CHAPTER II: CONCEPTS OF PARALLELISM, COOPERATION AND COMPETITION

I/ Introduction:

Operating systems have gone through several generations since the appearance of computing, and today include systems that embody the concept of parallelism: multiprogramming systems, time-sharing systems, distributed systems, real-time systems, etc.

The aim of parallelism is to have high-performance machines, both in terms of execution time and program complexity.

II/ Definitions:

a. Simultaneity or parallelism:

Simultaneity is the activation of several processes at the same time.

- If the number of processors is greater than or equal to the number of processes, we speak of total simultaneity or real parallelism.
- Otherwise, we speak of pseudo-simultaneity or apparent parallelism.

b. apparent parallelism:

This is the execution of several processes on a single processor, using a time switching function to toggle execution between these processes.

Example: in time-sharing systems, round robin scheduling is used to achieve this parallelism:

- - Conceptually, each process has its own virtual processor and ordinal counter.
- - In practical terms, a single processor and a single ordinal counter.

c. real parallelism:

This is the use of a set of processors (not necessarily identical), with communication between them via dedicated lines.

Example: distributed systems whose aim is to hide the distribution aspect from users.

- - Users access remote resources in the same way as local resources.

III/ Parallelism tools:

a. decomposition into tasks

1/ Definition:

A task is an elementary unit of processing with logical consistency.

The execution of process P consists of the sequential execution of tasks: T_1, T_2, ..., T_n.

$$P = T_1T_2...T_n$$

Each task T_i is associated with its start or initialization date d_i and its end or termination date f_i.

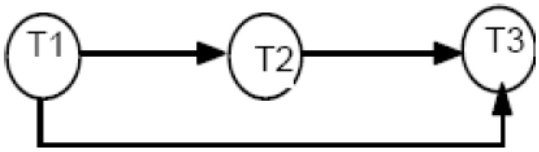
2/ Precedence relationship:

A precedence relation, denoted **<**, on a set **E** is a relation that verify:

- ∀ T ∈ E, we don't have T < T
- ∀ T ∈ E and ∀ T' ∈ E, we don't have simultaneously T < T' and T' < T
- The **<** relation is transitive

The relationship $T_i < T_j$ between tasks means that f_i is less than d_j between dates. If neither $T_i < T_j$, nor $T_j < T_i$, then T_i and T_j are said to be executable in parallel.

A precedence relationship can be represented by a directed graph. For example, the task chain S = ((T_1, T_2, T_3), ($T_i < T_j$ for i less than j)) has the graph :



3/ Bernstein conditions:

Bernstein's conditions determine whether two instructions (or tasks) can be executed in parallel or not. Consider an **inst** instruction;

- **R(inst):** is the set of variables contained in the Inst instruction that are not changed by the execution of this instruction. This set is called the read domain,
- **W(inst)**: the set of variables contained in the **inst** instruction that are modified by execution of the **inst** instruction. This set is called the write domain

We have:

The two instructions **inst1** and **inst2** can be executed in parallel if the following conditions are met:

R(inst1) ∩ W(inst2) = ∅

W(inst1) ∩ R(inst2) = ∅

W(inst1) ∩ W(inst2) = ∅

In this case, we say that the instructions (or, more generally, the tasks) **inst1** and **inst2** are **non-interfering**.

Example:

```
Read(a) ;
Read(b)
C=a + b
Write(c)
```

R(Read(a)) = { }	W(Read(a)) = {a}
•R(Read(b)) = { }	W(Read(b)) = {b}
•R(c=a+b) = {a,b}	W(c=a+b) = {c}
•R(write(c)) = {c}	W(write(c)) = {}

Read(a) and Read(b) can be executed in parallel, but Read(a) and c=a+b cannot because W(Read(a)) ∩ R(c=a+b) = {a}

4/ task parallelism tools :

A/ Fork, Join and Quit primitives (Conway 1963):

- **Fork lab:** This instruction will create a new child process running from the **lab**-labeled instruction. The parent process continues to run normally.
- **Quit**: This instruction terminates the process that executed it.
- **Join n**: This indivisible (atomic) instruction behaves as follows:

n = n -1 ; **if** n ≠ 0 **quit**;

```
n = 2 ;
fork E1 ;
Read(a) ;
Goto E2 ;
E1 : Read(b) ;
E2 : join n ;
c=a+b;
write(c) ;
```

B/ Parbegin and Parend primitives (Dijkstra 1968):

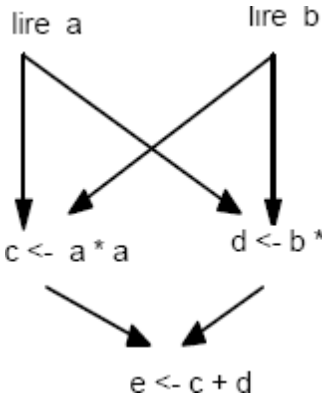
Paralleling is performed using the algorithmic structure :

Parbegin

……

Parend

Example:




```
debut
    parbegin
        lire a
        lire b
    parend
    parbegin
        c ← a * a
        d ← b * b
    parend
    e ← c + d
fin
```

5/ Determinism and maximum parallelism

- In a sequential system, a program that runs always gives the same result.
- On the other hand, in a parallel system, a program running in parallel may give different results depending on the system's behavior. In this case, the system is said to be indeterminate.

