CHAPTER 4

SYNCHRONIZATION OF PROCESSES BY MONITORS

INTRODUCTION

- The risk of programming errors is significant when using semaphores (forgetting to signal(s) or using P instead of V). In addition, synchronization using the P and V operations requires the study of the entire concurrent program to understand the synchronization aspect that it contains.
- To overcome these drawbacks, the concept of a **Monitor** was introduced.

DEFINITION OF A MONITOR

- **PRINCIPLE.** The principle of a monitor is to control synchronization by using a unit that encapsulates the definition of the "critical" resource and the operations that manipulate it.
- **DEFINITION.**
  - A monitor defines a set of variables that keep the state of the resource and a set of procedures that manipulate this resource.
  - A monitor also has an initialization part that initializes the variables before any operation on the resource is invoked.
  - The values of the variables of a monitor are only accessible through the procedures of the monitor itself. These procedures can in turn have parameters and local variables.

Declaration of a Monitor

- This is the general form of a monitor is:
- **Monitor** M;
- **Var** ……..; {declaration of shared variables}
- **Procedure** P1(Parameters);
- **Begin**
- …….
- **End;**

- **Procedure** P2(Parameters);
- **Begin**
- …….
- **End;**
- **………**
- **………**
- **………**
- **………**
- **Procedure** Pn(Parameters);
- **Begin**
- …….
- **End;**
- **Begin**
- **Initialization of shared variables;**
-
- **End;**

Definition of a monitor

- Call of a procedure is done by a classical call :
  - **Call M.P1**(effective parameters);

- The execution of a procedure **P1** is done in *mutual exclusion* with the rest of the procedures of the monitor (including itself). This guarantees the integrity of the permanent variables.

SYNCHRONIZATION

- Process synchronization by monitors is done by using conditions. These are defined as follows:

  - A condition is declared as a variable C.
  - Each condition C has a queue containing the processes blocked behind this condition (figure 1).

