Solutions for Tutorial Session N°2: Semaphores

Objective: Understand and practice process synchronization using semaphores through exercises on task ordering, resource sharing, and classic concurrency problems.

Exercise 1

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

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**.

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.

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**.

we need to add a semaphores **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 the problems that can occur?

- **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 2

We consider a set of six sequential tasks **{A, B, C, D, E, F}**. Task **A** must precede tasks **B**, **C**, and **D**. Tasks **B** and **C** must precede task **E**. Tasks **D** and **E** must precede task **F**. Implement the synchronization of these tasks using semaphores, blocking and unblocking tasks while respecting the above execution order.

SOLUTION

1- with 5 semaphores:

Var SA,SB,SC,SD,SE : **Semaphore** Initial Value = 0,0,0,0,0 ;		
Task A **begin** Execute; **V(SA) ; V(SA) ; V(SA) ;** **end**;	**Task B** **begin** **P(SA) ;** Execute; **V(SB) ;** **end**;	**Task C** **begin** **P(SA) ;** Execute; **V(SC) ;** **end**;

Task D **begin** **P(SA) ;** Execute; **V(SD) ;** **end** ;	**Task E** **begin** **P(SB) ; P(SC) ;** Execute; **V(SE) ;** **end**;	**Task F** **begin** **P(SE) ; P(SD) ;** Execute; **end**;

2- with 3 semaphores:

Var SA,SBC,SDE : **Semaphore** Initial Value = 0,0,0 ;		
Task A **begin** Execute; **V(SA) ; V(SA) ; V(SA) ;** **end**;	**Task B** **begin** **P(SA) ;** Execute; **V(SBC) ;** **end**;	**Task C** **begin** **P(SA) ;** Execute; **V(SBC) ;** **end**;
Task D **begin** **P(SA) ;** Execute; **V(SDE) ;** **end** ;	**Task E** **begin** **P(SBC) ; P(SBC) ;** Execute; **V(SDE) ;** **end**;	**Task F** **begin** **P(SDE) ; P(SDE) ;** Execute; **end**;

Exercise 3

Consider the parallel execution of the following two processes:

Process A:	**Process B:**

begin
repeat
T1;
until false;
end;
	begin
	repeat
	T2;
	until false;
	end;

1. Use one semaphore to synchronize the two processes so that the execution of task **T1** never occurs simultaneously with task **T2**.
2. Use two semaphores to synchronize the two processes so that the tasks always execute consecutively: **T1 T2 T1 T2 T1 T2 ...**
3. Use two semaphores to synchronize the two processes so that the tasks always execute in the following order: **T1 T2 T2 T1 T2 T2 T1 T2 T2 ...**

SOLUTION

1) Semaphore s := 1 ;

Process A :
begin
repeat
P(s) ;
T1 ;
V(s) ;
until false
end ;
	Process B :
	begin
	repeat
	P(s) ;
	T2 ;
	V(s) ;
	until false
	end ;

2) Semaphore s1 := 1; // Allows process A to start
Semaphore s2 := 0; // Prevents process B from starting until A signals it

Process A :
begin
repeat
P(s1) ;
T1 ;
V(s2) ;
until false
end ;
	Process B :
	begin
	repeat
	P(s2) ;
	T2 ;
	V(s1) ;
	until false
	end ;

3) Semaphore s1 := 2;
Semaphore s2 := 0;

Process A :
begin
repeat
// Wait for two signals from B before proceeding
P(s1) ;
P(s1) ;
T1 ;
// Give B two signals to allow it to proceed twice
V(s2) ;
V(s2) ;
Until false
End ;
	Process B :
	begin
	repeat
	P(s2) ;
	T2 ;
	V(s1) ;
	Until false
	End :

- Or we can use this initialization:

Semaphore s1 :=1 ;
Semaphore s2 := 0;

Process A :
begin
repeat
// Wait for two signals from B before proceeding
P(s1) ;
T1 ;
// Give B two signals to allow it to proceed twice
V(s2) ;
V(s2) ;
P(s1);
Until false
End ;
	Process B :
	begin
	repeat
	P(s2) ;
	T2 ;
	V(s1) ;
	Until false
	End :

Exercise 4 (Barber Shop)

Consider a barber shop with **N chairs** for waiting customers, and a **chair C** for the customer being served, and a **barber B**.

- ✓ If there are no customers, the barber **B** sleeps in chair **C**.
- ✓ When a customer arrives while the barber **B** is sleeping, the customer wakes him up, sits in **C**, and gets a haircut.
- ✓ If a customer arrives while the barber is busy: if one of the **N** chairs is free, the customer sits and waits; otherwise, the customer leaves.
- ✓ When the barber finishes a haircut and there are waiting customers, he serves the next customer.

The task is to synchronize the activities of the barber and his customers using semaphores. Detail the barber's and customers' operations and the role of each semaphore used.

