Larbi Ben M'hidi University – Oum El Bouaghi

Faculté of Exact Sciences and Nature and Life Sciences

Department of Mathematics and Computer Science

Modul: Operating Systems II

Lecturer: Dr. Moustafa Sadek KAHIL

2025 – 2026

Chapter I: General Overview on Operating Systems

1 Reminder of the notion of operating system

An operating system (OS) is a set of programs that:

- Manages the computer's hardware resources
- Provides services to applications
- Ensures the interface between hardware and applications It is a layer between hardware and software.

The objective is to make the hardware useful to users.

In reality, the clients of the OS are programs and applications (graphical or commandbased (command-line Shell)). They work with its abstractions.

Figure 0-1 : Role of the operating system in the computer: link between hardware and programs

1.1 Main functions of an OS

- Memory management: Allocation of memory spaces for processes (Paging, Segmentation, Virtual Memory, Swap, Dynamic Allocation)
- Processor management: Scheduling processes by managing CPU allocation through an algorithm
- Process management: Assigning the resources necessary for their execution (Scheduling, Creation/destruction, Synchronization, Interprocess communication)
- Resource management (input/output devices):
    - Access control ○ File access rights through file management systems in read and write ○ Device drivers ○ Buffering ○ Spooling ○ Interrupt management
- File management:
    - Hierarchical organization ○ Access control ○ Disk space allocation ○ Journaling

IPC: Inter-Process Communication: It is interprocess communication that groups a set of mechanisms allowing concurrent processes to communicate.

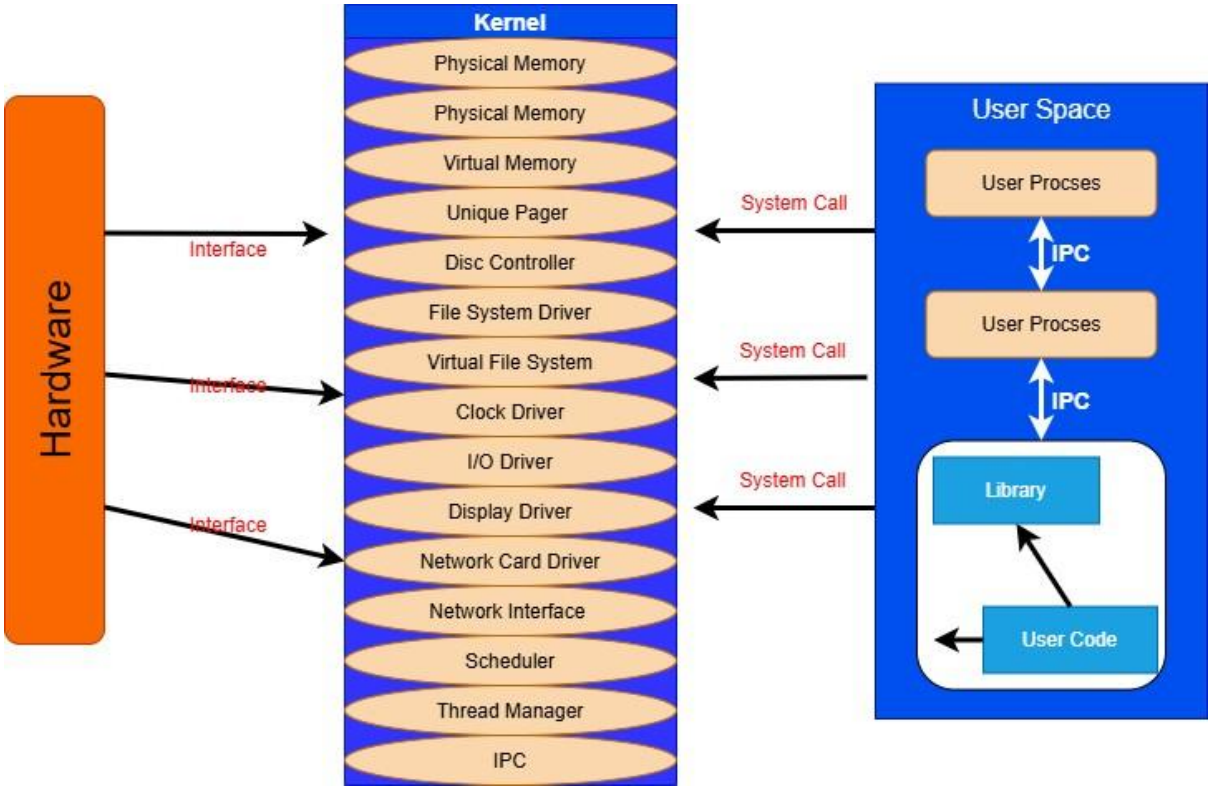


Figure 0-2 : Functional abstraction of the role of the operating system

1.2 Operating System Architecture

Kernel Mode

- Full access to hardware
- Execution of privileged instructions
- Protected memory management

