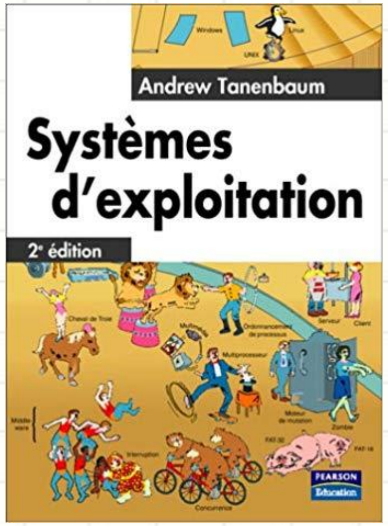
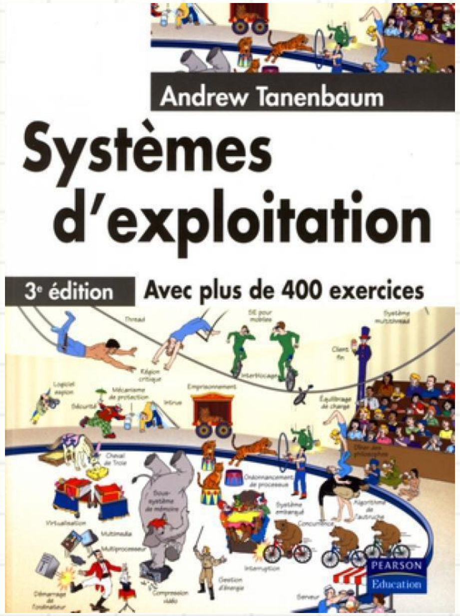
Chapter 3

Process Synchronization Using Semaphores

Reminder

- Don’t forget that:
  - You have to Visit the cours page at:
    - http://moodle.univ-dbkm.dz/course/view.php?id=5142
  - The text book :





Introduction

- The solutions proposed for the mutual exclusion problem cannot be used when dealing with more complex problems. In these problems, synchronization is at issue in its broadest sense. In other words, a process acts on one or more other processes by **blocking** and **unlocking**..
- Synchronization tools aim to control the competition and evolution of processes. They also play a role in achieving cooperation in general.
  - ***Coopération= Communication +synchronization***
- The goal of synchronization tools is to avoid active waiting, "processor monopolization in an empty wait loop".

Semaphore Notion

- **Principle**. The principle is to control synchronization by using an abstract data type called a semaphore..
- **Definition**.
  - A semaphore is an integer variable that, once initialized, can only be used or modified by two atomic operations.
  - These two operations are **P** and **V,** which execute in mutual exclusion.
  - The state of this variable is used to determine whether or not a process can continue its execution.
  - Processes that cannot continue their execution are placed in a queue associated with the semaphore and enter the blocked state.

Semaphore declaration

Semaphore declaration.

```
Type Semaphore = Record
        Valeur: Integer;
        L: List of process; {Process blocked behind the semaphore}
    END;

Var Sem : Semaphore initial value VAL ; {VAL is always an integer indicating the number of processes that can use the semaphore without blocking}
```

SEMAPHORE DECLARATION

The semaphore is manipulated by two primitives P and V

```
Primitive P (Var  Sem : semaphore)
Begin
Sem.valeur:=Sem.valeur-1;
If Sem.valeur < 0 then
                bloquer le processus {le mettre dans la liste associée au sémaphore"Sem.L"}
End ;
Primitive V (Var Sem : semaphore)
Begin
Sem.valeur:=Sem.valeur+1;
If Sem.valeur <= 0 then
        débloquer un  processus de la liste L ;
        {retirer un processus de la liste associée au sémaphore "Sem.L",
        et activer ce processus retiré}
End ;
```

PROPERTIES OF SEMAPHORES

The definition of a semaphore and the **P** and **V** primitives have the following consequences:

- A semaphore cannot be initialized to a negative value, but it can become negative after a certain number of **P** operations.
- A process that invokes the V primitive on a semaphore will wake up one other process blocked behind this semaphore, if its value is less than or equal to 0.
- Invoking the P primitive on a semaphore by a process can have one of the following effects:
  - The process will be blocked and put in the list associated with the semaphore; when the value of the semaphore is less than zero.
  - When the value of the semaphore is greater than or equal to zero; the process continues its execution normally.