SOLUTION

Barber

Repeat
  If the waiting queue of customers is empty, then the barber sleeps (blocks);
  Perform haircut;
Until end of day;

Customer

- If the number of customers in the shop is equal to N+1, the customer leaves the shop;
- Otherwise, increment the number of customers (with mutual exclusion);
- Add themselves to the waiting queue (may block if barber is busy);
- Get a haircut and leave;

```
Program BarberShop;

Const
   N = … ;  // Maximum size of the waiting queue

Var
   clientCount: Integer;           // Counts the number of customers in the shop
   mutex: Semaphore InitialValue = 1;      // Protects access to clientCount
   waitingClients: Semaphore InitialValue = 0;  // Blocks waiting customers
   barberReady: Semaphore InitialValue = 0;    // Blocks the barber until a customer arrives
```

```
Process Barber;
Begin
   While true Do
   Begin
      // Wait until a customer arrives
      P(barberReady);

      PerformHaircut;   // Cut hair
   End;
End;
```

```
Process Customer-i;
Begin
   P(mutex);  // Enter critical section to update client count
   If clientCount = N + 1 Then      // room full
   Begin
      V(mutex);              // Release mutex
      Exit();                // Customer leaves
   End
   Else
   Begin
      clientCount := clientCount + 1;
      // More than one customer: wait
      If clientCount > 1 Then
      Begin
         V(mutex);           // Release mutex before waiting
         P(waitingClients);     // Block until barber is ready
      End
      Else          // First customer
         V(mutex);

      V(barberReady);          // Wake up the barber
      GetHaircut();           // Customer gets haircut

      P(mutex);               // Enter critical section again
      clientCount := clientCount - 1;

      // Wake up a waiting customer if any
      If clientCount > 0 Then
         V(waitingClients);

      V(mutex);               // Exit critical section
   End;
End;
```

```
Begin
   clientCount := 0;

   ParBegin
      Customer-1; Customer-2; ...; Customer-m;
      Barber;
   ParEnd;
End;
```

Exercise 5 (ATM Alternation)

An ATM can be used by only one person at a time. When there are women and men waiting, a strict alternation must be ensured: after a woman uses the ATM, it must be a man's turn (if a man is waiting), otherwise women continue. The same applies for men. Initially, the turn is given to women.

Provide a synchronization scheme for the processes **WOMAN** and **MAN** using a semaphore mechanism.

SOLUTION

```
Program ATM_System
Var
semaphore mutex := 1;        // protects counters and 'turn'
semaphore womanQueue := 0;   // queue for waiting women
semaphore manQueue := 0;     // queue for waiting men

int waitingWomen := 0;
int waitingMen := 0;
int turn := 0;               // 0 = woman's turn initially
```

```
process WOMAN-i;
begin
   P(mutex);
   waitingWomen := waitingWomen + 1;

   if (turn = 0 AND waitingWomen>1) OR (turn = 1 AND
waitingMen > 0) then
   begin
      V(mutex);
      P(womanQueue);     // wait until man or woman finish
   end
   else
   begin
      turn := 0;          // claim turn for women
      V(mutex);
   end;

   use_ATM();
```

```
process MAN-j;
begin
   P(mutex);
   waitingMen := waitingMen + 1;

   if (turn = 0 AND waitingWomen>0) OR (turn = 1
AND waitingMen > 1) then
   begin
      V(mutex);
      P(manQueue); // wait until man or woman finish

   end
   else
   begin
      turn := 1;          // claim turn for men
      V(mutex);
   end;

   use_ATM();
```

```
   P(mutex);
   waitingWomen := waitingWomen - 1;  // done using ATM

   if (waitingMen > 0) then
   begin
      turn := 1;            // give turn to men
      V(manQueue);
   end


 else if (waitingWomen > 0) then
      V(womanQueue);      // continue with next woman

   V(mutex);
end;
```

```
   P(mutex);
   waitingMen := waitingMen - 1;    // done using ATM

   if (waitingWomen > 0) then
   begin
      turn := 0;            // give turn to women
      V(womanQueue);
   end


 else if (waitingMen > 0) then
      V(manQueue);          // continue with next man

   V(mutex);
end;
```

```
    Begin
       Parbegin
          woman1; woman2; … ; woman_i;
          man1; man2; … ; man_j;
       Parend;
    End.
```

Exercise 6 (Swimming Pool)

An Olympic swimming pool can receive swimmers from three synchronized swimming teams: **T1**, **T2**, and **T3**. To organize training sessions, the rule is: at any given time, the pool can accommodate any number of swimmers but from at most 2 teams. For example, swimmers from teams **T1** and **T3** can train simultaneously, but if a swimmer from team **T2** wants to enter, they must wait until all swimmers from one team have left the pool.

Propose a semaphore-based synchronization scheme for processes **T1**, **T2**, and **T3**, corresponding to swimmers from each team.

