Mohamed Boudiaf
University
-M'sila-

جامعة محمد بوضياف - المسيلة
Université Mohamed Boudiaf - M'sila

Faculty of Mathematics
and Computer Science

Computer Science
Department

OPERATING SYSTEMS II

CHAPTER II
PROCESS SYNCHRONIZATION

EVENTS AND LOCKS

3rd Year, Computer Science – SI

Dr. Hanene CHERAIT

2025-2026

PROCESS SYNCHRONIZATION TECHNIQUES

SYNCHRONIZATION

- The solutions proposed for the mutual exclusion problem cannot be used when it comes to more complex problems.
- In these problems, the focus is on **synchronization** in its broadest sense.
- A process acts on one or more other processes through **blocking and unblocking**.
- **Synchronization tools** aim to control competition and the evolution of processes.
- They are also involved in implementing **cooperation** in general.
- **Cooperation = Communication + Synchronization**

SYNCHRONISATION

- Events
- Locks
- Semaphores
- Monitors
- Critical Sections
- Path Expressions

SYNCHRONISATION

- Events
- Locks
- Semaphores
- Monitors
- Critical Sections
- Path Expressions

EVENTS

- Processes that are **blocked** after requesting access to their critical sections to use a shared resource **R** wait for the occurrence of an **event**, which in this case is the **release of the resource R.**

- An event is represented by a **Boolean variable**, initialized to **false**:
  - **False:** R is busy (occupied)
  - **True:** R is free (available)

EVENTS

- Two primitives are used by processes to handle the event:
  - **wait(event):** the calling process is blocked only when the ***event*** has not been signaled.
  - **signal(event):** wakes up the blocked process(es) and sets the variable ***event*** to *true*.

event R_free = false **// Resource is busy**

Process A:	**Process B:**
wait(R_free)	**wait(R_free)**
use_resource()	**use_resource()**
signal(R_free)	**signal(R_free)**

Only one process at a time accesses the resource; others wait for the **signal** indicating it's free again.

TYPES OF EVENTS

- **Manual-reset event**
  - Once signaled, it stays *true* until it's manually reset to *false*.
  - Multiple waiting processes can be released.
    - *Used when several processes must respond to the same signal.*
- **Auto-reset event**
  - Automatically returns to *false* after releasing **one** waiting process.
    - Useful when only one process should proceed after a signal.

ADVANTAGES OF EVENTS

- **Avoids Active Waiting (CPU-Efficient)**
  - processes can wait passively for an event to occur instead of continuously checking a condition (busy waiting).
  - This saves CPU time and power, improving system efficiency.
- **Asynchronous Communication**
  - Allow processes to communicate asynchronously — one thread signals when a condition is met, while others respond when notified.
  - This decouples their execution and improves responsiveness.
- **Suitable for I/O Operations**
  - Ideal for situations where a process must wait for I/O completion, device input, or network responses without blocking the entire system.
- **Efficient Task Coordination**
  - Events can be used to notify when tasks finish, resources become available, or new data is ready
    - enabling smooth coordination between processes.
- **Better Resource Utilization**
  - Since processes sleep while waiting for events, system resources are used only when necessary.

DISADVANTAGES OF EVENTS

- **Possible Missed Signals**
  - If a signal is sent before another thread starts waiting, the event may be lost,
    - if a thread starts waiting *after* the event was signaled, it won't know that the event already occurred.
    - causing the waiting thread to stay blocked forever.
- **Timing Sensitivity**
  - Correct behavior often depends on precise timing (who waits first, who signals first)
    - making programs **non-deterministic** and hard to debug.
- **Complex Dependency Management**
  - In systems with multiple interdependent events, coordinating the order and logic between them becomes difficult and error-prone.

DISADVANTAGES OF EVENTS

- **Limited Functionality**
  - Unlike other synchronization mechanisms, events can't handle counting, ownership, or resource tracking directly.
- **Difficult Error Handling**
  - If an event is missed or triggered incorrectly, there's no built-in mechanism to recover or detect that failure.
- **Potential Deadlocks or Starvation**
  - Poor event handling logic can cause processes to wait indefinitely for signals that never arrive.
- **Debugging Challenges**
  - Because event-triggered execution is asynchronous, tracing program flow and identifying timing bugs can be very difficult.