User Mode

- Restricted access to resources
- No privileged instructions • Process isolation

Generations of operating systems

1.3 Générations des systèmes d'exploitation

Table 0-1 : Generations of operating systems

Generation	Description	Characteristics
Single-user / Single-task	The simplest system, allowing only one user and one task at a time.	- Operating systems of early microcomputers. Examples: MS-DOS, early versions of Unix.

Multitasking	Allows sharing processor time among multiple programs, giving the impression of simultaneity.	- Cooperative execution (programs manage the switch) or preemptive (the operating system manages the switch). Examples: Windows, Linux.
Multi-user (timesharing)	Multiple users can simultaneously access the same machine.	- Each user has the impression of being the only one using the computer. Examples: Unix, Linux, mainframe systems.
Multiprocessor	Multiple processors work together to execute several tasks in parallel.	- Improves performance through parallelism. Examples: multiprocessor systems, clusters.
Real-time	Designed for controlling and managing external events with critical timing constraints.	- Guarantee specific response times for urgent signals. Examples: embedded systems, industrial control systems.
Distributed	Allow the execution of a program across multiple machines.	- Distribution of tasks and aggregation of results for complex computations. Examples: distributed computing systems, cloud computing.

2 Basic notions of operating systems

2.1 Program

A program is a set of instructions written in a programming language.

Characteristics:

- Static (stored on disk)
- Passive (does nothing until it is executed)
- Source code + data

2.2 Process

Instance of a program in execution

Characteristics :

- Dynamic
- Active
- Possesses:

- A memory space
- System resources
- An execution context
- A PID (Process Identifier)

Structure of a process

PCB (Process Control Block): it is a data structure that contains all the essential information about a process. As indicated in the table below, this information essentially includes: the state of the process (ready, waiting, running), the CPU time used, the remaining CPU time, the amount of allocated memory, and the priority level.

PCB
PID (Process Identifier)
State (waiting, ready, …)
Contexte processeur
Memory context (pointer)
Registers
Priority
Resource Information
Thread Information
Other information

Figure 0-3: Structure of PCB

Process Hierarchy

- Parent process
- Child process
- Parent-child relationships
- Resource inheritance

States of a process

1. Ready
2. Running
3. Blocked (Blocked/Waiting)
4. Terminated

Process state transitions:

1. New → Ready

- Allocation of resources
- Creation of the PCB
- Loading into memory

2. Ready → Running
   - Selection by the scheduler
   - Context switch
   - Assignment of the processor
3. Running → Blocked
   - Waiting for I/O
   - Waiting for resources
   - Waiting for synchronization
4. Blocked → Ready
   - Availability of resources
   - End of I/O
   - Signal received