PROPERTIES OF SEMAPHORES

- The value of a semaphore denotes:
  - Let the number of processes blocked behind this semaphore **(value <0)**,
  - Let the number of processes that can execute the P primitive without being blocked ***(value>=0).***
- The correct use of semaphores and the P and V primitives can be used to solve a variety of synchronization problems. We will illustrate this by providing several classic examples of semaphore usage.

Order relations between two processes

- **Hypothesis**: Let us consider a process P0 whose execution is dependent on the emission of a signal by process P1.

Process *P0* ;	Process *P1*;
Begin	**Begin**
A1; A2;.....;An;	B1;B2;.....;Bm;
Attendre le signal de ***P1***;	Envoyer le signal **d'activation à** ***P0***;
.............;	.........;
End ;	**End ;**

- Solution: We define a semaphore called signal, initialized to 0.

ORDER RELATIONS BETWEEN TWO PROCESSES

- Var signal: Sémaphore initial value 0;

Process *P0* ;	**Process *P1*;**
<u>Begin</u>	**<u>Begin</u>**
A1; A2;.....;An;	B1;B2;.....;Bm;
P(signal);	**V(signal);**
............;	**.........;**
<u>End ;</u>	**<u>End ;</u>**

In this example, two cases can occur :

- **<u>Case 1</u>**: Process P0 is already blocked on the P(signal) primitive when the signal arrives. When process P1 executes the V(signal) primitive, it wakes up process P0.
- **<u>Case 2:</u>** Process **P0** is active when the signal is emitted **(it is executing instruction Ai)**. It is as if the signal were memorized; in fact, the value of the semaphore signal is set to **1** and when process **P0** executes the **P(signal)** primitive, it will not block.

Mutual exclusion problem for access to a critical section

- **Hypothesis**: Let us consider two processes **P0** and **P1**, competing for entry to a critical section.

- **Solution**: Mutual exclusion can be guaranteed by a semaphore initialized to 1 (often **Mutex** is the symbolic name given to this semaphore).

MUTUAL EXCLUSION PROBLEM FOR ACCESS TO A CRITICAL SECTION

- On

```
Program Exclusionmutuelle;
Var Mutex : Semaphore Initial Value = 1;
```

Process *P0* ;	Process *P1*;

Begin
............;
P(Mutex);
Section critique;
V(mutex);
........;
End ;
	Begin
	............;
	P(Mutex);
	Section critique;
	V(mutex);
	........;
	End ;

```
Begin
    ParBegin
        P0 ; P1
    ParEnd;
End;
```

PRODUCER-CONSUMER PROBLEM

HYPOTHESIS: Consider two categories of processes: **producers** and **consumers**

These are producers that produce objects (any value) and deposit them in a shared memory called. **Buffer**.

Consumer processes use the values deposited in the buffer.

The buffer is of limited size **N**.

SYNCHRONIZATION CONSTRAINTS: (Synchronization scheme)

The operation of these two categories of processes must meet the following constraints:

Producers do not deposit objects when the buffer is full.

Consumers do not consume from the buffer when it is empty.

Only one process can access the buffer at a time.

Objects should not be lost or consumed twice.

PRODUCER-CONSUMER PROBLEM

SOLUTION: Use of three semaphores.

- Plein Full blocks production
- Vide Empty blocks consumption
- Mutex ensures mutual exclusion for access to the buffer.