SYNCHRONISATION

- Events
- **Locks**
- Semaphores
- Monitors
- Critical Sections
- Path Expressions

LOCKS

- This lock-based synchronization solution is designed to avoid **busy waiting** at the processor level.
- It assigns to the **critical resource (CR)** a waiting queue **"Q(CR)"**, which is used to store the processes that are unable to access their **critical sections (CS).**
- A Boolean variable called "**lock**" is also used to manage process access to critical resources.

LOCKS

Common context: lock: boolean; // initialized to false

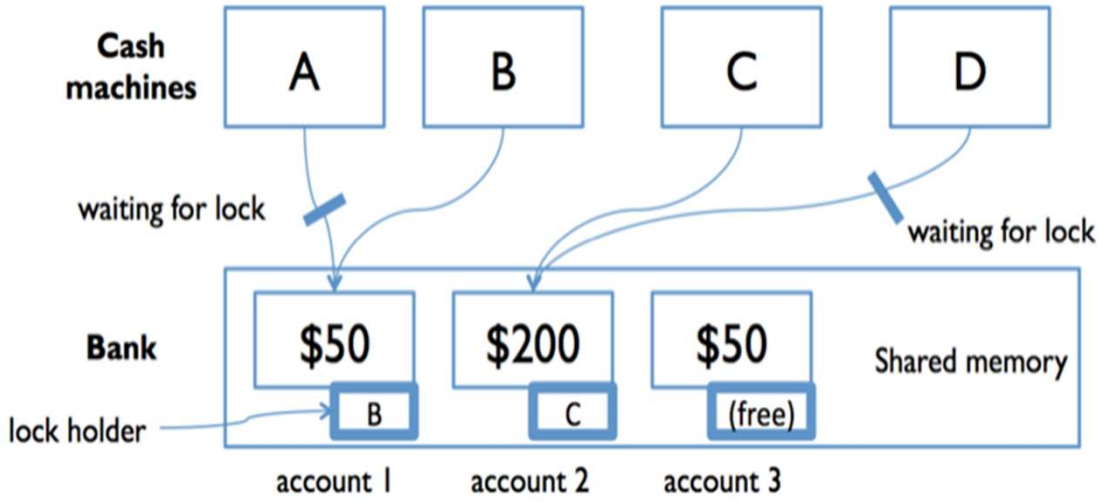
```
Entry primitive (lock);

Begin
    If (not lock) then
        lock := true;
    Else
        enqueue(Q(CR)); /* block and add the process to the waiting queue */
End;
```

```
Exit primitive (lock);

Begin
    If (not empty(Q(CR))) then
        remove(Pj);       /* remove a process from the waiting queue */
    Else
        lock := false;
End;
```