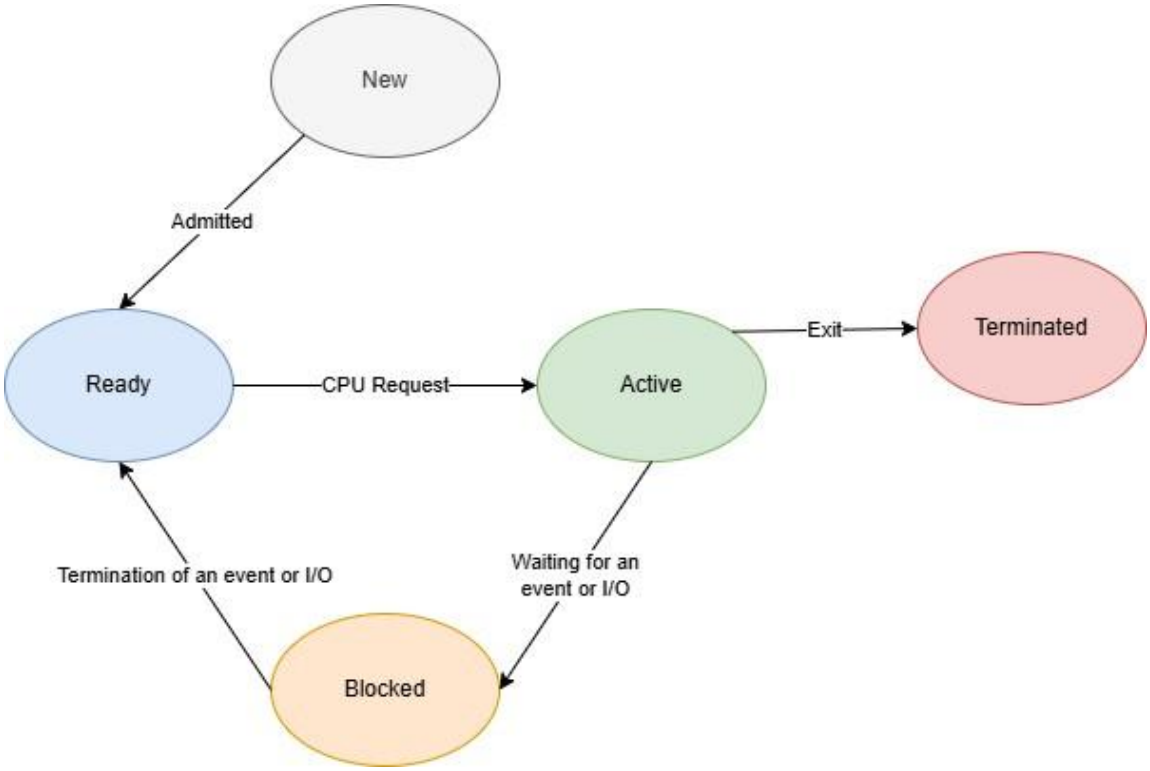


Figure 0-4 : States of a process

Table 0-2 : Description of process states

State	Description	Scenario	Characteristics / Actions	Transitions
New	Initial state of a process immediately after creation	A user launches an application	Data structures initialized, PCB (Process Control Block) created	**New → Ready (Admitted)**: Resource availability check
Ready	Loaded into memory, waiting for CPU allocation	Process resides in the ready queue	Multiple processes ordered by priority or scheduling policy	**Ready → Running (Dispatch)**: Selected by the scheduler for CPU execution
Running	Currently executing on the CPU	Instructions are being processed	Only one process per CPU core; execution limited by time quantum (in preemptive systems)	**Running → Blocked (Wait)**: Awaiting a resource or external event (e.g., I/O)
Blocked	Waiting for a resource or event to occur	Waiting for disk I/O, user input, etc.	Does not consume CPU time; may reside in various wait queues	**Blocked → Ready (Event completion)**: Required resource becomes available
Terminated	Execution has completed	Normal termination,	Resources released, PCB	**Running → Terminated**
		forced kill, or fatal error	deallocated, parent process notified	**(Exit)**: Cleanup and state update