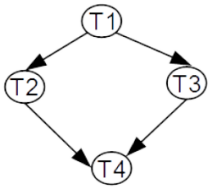
The following program consists of four tasks:

T1 : N:=0;

T2 : N:=N+1;

T3 : N:=N+1;

T4 : Write(N);



- Each task Ti is associated with its start or initialization date di (reading input parameters, acquiring necessary resources, loading information) and its end or termination date fi (writing results, releasing resources, saving information).

The preceding program will give different results depending on its behavior:

Behavior 1: w =d1f1d2f2d3f3d4f4 result = 2

Behavior 2: w' =d1f1d3d2f2f3d4f4 result = 1

Cell memory	d1	f1	d2	f2	d3	f3	d4	f4
C1	0	0	0	1	1	2	2	2

Cell memory	d1	f1	d3	d2	f2	f3	d4	f4
C1	0	0	0	0	1	1	1	1

Definition 1: A task system S = (T; <) is determined if, for all behaviours **w** and **w'** and for any memory cell **Ci**, **V(Ci;w) = V(Ci;w')**

V(Ci,w) represents the succession of values assigned to the cell (variable) **Ci**.

In the example: *V(C1,w) = (0,1,2)* *V(C1,w') = (0,1)*

Definition2: A system **S = (T;<)** has maximal parallelism if it is determinate and any system **S' = (T;<')** is non-determinate,

where **"<'"** is obtained by deleting from the graph of **"< "** a pair which is in the precedence graph of **"<"**.

In another way, the system is determined and if we remove an arc from the precedence graph the system becomes indeterminate.

6/ Multitasking:

Several tasks can be presented and simultaneously active at any given time in the main memory. These tasks are totally independent of each other.

The OS allows CPU time to be shared between several programs that appear to be running simultaneously.

Example:

- Windows Me → cooperative multitasking, since free time of main tasks are used to process the background tasks.
- Windows NT4, 2000, XP, UNIX → consists in installing a scheduler program which allocates processor time to a program according to priority criteria.

7/ Parallel languages:

For example: CSP, Occam, ADA, Java, C, Fortron, …

b. The multithreading:

1/ Definition:

A thread, (lightweight processes) also known as a thread of execution (other names: processing unit, execution unit, instruction thread, lean process), is similar to a process in that both represent the execution of a set of instructions in a processor's machine language. From the user's point of view, these executions appear to run in parallel. However, where each process has its own virtual memory, threads are belonging to the same parent process, share its virtual memory, code section, data section and other system resources. On the other hand, all lightweight processes have their own system stack.

It includes :

- An identifier,
- A program counter,
- A set of registers,
- And a stack

It shares with its creator :

- Global variables,
- File descriptors,
- File management functions.

2/ use:

threads are typically used in conjunction with a program's graphical user interface (GUI). In effect, the user's interactions with the process, via input devices, are handled by a lightweight process, while heavy calculations (in terms of computation time) are handled by one or more other lightweight processes. This software design technique is advantageous in this case, as the user can continue to interact with the program even while it is executing a task. A practical application is found in Word program, where spelling correction is performed while allowing the user to continue entering text.

The use of threads makes an application run more smoothly, as there are no blockages during intense processing phases.

Example: a Word program can have :

- A thread that displays graphics.
- A thread that reads keystrokes.
- A thread that makes spelling corrections.

3/ The advantages of multithreading:

- Allows an application to continue execution even if one of its parts is blocked or performing a long-running operation.
- Resource sharing between threads in the same process.
- Saves memory allocation and execution time → more time to create a process than to create a thread.
- Good method in parallel architectures.

IV/ Issues:

Two processes, User and Client, can be considered logically parallel, but they don't always run simultaneously. When User uses the processor, Client execution is suspended.

This conflict is due to insufficient resources and other interactions such as :

- Client must not be able to access a memory area that User is writing.
- When a memory area contains a value, User must wake up Client if it is inactive.

This example highlights the different types of interaction between processes that cooperate to accomplish a specific task:

- Conflict for simultaneous access to a resource that can only be used by one process at a time.
- Direct action by one process on another → hold or wake-up.

So we can think of an operating system as a set of parallel processes that can interact; and to implement these processes while realizing these interactions, we have two problems to solve:

1/ Write programs describing each individual activity.

2/ Design interaction mechanisms to enable the various means of cooperation and competition between these processes: Mutual Exclusion, Synchronization and Information Communication.