respect to all possible execution sequences, and justify your answer. **(1 pts)**

We need to consider **all possible permutations** of the three operations:

1. **P1 → P2 → P3**: x = 3 → 3 + 1 = 4 → 4 * 2 = 8 → 8 - 4 = 4
2. **P1 → P3 → P2:** x = 3 → 3 + 1 = 4 → 4 - 4 = 0 → 0 * 2 = 0
3. **P2 → P1 → P3**: x = 3 → 3 * 2 = 6 → 6 + 1 = 7 → 7 - 4 = 3
4. **P2 → P3 → P1**: x = 3 → 3 * 2 = 6 → 6 - 4 = 2 → 2 + 1 = 3
5. **P3 → P1 → P2**: x = 3 → 3 - 4 = -1 → -1 + 1 = 0 → 0 * 2 = 0
6. **P3 → P2 → P1**: x = 3 → 3 - 4 = -1 → -1 * 2 = -2 → -2 + 1 = -1

So all possible final values of x are: **–1, 0, 3, 4**

4) Modify the above processes (using the semaphore mechanism), so that the final value of **x** is always guaranteed to be **3**. **(2 pts)**

we need to add one semaphore **S** initialized to **0** to respect the order "P2->P1->P3" (or "P2->P3->P1"), so the three processes are modified as follows:

Mutex: semaphore :=1; **S: semaphore:=0;**		x: integer :=3;
Process P1 **P(S);** (a) P(Mutex); (b) x := x + 1; (c) V(Mutex);	Process P2 (d) P(Mutex); (e) x := x * 2; (f) V(Mutex); **V(S); V(S);**	Process P3 **P(S);** (g) P(Mutex); (h) x := x - 4; (i) V(Mutex);

5) If we use the TAS (Test And Set) instruction instead of the mutex to protect the shared variable x, list two (2) main problems that can occur? **(1 pts)**

- **Busy waiting**: Processes spin in a loop while waiting for the lock, wasting CPU time instead of sleeping.
- **Starvation**: TAS provides no fairness; some processes may repeatedly acquire the lock while others wait indefinitely.
- **Priority inversion**: A low-priority process holding the lock can block a high-priority process, and TAS has no priority-inheritance mechanism to fix this.

Exercise 3 (8 points)

A car park has **N parking spaces**. Incoming cars **(C-entering)** and outgoing cars **(C-exiting)** are considered as processes. Entry to and exit from the car park are carried out through a **single narrow passageway**. To manage the car park, the following conditions must be respected:

1. A car cannot enter the car park, and therefore cannot cross the passageway, if the car park is full.
2. The passageway can accommodate only one car at a time.
3. When cars wish to enter the car park while other cars wish to exit, priority is given to the exiting cars (C-exiting).

Write the algorithms for the processes **C-entering(i)** and **C-exiting(j)**, synchronized using either a semaphore or a monitor mechanism, **of your choice**.

Semaphore solution:

```
Var
   Park : semaphore := N;      // access to the car park (initially N free spaces)
   Passageway : semaphore := 1;      // access to the passageway in mutual exclusion
   SE  : semaphore := 0;      // block entering processes
   countE, countX : integer := 0;   // counters of entering and exiting processes
   Mutex1, Mutex2 : semaphore := 1;   // protect counters
```

Process C-entering(i)	**Process** C-exiting(j)

Begin
P(Park);
P(Mutex1);
countE++;
	Begin
	P(Mutex2);
	countX++;
	V(Mutex2);

```
   V(Mutex1);

   P(Mutex2);
     If (countX > 0) then begin
                                V(Mutex2);
                                 P(SE);
                          end;
   P(Passageway);

   Cross_passageway_to_enter();

   V(Passageway);

   P(Mutex1);
     countE--;
   V(Mutex1);
End;
```

```
   P(Passageway);

   Cross_passageway_to_exit();

   V(Passageway);

    V(Park);

    P(Mutex2);
      countX--;

    P(Mutex1);
    If (countX == 0) and (countE > 0) then V(SE);
    V(Mutex1);
    V(Mutex2);
End;
```

MONITOR:

```
Process C-entering(i)
Begin
   M.enter_CE();
   Cross_passageway_to enter();
   M.exit_CE();
End;
```

```
Process C-exiting(j)
Begin
   M.enter_CX();
   Cross_passageway_to_exit();
   M.exit_CX();
End;
```

```
Monitor M;
Var
   CE, CX : condition;   // conditions for entering and exiting cars
   place : integer;      // number of cars in the parking
   countX  : integer;      // number of cars waiting to exit
   busy  : boolean;      // passageway occupied
```

```
Procedure enter_CE();
Begin
   While (place == N) or (countX > 0) or (busy) then CE.wait;
   place ++ ;
   busy := true;
End;
```

```
Procedure exit_CE();
Begin
   busy := false;
   If (cptX > 0) then CX.signal
   Else  CE.signal;
End;
```

```
Procedure enter_CX();
Begin
    countX++;
   If (busy) then CX.wait;
   busy := true;
End;
```

```
Procedure exit_CX();
Begin
   countX--;
   place - -;
   busy := false;
   If (countX > 0) then CX.signal;
   Else CE.signal;
End;
```

```
Begin
   place := 0;
   cptX := 0;
   busy := false;
End;
```