- Consumption and production are done outside the critical section in order to minimize the time spent in the critical section.

```
Program ProducteursConsommateurs;
Const  N=...;
Type   objet=....;
Var    Tampon : Array [0...N-1] of objet;
         Mutex : Semaphore Initial Value = 1;
         Vide  : Semaphore Initial Value = N;
         Plein : Semaphore Initial Value = 0;

Process Producteur-I;
Var objetproduit:objet;
Begin
  Repeat
        Produire (objetproduit);
        P(Vide)
        P(Mutex)
          Deposer(objetproduit,tampon);
        V(Mutex)
        V(Plein)
  Until Fin= true;
End ;

Process Consommateur-j;
Var objetconsomme: objet;
Begin
Repeat
        P(Plein)
        P(Mutex)
          Retirer(tampon , objetconsomme);
        V(Mutex)
        V(Vide)
consommer(objetconsomme);
 Until Fin= true;

End ;

Begin
   ParBegin
      Producteur-1;Producteur-2; Producteur-3; .......;     Producteur-I;
     Consommateur-1; Consommateur-2; Consommateur-3,......;Consommateur-j;
   ParEnd;
End;
```

READERS-WRITERS PROBLEM

HYPOTHESIS: Consider two categories of processes that access a single shared resource (file, database).

- The first category represents **Readers**: they are only allowed to read the resource.
- The second category, called **Writers**, can read and update the resource.

SYNCHRONIZATION CONSTRAINTS: (Synchronization scheme)

- Avoid simultaneous access of writer processes to the resource.
- Avoid simultaneous access of a writer process with one or more reader processes.
- Reader processes can access the resource simultaneously.

READERS-WRITERS PROBLEM

SOLUTION:

- Readers are not in mutual exclusion.
- *Lectcompteur* counts the number of readers using the resource simultaneously.
- *Lectcompteur* cannot be replaced by the value of a semaphore because the number of reader processes is not limited.

- Use of two semaphores.
- **Redact** which guarantees mutual exclusion between writers.
- **Mutex** semaphore of mutual exclusion between readers that protects the Lectcompteur variable.

READERS-WRITERS PROBLEM

```
Program LecteursRedacteurs;
Var       Lectcompteur:integer;
          Mutex : Semaphore Initial Value = 1;
          Redact   : Semaphore Initial Value = 1;
```

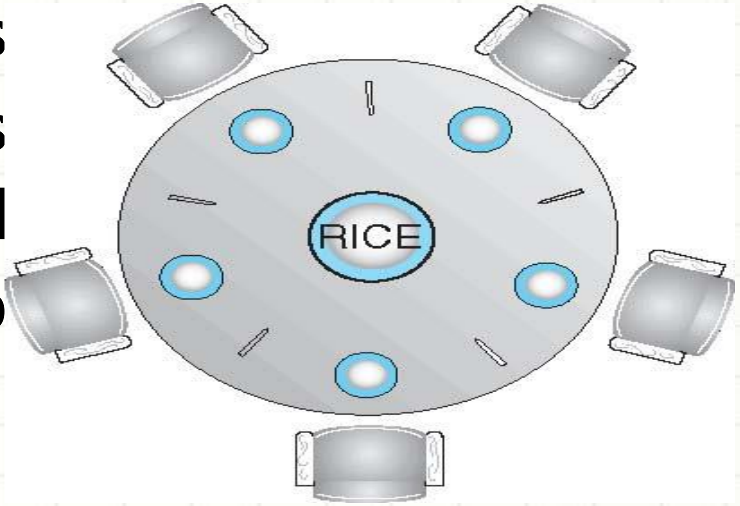
```
Process Lecteur -I;

Begin

P(Mutex)
Lectcompteur.=Lectcompteur+1;
IF  Lectcompteur =1 then  P(Redact);
V(Mutex);

LECTURE;

P(Mutex)
Lectcompteur.=Lectcompteur-1;
IF  Lectcompteur =0 then  V(Redact);
V(Mutex);

End ;
```

```
Process Redacteur-j;

Begin

P(Redact);

ECRITURE;

V(Redact);

End ;
```

```
Begin
     Lectcompteur:=0;
     ParBegin
        Lecteur-1; Lecteur -2; Lecteur -3; .......;      Lecteur -I;
       Redacteur-1; Redacteur-2; Redacteur-3;......; Redacteur-j;
      ParEnd;
End;
```