SYNCHRONIZATION

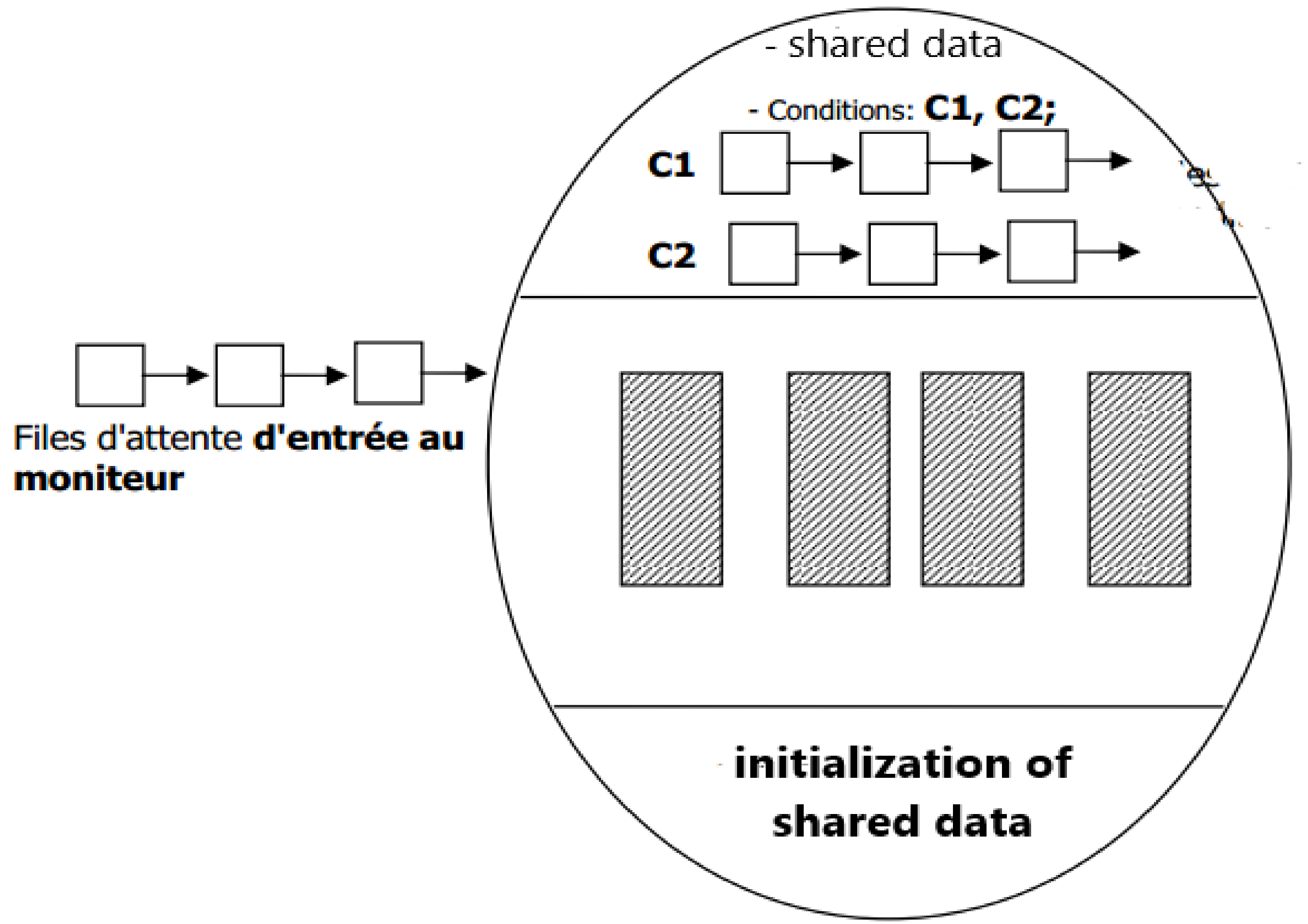

Figure 1. **Structure of a monitor**

Synchronization

- Each monitor can have a variety of conditions that can only be manipulated by two operations: **Wait** and **Signal**.
- **C.wait :** When executed by a process, it blocks the process and places it in a queue associated with the condition C.
- **C.Signal:** When executed by a process, it checks if the queue of C is not empty, in which case, it releases one of the processes waiting. *Note* that if no process is blocked behind the condition, the Signal operation has no effect.

SYNCHRONIZATION

- When a process executes **C.signal**, it will be blocked until the awakened process leaves the monitor.
- Processes blocked by a Condition are prioritized for access to the monitor before a new process can access to execute a monitor procedure.
- Only one process accesses the monitor at a time. Others wait in an input queue (Figure 1).
- The monitor structure ensures that only one process can be **active** in the monitor at a time.

Examples of synchronization using Monitors

Rendezvous point between processes

- **Hypothesis**: Consider N processes that evolve in parallel but when they reach a point in their execution (called the rendezvous point), each process waits for the arrival of all the others at their rendezvous points.
  - The last one to arrive will wake up the others.
  - The awakening is done in cascade: each one wakes up the other by executing *tousarrivés.signal*.

Examples of synchronization using Monitors

Rendezvous point between processes

```
Program gestprocess;
.....
Monitor Rendezvous;
Const N=5;
Var  compteur: integer;
Tousarrives : condition;

Procedure jesuisarrivé;
Begin
        Compteur:=compteur+1;
        If compteur <N then tousarrives.Wait;
        tousarrives.Signal;
end;
begin
compteur:=0;
end;
```

```
Process Pi;
Begin
                ...........
Call Rendezvous.jesuisarrive
            .........;
End ;
......
Begin
Parbegin
      P1,P2,.....,P5
Parend;
End;
```

EXAMPLES OF SYNCHRONIZATION USING MONITORS

Producer/Consumer problem

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

- **Producers** produce objects (any value) and deposit them in a shared memory called: **Buffer**.
- **Consumer** processes use the values deposited in the buffer.
- The buffer has a limited size of **N**.

Examples of synchronization using Monitors

Producer/Consumer problem

Synchronization constraints: (*Synchronization scheme*)

The operation of these two categories of processes must satisfy 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 must not be lost or consumed twice.

- **Solution**:
  - Using a monitor that will manage the shared resource buffer.
  - The monitor contains the procedures ***depoer*** and ***retirer***.
  - The monitor ensures synchronization between producer and consumer.

Examples of synchronization using Monitors

Producer/Consumer problem

```
Program ProducteursConsommateurs;
Const   N=... ;
Type    objet=....;

Monitor Gesttampon;
Const  N=...;

Var      Tampon : Array [0...N-1] of objet;
           nonVide , nonPlein   : condition;
           in,out  : integer
           compteur:0... .N-1;

Procedure déposer(ob:objet);
Begin
        If compteur=N then nonplein.wait;
        Tampon[in]:=ob;
        In:=in+1modN;
        Compteur:=compteur+1;
        nonvide.signal;
End;

Procedure retirer (var ob:objet);
Begin
        If compteur=0 then nonvide.wait;
        ob:= Tampon[out];
        out:=out+1modN;
        Compteur:=compteur-1;
        nonplein.signal;
End;

Begin
        Compteur:=0;
        In:=0;
        Out:=0;
End;
```

EXAMPLES OF SYNCHRONIZATION USING MONITORS

Producer/Consumer problem

```
Process Producteur-I;
Var   objetproduit:objet;
Begin
      Repeat
            Produire (objetproduit);
Call Gesttampon.Deposer(objetproduit);

        Until Fin= true;

End ;
```

```
Process Consommateur-j;
Var  objetconsomme: objet;
Begin
Repeat
Call Gesttampon.Retirer (objetconsomme);

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;
```