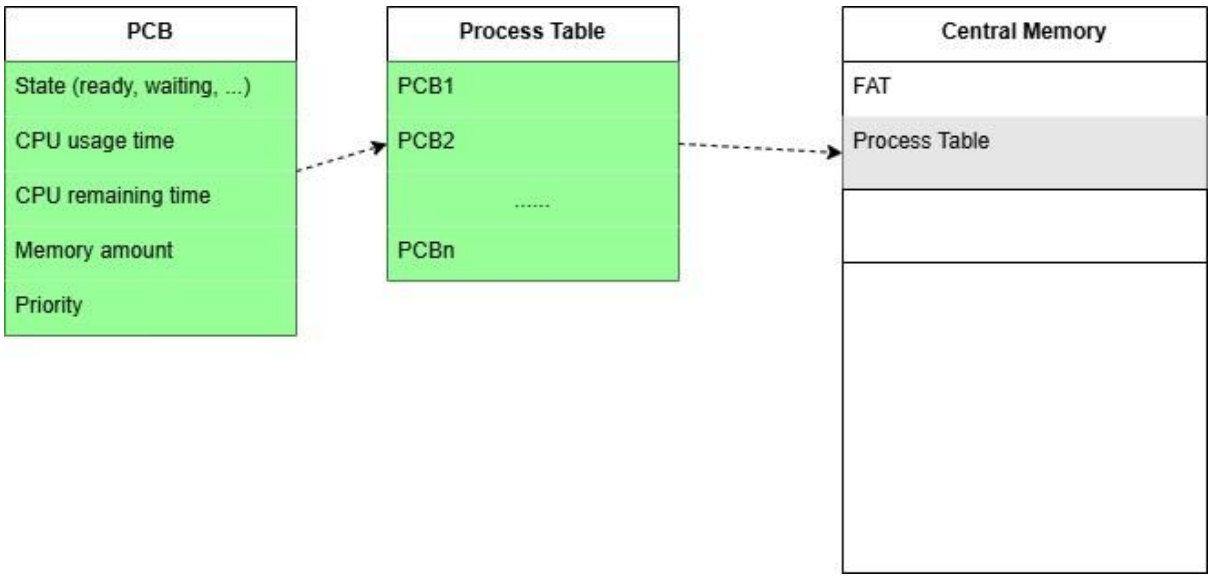


Figure 0-5: Organization of PCBs in system memory

The above diagram represents the organization of Process Control Blocks (PCB) in system memory.

Process Table

The above diagram represents the organization of Process Control Blocks (PCB) in system memory.

2.3 Threads (Lightweight Processes)

A process with a large number of instructions can be broken down into elementary units according to a coherent logical order. These units are called “threads”.

A thread is an execution unit (a lightweight process) within a process that shares its resources. Its lifetime cannot exceed that of the process that creates it. It is characterized by:

- Its creation being faster than a process.
- A lighter context switch.
- Easier communication between threads of the same process.

There is always at least one thread: the main thread.

Characteristics

A thread possesses:

- Its own program counter
- Its execution stack
- Its registers It shares:
- The memory space

- The resources of the parent process Therefore, thread execution is concurrent.

Multi-threading

Multithreading is a programming technique allowing the simultaneous execution of several threads within the same program. As illustrated in the figure below, these threads share the process resources but execute independently, hence the notion of parallelism.

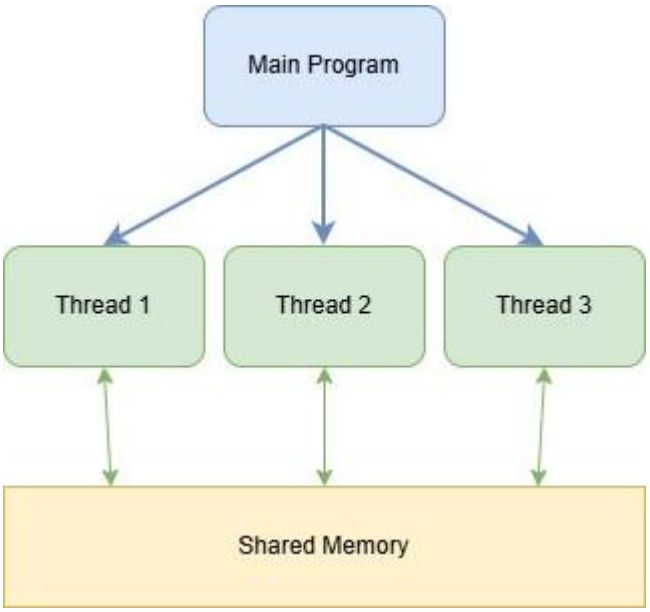


Figure 0-6 : Thread mechanism

Le parallélisme est soumis à une condition qui détermine si des segments d'un programme peuvent s'exécuter en parallèle. Cette condition est nommée : la condition de Bernstein.

Parallelism is subject to a condition that determines whether segments of a program can execute in parallel. This condition is called: Bernstein's condition.

For a code segment of a program, Bernstein's condition is characterized by two sets:

$R(Si)$ and $W(Si)$.

- $R(Si)$: set of variables read by Si
- $W(Si)$: set of variables written by Si

Two segments $S1$ et $S2$ are parallelizable if:

- $W(S1) \cap W(S2) = \emptyset$
- $R(S1) \cap W(S2) = \emptyset$
- $W(S1) \cap R(S2) = \emptyset$

The set of tasks constituting a process can be represented by precedence relations in a graph called: precedence graph.

A precedence graph is a representation tool that shows dependencies and execution order between different tasks or instructions in a program. It is a directed acyclic graph (DAG) where:

- The nodes represent the set of E tasks/instructions.
- The directed arcs (→) represent dependency (precedence) relations.
- If an arc goes from A to B (A → B), then A must be executed before B.
- $\forall A \in E$, the relation $A < A$ is never verified.
- $\forall (A, B) \in E$, one cannot have simultaneously $A < B$ and $B < A$.
- $\forall (A, B, C) \in E$, if $A < B$ and $B < C$ then $A < C$. **Example:**

Consider the following program:

$Instruction\ 1: x = a + b$
$nstruction\ 2: y = c \times d$
$Instruction\ 3: z = x \times y$