Five-Dining Philosophers Problem

- **Hypothesis:** Five philosophers, gathered to philos problem to solve at mealtime. Indeed, the meal is which, according to the savoir vivre of these phil two forks. However, the table is only set with o philosophers decide to adopt the following ritual:

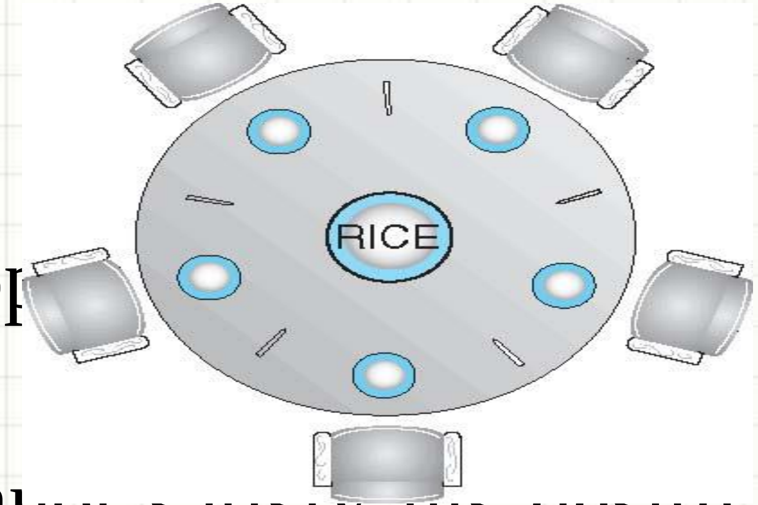


- ƒEach philosopher takes a fixed seat.
- ƒAny philosopher who eats uses the right fork and the left fork.
- ƒTwo neighboring philosophers cannot therefore eat at the same time.
- ƒAt any time, each philosopher is in one of the following three states:

Five-Dining Philosophers Problem

- **Hypothesis:**

- Initially all philosophers think. Any philosop[illegible] not use any fork.
- A philosopher who decides to eat, and cannot satisfy his desire due to lack of forks. In this case, he waits until the two forks (right and left) are available.
- If the two forks are available then the philosopher eats. Any philosopher who eats stops eating after a finite time.



Five-Dining Philosophers Problem

- c

```
Program philosophes;
Var       Etat: Array [0..4] of ( Pense, Afain, Mange);
          Mutex : Semaphore Initial Value = 1;
          Sempriv    : Array [0..4] of Semaphore Initial Value = 0;
           I: integer:
```

```
Process philosophe  ( Var i:integer);

Begin

Repeat

Penser;
Prendrefourchette(i);
Manger
Posefourchette(i);

Until false;

End ;
```

```
Procedure Prendrefourchettes (i: integer);
Begin

P(Mutex);
    Etat[i]:= Afain;
    Test(i);
V(Mutex);

P(sempriv[i]);

End;
```

FIVE-DINING PHILOSOPHERS PROBLEM

- c

```
Procedure Posefourchette(i:integer);
Begin
P(Mutex)

    Etat[i]:= Pense;
    Test(i+4 mod 5);
    Test(i+1 mod 5);

V(Mutex);

End;

Procedure Test( i:integer);
Begin

If (Etat[i]= Afain) and (Etat[i+4 mod 5]<> Mange) and (Etat[i+1 mod 5] <>
Mange) Then
Begin
      Etat [i]:= Mange;
      V(sempriv[i]);
End;

End;
```

```
Begin
For i:=0 to 4 Do Etat[i]:= Penser;
    ParBegin
       Philosophe(0); Philosophe(1);........ Philosophe(4);
    ParEnd;
End;
```