SOLUTION

```
Program Olympic_Pool
Common context:
count1, count2, count3 : integer // counters for the number of swimmers in each team
mutex1, mutex2, mutex3 : Semaphore := 1   //ensures mutual exclusion access to variables   count1, count2, count3
training : Semaphore := 2   //ensures that swimmers from at most two teams are training  simultaneously
```

```
Process Team1
begin
 P(mutex1);
 if (count1 = 0) then
    P(training);
 count1++;
 V(mutex1);
  train();
```

```
Process Team2
begin
  P(mutex2);
  if (count2 = 0) then
    P(training);
  count2++;
  V(mutex2);
  train();
```

```
Process Team3
begin
   P(mutex3);
   if (count3 = 0) then
     P(training);
   count3++;
   V(mutex3);
   train();
```

P(mutex1);
count1--;
if (count1 = 0) then
V(training);
V(mutex1);
end;
	P(mutex2);
	count2--;
	if (count2 = 0) then
	V(training);
	V(mutex2);
	end;
		P(mutex3);
		count3--;
		if (count3 = 0) then
		V(training);
		V(mutex3);
		end;

```
begin
  count1 := 0; count2 := 0; count3 := 0;
  parbegin
    Team1-1; … ; Team1-i;
    Team2-1; … ; Team2-j;
    Team3-1; … ; Team3-k;
  parend
end.
```

Exercise 7 (Bridge Load)

A bridge can support a maximum load of **20 tons**. The bridge is crossed by **trucks weighing 20 tons** and **cars weighing 5 tons**. You are asked to manage access to the bridge so that:

- ✓ The maximum bridge load is respected.
- ✓ Priority is given to cars: when a car and a truck request access simultaneously, the car must be allowed first, provided the bridge's maximum capacity is not exceeded.

Propose a synchronization scheme for **truck** and **car** processes using semaphores.

SOLUTION

```
Program BRIDGE;
truck: Semaphore := 1;   // The bridge can support only one truck at a time
car: Semaphore := 4;     // The bridge can support up to 4 cars at a time (20/5)
count: integer;          // To count the number of cars on the bridge
mutex: Semaphore := 1;   // To protect the count variable
```

```
Process Car(i)
Begin
  P(mutex);   // Enter critical section to update car count
  count := count + 1;

  /*  The first car blocks on the 'truck' semaphore if a truck is on the bridge.  Other cars wait on 'mutex' until they can proceed.  */
  If (count = 1) Then
    P(truck);

  /* Release cars blocked behind 'mutex' */
  V(mutex);
```

```
Process Truck(j)
Begin

 /* Request access to the bridge with mutual exclusion */
  P(truck);

  CrossBridge();   // Truck crosses the bridge

  /* Release the bridge */
  V(truck);

End;
```

```
    /* Request access to the bridge */
    P(car);
    CrossBridge();     // Car crosses the bridge

    /* Allow next car to access the bridge */
    V(car);

    P(mutex);
    count := count - 1;

/* If no cars are on the bridge, unblock a waiting truck */
    If (count = 0) Then
      V(truck);

    V(mutex);         // Exit critical section
     End;
```

```
Begin
   count:=0 ;
   Parbegin
        Car-1 ; Car-2 ;……. Car-n ;
        Truck-1; Truck-2;……. Truck-m;
   Parend ;
End.
```

Exercise 8

Cars coming from the north and the south must cross a bridge. On this bridge, only one lane of traffic can be used at a time, and only in one direction. Write an algorithm that allows cars to pass from north to south or from south to north by synchronizing them using semaphores at the bridge. There is no priority between cars coming from the north or the south; therefore, the solution must be fair (no starvation). Cars must cross the bridge in their order of arrival.

SOLUTION

```
program bridge-crossing
var
  northCount : integer := 0;
  southCount : integer := 0;
  mutex1 : semaphore := 1;
  mutex2 : semaphore := 1;
  bridge : semaphore := 1;
  order : semaphore := 1; //insure that cars pass in their arrival order to avoid starvation
```

```
process NorthToSouthCar-i;
begin
   P(order);
   P(mutex1);
   northCount := northCount + 1;
   if northCount = 1 then P(bridge);
   V(mutex1);
    V(order);
   CROSS ();
   P(mutex1);
   northCount := northCount - 1;
   if northCount = 0 then V(bridge);
   V(mutex1);
end;
```

```
process SouthToNorthCar-j;
begin
   P(order);
   P(mutex2);
   southCount := southCount + 1;
   if southCount = 1 then P(bridge);
   V(mutex2);
   V(order);
   CROSS ();
   P(mutex2);
   southCount := southCount - 1;
   if southCount = 0 then V(bridge);
   V(mutex2);
end;
```

```
Begin
      Parbegin
         NorthToSouthCar-1;…..; NorthToSouthCar-i;
         SouthToNorthCar-1 ;…….; SouthToNorthCar-j;
      Parend ;
End.
```