If we apply Bernstein's condition to this example:

- We define the sets of variables read ($R(Si)$) and written ($W(Si)$) by the three instructions:
  - $R(Instruction\ 1) = \{a, b\} \circ W(Instruction\ 1) = \{x\} \circ R(Instruction\ 2) = \{c, d\} \circ W(Instruction\ 2) = \{y\} \circ R(Instruction\ 3) = \{x, y\} \circ W(Instruction\ 3) = \{z\}$
- We study the parallelizability of each pair of instructions:
  - Between Instruction 1 and instruction 2 :
    - $W(Instruction1) \cap W(Instruction2) = \emptyset$
    - $R(Instruction1) \cap W(Instruction2) = \emptyset$
    - $W(Instruction1) \cap R(Instruction2) = \emptyset$

      ❾ $Instruction\ 1$ and $Instruction\ 2$ are parallélizable. ○ Between Instruction 1 and instruction 3 :

      - $W(Instruction1) \cap W(Instruction3) = \emptyset$
      - $R(Instruction1) \cap W(Instruction3) = \emptyset$
      - $W(Instruction1) \cap R(Instruction2) = \{x\}$

      ❾ $Instruction\ 1$ and $Instruction\ 3$ are not parallelizable. Therefore,

$Instruction\ 1$ precedes $Instruction\ 3$. We note: $Instruction\ 1 <$

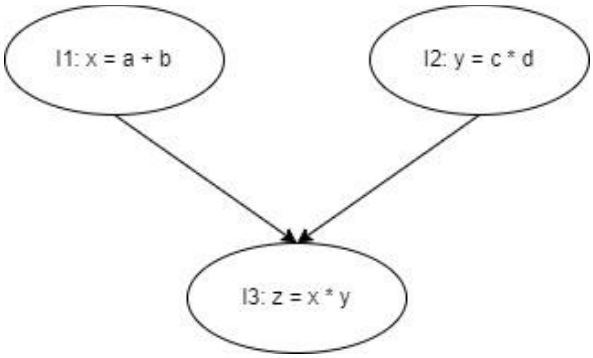
$Instruction\ 3$

- Between Instruction 2 and instruction 3 :
  - $W(Instruction2) \cap W(Instruction3) = \emptyset$
  - $R(Instruction2) \cap W(Instruction3) = \emptyset$
  - $W(Instruction2) \cap R(Instruction2) = \{y\}$

❾ $Instruction\ 2$ and $Instruction\ 3$ are not parallelizable. Therefore,

$Instruction\ 2 < Instruction\ 3$

The graph corresponding to this program is shown in the figure below:



We can observe that $I1$ and $I2$ must execute before (since $I2$ depends on and $I1$). The program therefore allows two segments (instructions) to execute in parallel. Thus, it can have the following form:

```
ParBegin
x = a + b
y = c × d
ParEnd
z = x × y
```

2.4 Shared Resources

A resource is any hardware or software element necessary for the execution of a process.

It may be physical (CPU, memory, devices) or logical (files, semaphores).

Main characteristics of resources

- State: free or occupied
- Access points: single or multiple
- Preemptible or non-preemptible

- Duration of use: short or long
- Exclusive or shareable

Types of resources

Ressources réutilisables :

Can be taken back and reused

- CPU
- Memory
- Devices

b) Consumable resources: Destroyed

after use

- Messages
- Signals

Degree of shareability

- **Exclusive:** one process at a time (printer)
- **Shareable for reading:** several simultaneous reads (files)
- **Fully shareable:** several simultaneous accesses (memory)

The table below summarizes the concept of shared resources with examples. Le tableau ci-dessous récapitule le concept des ressources partagées avec des exemples.

Table 0-3 : Types of shared resources with examples

Criterion	Characteristics/Types	Examples
State	Free Occupied (Busy)	CPU free/in use File open/closed
Concurrent access	Single Multiple	Printer (single access) Memory (multiple simultaneous accesses)
Preemptibility	Preemptible Non-preemptible	CPU (preemptible) Printer (non-preemptible)
Reusability	Reusable Consumable	Reusable: CPU, memory Consumable: messages, signals

Shareability	Exclusive Read-shareable Fully shareable	Exclusive: printer Read-shareable: files Fully shareable: memory
Holding time	Short Long	Short: CPU (used briefly) Long: open files
Allocation mode	Immediate Deferred	Immediate: memory allocation Deferred: waiting for an occupied resource to become available

Allocation Algorithm

```
Request_Resource(P, R) {
if (R is free) {
allocate R to P
mark R as occupied
return SUCCESS}   else
{      put P on hold
block P       return
WAIT}
}

Release_Resource(P, R) {
release R  if (process waiting
for R) {      select a waiting
process P'       unblock P'
allocate R to P'}
}
```

3 Parallelism and task systems

3.1 Task systems

A task system, denoted $S = (E, <)$, is a formal model used to represent a set of tasks to be executed, as well as the precedence constraints between them.

