CHAPTER III: PARALLELISM MANAGEMENT: SYNCHRONIZATION AND COMMUNICATION TOOLS

PART 1: MUTUAL EXCLUSION

I/ Introduction:

The cooperation of processes to accomplish a common task requires the existence of a mechanism that enables the exchange of information between them, as well as synchronization and mutual exclusion tools to control their execution order.

II/ Mutual Exclusion:

a. Introductory example:

Let's consider two processes (can be heavy or light):

Integer X = 2000	
Process 1 ------ Instructions ------- X := X+1000 ------ Instructions -------	**Process 2** ------ Instructions ------- X := X-2000 ------ Instructions -------

There are several possible cases of execution:

1st case: sequential execution

Integer X = 2000	
Process 1 ------ Instructions ------- X := X+1000 ------ Instructions -------	**Process 2** ------ Instructions ------- X := X-2000 ------ Instructions -------
mov (Register Ax, X) mov (Register Bx, 1000) add (Register Ax, Register Bx) store (X, Register Ax)	
	mov (Register Ax, X) mov (Register Bx, 1000) sub (Register Ax, Register Bx) store (X, Register Ax)

⇨ Running both processes either P1<P2 or P2<P1 gives the same, correct result.

2nd case: Parallel execution (in a time-sharing system)

Integer X = 2000	
Process 1 ------ Instructions ------- X := X+1000 ------ Instructions -------	**Process 2** ------ Instructions ------- X := X-2000 ------ Instructions -------
mov (Register Ax, X) mov (Register Bx, 1000)	
	mov (Register Ax, X) mov (Register Bx, 1000) sub (Register Ax, Register Bx) store (X, Register Ax)
add (Register Ax, Register Bx) store (X, Register Ax)	

⇨ In this case, it is possible to imagine several execution scenarios where different results are obtained at each run → inconsistent results

⇨ The problem of using the same variable X → a critical resource.

b/ Definitions:

1/ Resource: is any object that a process needs to progress in its execution (main memory, CPU, signals, etc.).

2/ Shareable resource: can be allocated at the same time to several processes ◊ can be shared at n access points.

3/ Critical resource: can be shared at a single access point ◊ can only be allocated to one process at a time.

4/ Critical section: a critical section of a process is a sequence of instructions that use a critical resource.

c/ Mutual exclusion definition:

Mutual exclusion is a protocol that protects a critical resource from simultaneous access by several cooperating or competing processes. It allows access to be restricted to one process at a time.

In other words, processes exclude each other from accessing a critical resource.

d/ Mutual exclusion protocol properties:

Each solution used to achieve mutual exclusion must comply with :

- Definition: at any time, no more than one process may be in the critical section.
- Reachability: if several processes are blocked waiting for the critical section and no process is in the critical section, then one of them must reach it in a finite time.
- Progression: a process waiting to reach the critical section in a finite time.
- Same solution: all processes must use the same solution, and no process plays a privileged role.
- A process outside its critical section or entry protocol must not influence the mutual exclusion protocol (on one of the other processes).

- No assumptions should be made about process speeds.

e/ Mutual exclusion solutions:

To write the mutual exclusion between processes, we most often use :

- Active waiting: lock variables, test & set, alternation)
- Passive wait: (semaphores, monitors)
- Hardware solutions (shared memory, interrupt masking)

1/ interrupt masking:

Each process hides interrupts before entering its critical section, so the clock interrupt won't occur and the processor can't be allocated to another process.

Disadvantages:

1/ This approach is not interesting: it's dangerous to allow user processes to mask interrupts → Fear of forgetting to unmask interrupts → would be the end of the system.

2/ If the system has several processors with shared memory, interrupt masking will only take place in the original processor, while the others continue to access the shared memory.

2/ Mutual exclusion by active waiting:

- Lock variables:

Consider a single lock variable that initially has the value 0. The process must test this lock before accessing the critical section:

```
if verrou = 0 than
      Verrou ← 1

      Process enten in his SC
else if verrou = 1 than

      Waits until the process in
      SC returns it 0 after the
      end of its SC
```

integer X = 2000, integer verrou = 0	
Process 1	**Process 2**
------ Instructions ------- if verrou = 0 than Verrou ← 1 else Waits (tests the value of verrou) Verrou ← 1 end if X := X+1000 Verrou ← 0 ------ Instructions -------	------ Instructions ------- if verrou = 0 than Verrou ← 1 else Waits (tests the value of verrou) Verrou ← 1 end if X := X-2000 Verrou ← 0 ------ Instructions -------

Disadvantage: simultaneous lock test by two processes (for example) and lock is set to 0 → both change the value to 1 and enter their SCs.