- E : set of tasks. Each task T_i is characterized by a start (d_i) and an end (f_i).
- $<$: precedence relations (constraints) that define dependencies between tasks as well as their execution order

For all tasks T_i and T_{i+1}, if $T_i < T_{i+1}$ then $f_i < d_{i+1}$

3.2 Language of the task system

It is the set of words that satisfy the constraints of the precedence graph. It serves to clearly and concisely represent the precedence relations ($<$) between tasks. The behavior of the task system is thus defined.

If we take again the example of the precedence graph shown below, it can be described by one of the two following task sequences:

- $T1T2T3$
- $T2T1T3$

These sequences constitute the language of this task system.

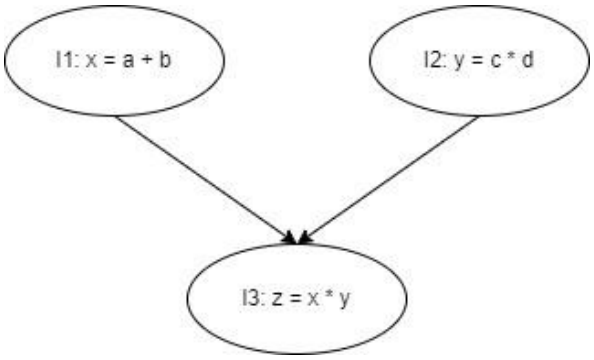


Figure 0-7 : Example of a task system

And since each task T_i is characterized by a start b_i and an end e_i, we can define the following sequences:

- $b_1e_1b_2e_2b_3e_3$
- $b_2e_2b_1e_1b_3e_3$
- $b_1b_2e_2e_1b_3e_3$

- $b_2 b_1 e_2 e_1 b_3 e_3$
- . ..

3.3 Determinism of a task system

Consider the example of system S expressed by the precedence graph shown in the figure below :

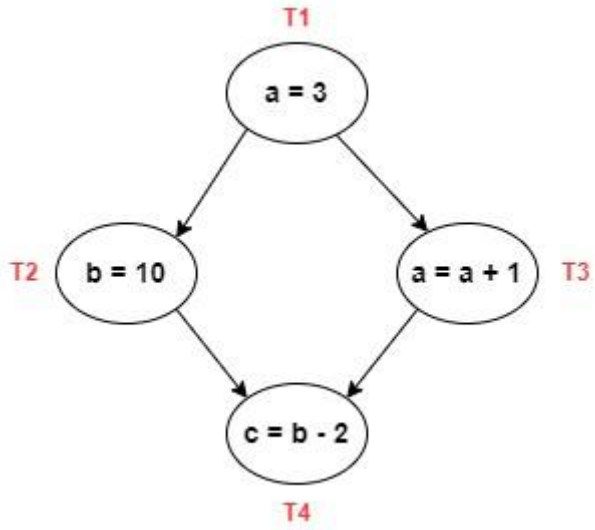


Figure 0-8 : Example of a system expressed by a precedence graph

We notice that tasks T2 and T3 could be executed in parallel. We distinguish two scenarios: $T1, T2, T3, T4$ ($b1e1b2e2b3e3b4e4$) et $T1, T3, T2, T4$ $b1e1b3e3b2e2b4e4$.

Suppose that all variables are initialized to 0:

- If we execute the tasks according to the first scenario, we obtain the following sequence of values:

	S0	T1	T2	T3	T4
a	0	3	3	4	4
b	0	0	10	10	10
c	0	0	0	0	8

  - For a : (0, 3, 3, 4, 4)
  - For b : (0, 0, 10, 10, 10)
  - For c : (0, 0, 0, 0, 8)

- If we execute the tasks in the following order (T1, T3, T2, T4), we obtain the following sequence of values:

	S0	T1	T2	T3	T4

A	0	3	4	4	4
B	0	0	10	10	10
C	0	0	0	0	8

- For a : (0, 3, 4, 4, 4)
- For b : (0, 0, 10, 10, 10)
- For c : (0, 0, 0, 0, 8)

We observe that the sequence of values for each variable is the same for the two possible scenarios, i.e., the behavior of S does not change in both cases and both executions lead to the same result. We can deduce that S is determined.

Let's take another example of a task system S expressed by the precedence graph shown in the figure below.

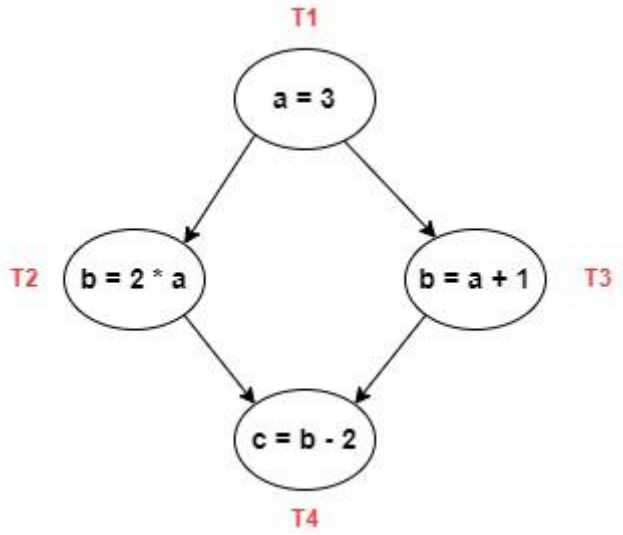


Figure 0-9 : Example 2 of a system expressed by a precedence graph

As in the first example, we distinguish two scenarios $T1, T2, T3, T4$ and $T1, T3, T2, T4$.

- If we execute the tasks according to the first scenario, we obtain the following sequence of values:

	S0	T1	T2	T3	T4
a	0	3	3	3	3
b	0	0	6	4	4
c	0	0	0	0	2

- For a : (0, 3, 3, 3, 3)

- For b : (0, 0, 6, 4, 4)
- For c : (0, 0, 0, 0, 2)

- If we execute the tasks according to the second scenario, we obtain the following sequence of values:

	S0	T1	T2	T3	T4
a	0	3	3	3	3
b	0	0	4	6	6
c	0	0	0	0	4

- For a : (0, 3, 3, 3, 3)
- For b : (0, 0, 4, 6, 6)
- For c : (0, 0, 0, 0, 4)

We observe that the sequence of values for variable a is the same for the two possible scenarios, but that of b and c is different, i.e., the behavior of S changes in the two cases. Therefore, S does not have a unique behavior. We can deduce that S is non-determined.

Non-interfering tasks:

Let $Ti, Tj \in (E, >)$. We say that Ti are Tj are non-interfering if and only if $Ti < Tj \; xor \; Tj < Ti$, in other words: $Ri \cap Wj = Wi \cap Rj = Wi \cap Wj = \emptyset$. They must then satisfy Bernstein's. Thus, in the previous example, it is sufficient to apply Bernstein's rule to the two tasks to verify whether they are non-interfering:

- $R2 = \{a\}, W2 = \{b\}$
- $R3 = \{a\}, W3 = \{b\}$

We see that $W2 \cap W3 \neq \emptyset$, therefore $T1$ and $T2$ interfering and the system is not determined.

Détermination :

$S = (E, <)$ is determined if and only if all variables (memory cells) manipulated by S receive the same sequences of values for each behavior (execution) $\omega \in L(S)$.

In a system $S = (E, >)$, if every two tasks Ti and Tj are non-interfering then the system S is determined.

3.4 Maximal parallelism

A system S is said to have maximal parallelism if:

a. S is determined and

b. Its precedence graph verifies: removing any arc (Ti, Tj) causes interference between tasks Ti and Tj. (Ti, Tj).

Theorem:

For a determined system $S = (E, <)$, there exists a unique system $S' = (E, <')$ equivalent to it such that $<'$ is the transitive closure of the relation:

$$R = \{(T, T') | T < T' \; et (R(T) \cap W(T') \neq \emptyset \; ou \; W(T) \cap R(T') \neq \emptyset \; ou \; W(T) \cap W(T') \neq \emptyset) \; et \; W(T) \neq \emptyset \; et \; W(T') \neq \emptyset\}$$

This relation guarantees maintaining all transitive relations and eliminating redundant relations.

Example:

Consider the task system represented by the precedence graph below, we verify whether has maximal parallelism or not.