- Alternation:

Let's consider the integer variable **Tour** taking the values 0 and 1 alternately between two processes:

integer X = 2000, integer Tour = 0	
Process 1 ------ Instructions ------- While (Tour <> 0) do Waits (tests the value ofTour) X := X+1000 Tour ← 1 ------ Instructions -------	Process 2 ------ Instructions ------- While (Tour <> 1) do Waits (tests the value of Tour) X := X-2000 Tour ← 0 ------ Instructions -------

Disadvantage: Not a valid solution if there is a big difference in speed between the processes.

- Test & test instruction:

Most processors have a TAS instruction (elementary hard-wired mechanism) which allows both testing and changing the value of a memory word in an exclusive way. Algorithmically, TAS can be written as follows:

```
Block the access to memory word M
if (M = 0) than
    M ← 1
    Enable access to memory word M
    CO←CO + 2
End if
if (M = 1) than
    Enable access to memory word M
    CO ← CO + 1
End if
End
```

Note: this instruction is executed indivisibly.

integer X = 2000, integer P = 0	
Process 1 ------ Instructions ------- E : TAS(P) Go to E X := X+1000 P ← 0 ------ Instructions -------	**Process 2** ------ Instructions ------- E : TAS(P) Go to E X := X-2000 P ← 0 ------ Instructions -------

3/ Mutual exclusion by passive waiting:

- Semaphores:

A semaphore is an integer variable that is the solution commonly used to restrict access to shared resources. The semaphore was invented by Edsger Dijkstra in 1965.

A semaphore S consists of :

- An intgere **e(s)**.
- A queue **f(s)**.
- Two primitives **P(s)** et **V(s)**
  - P = Proberen (can I)
  - V = Verhogen (go to)

Such that:

```
P(s)
    e(s) ← e(s) – 1
    if (e(s) < 0) than
        stat(p) ← blocked
        enter(p, f(s))
end if
end
```

```
V(s)
    e(s) ← e(s) + 1
    if (e(s) <= 0) than
        go out (f(s), q)
        stat(q, ready)
        enter(q, f(p.ready))
end if
end
```

Use for mutual exclusion:

The semaphore used for mutual exclusion is initialized to 1, and all processes using this semaphore must respect:

```
Process
P(s)
        < SC >
V(s)
end
```

- At any one time, at most one process is in its critical section.
- When no process is in its critical section, SC is entered after a finite time.
- Both primitives run indivisibly.

Example:

intger X = 2000, semaphore s = 1	
Process 1	**Process 2**
------ Instructions -------	------ Instructions -------
P(s)	P(s)
X := X+1000	X := X-2000
V(s)	V(s)
------ Instructions -------	------ Instructions -------

Notes:

1/ The choice of a process in the queue depends on the way queues are managed in the system; the description of **V** does not indicate how a process is chosen.

2/ **P** and **V** are implemented as system primitives.

3/ A semaphore cannot be initialized to a negative value, but can become one after a certain number of executions of **P**.

- Monitors:

Definition:

The monitor is a programmed concept for implementing OS, offering facilities for ensuring mutual exclusion and synchronization between processes.

Monitors are proposed by Hoare and Brinch Hensen.

Features:

- A monitor is a set of:
  - Stat variables
  - Internal procedures
  - External procedures (point of entry)
  - Conditions
  - Synchronization primitives

State variables can only be manipulated by external procedures.

- Only one process can be active in the monitor at any given time.
- Mutual exclusion is ensured at monitor level by the compiler.
- Instructions that manipulate critical resources are stored in the monitor's internal procedures.

To block processes, use conditions with two associated primitives **Wait** and **Signal**.

```
Wait(C)
    stat(p) ← blocked     place the process in the queue associated with condition C
    Enter(p, f(C)
```

```
Signal (C)
    if (f(C) != null) than go
        out(f(C), q)
        stat(q) ←ready /  output the following process from the queue associated with C
        Enter(q, f(ready process))
    end if
Fin
```

Several processes may be waiting for a signal from the scheduler.

example:

```
integer X = 2000,
Monitor Example X-
    libre : condition
    Test : boolean
Procedure ProtegerX()
    if Test than
        Wait(X-libre)
    end if
    Test ← true
Fin
Procedure LibérerX()
    Test ← false
    Signal(X-libre)
end
begin
    Test ← false
end
```

Process 1	**Process 2**
------ Instructions -------	------ Instructions -------
Example.ProtegerX()	Example.ProtegerX()
X := X+1000	X := X-2000
Example.LibererX()	Example.LibererX()
------ Instructions -------	------ Instructions -------

Notes :

1/ Monitors are a programmed concept, the compiler needs to know about them, C and Pascal don't have monitors, they are predefined in a few rare languages like Concurrent Euclid and Java.

2/ Brinch Hensen and Hoare each define a different approach to monitors:

Brinch Hensen	**Hoare**
When process P1 wakes up process P2, it exits the monitor and completes its execution outside the parallel monitor.	When process P1 wakes up process P2, P2 enters the monitor. P1 goes directly to the activator queue and waits until the monitor is free to enter and continue execution.