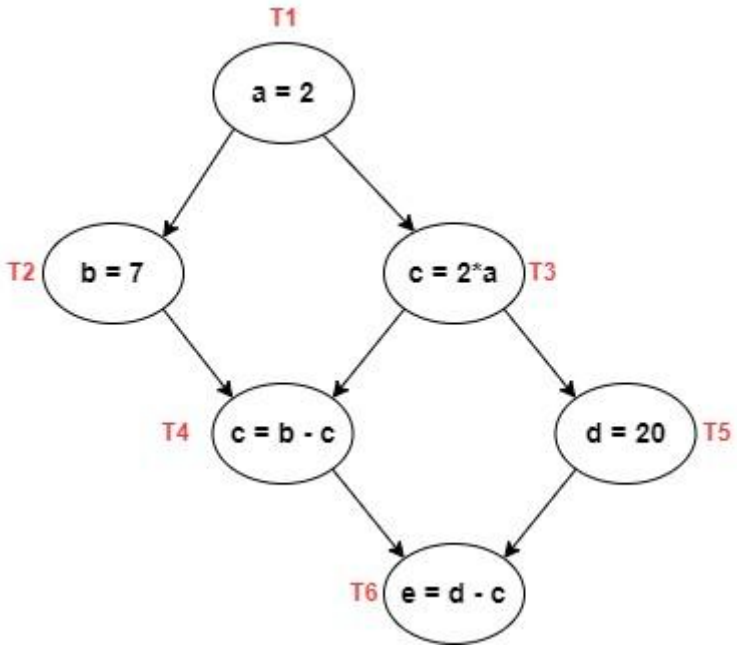


Figure 0-10 : Example of a task system

From this graph, we have:

- $R1 = \emptyset, W1 = \{a\}$
- $R2 = \emptyset, W2 = \{b\}$
- $R3 = \{a\}, W3 = \{c\}$
- $R4 = \{b, c\}, W4 = \{c\}$

- $R5 = \emptyset, W5 = \{d\}$
- $R6 = \{d, c\}, W6 = \{e\}$ **a-** We verify whether S is determined: According to the precedence graph, the pairs of

parallel tasks are $(T2\ et\ T3)$ and $(T4\ et\ T5)$.

We verify whether non-interference is satisfied for the two pairs of tasks:

1. $R2 \cap W3 = W2 \cap R3 = W2 \cap W3 = \emptyset$
2. $R4 \cap W5 = W4 \cap R5 = W4 \cap W5 = \emptyset$

❾ Therefore S is determined.

b- We verify whether S has maximal parallelism:

If we verify non-interference between entre $T1$ and $T2$:

$R1 \cap W2 = W1 \cap R2 = W1 \cap W2 = \emptyset$, this means that arc $(T1, T2)$ is not necessary.

❾ S does not have maximal parallelism.

If we want to find S', we must eliminate redundant arcs and maintain only transitive relations:

5. Similarly to arc $(T1, T2)$, by applying Bernstein's rule, we deduce that arc $(T3, T5)$ is also eliminated.
6. The other arcs are maintained.

The precedence graph representing then becomes as shown below.

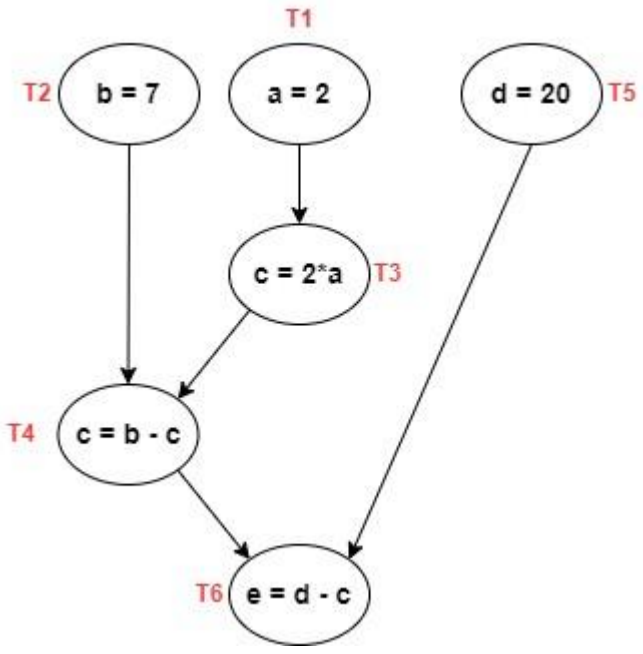


Figure 0-11 : System with maximal parallelism