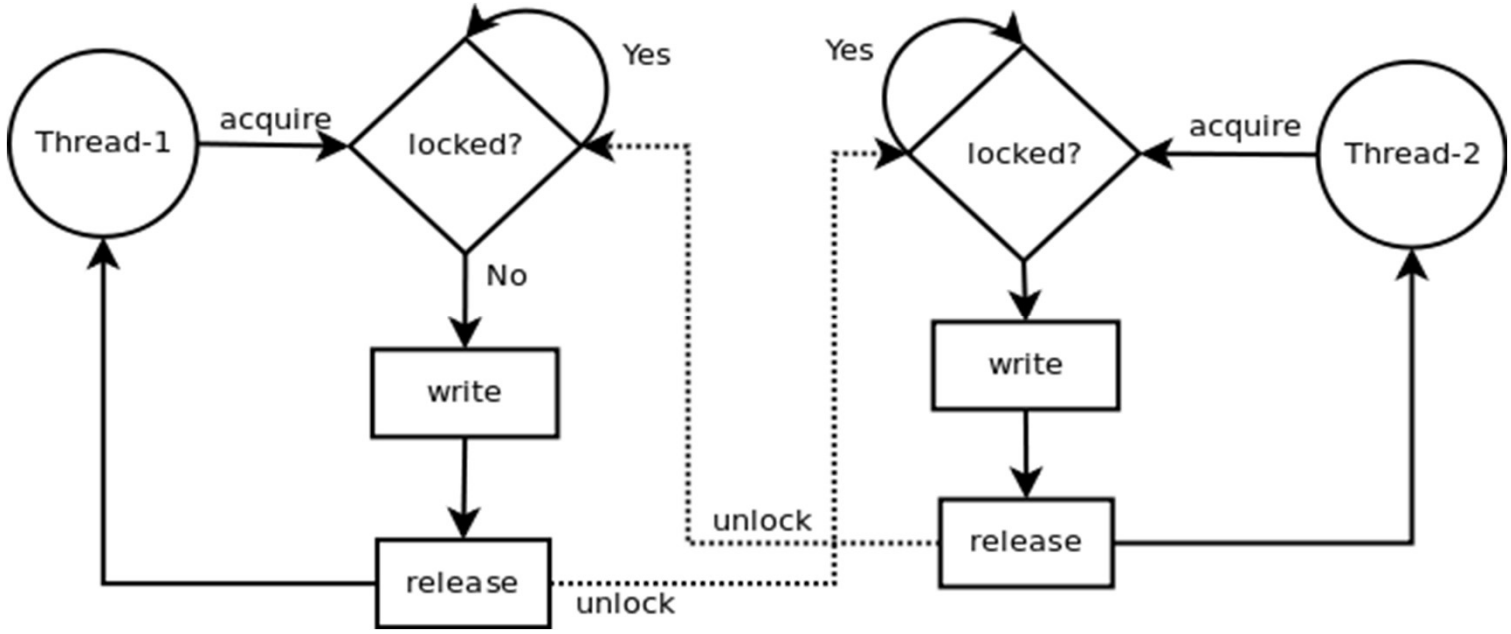
LOCKS

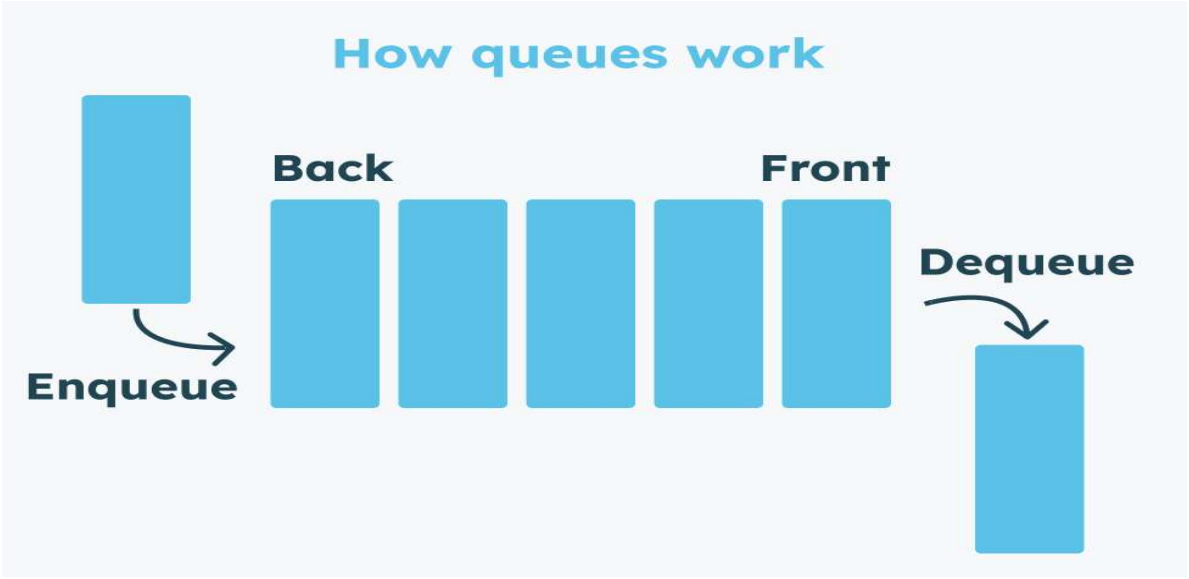



```
Process Pi;
Begin
    Repeat
        ::::
        Entry(lock);   // entry protocol
        CRITICAL SECTION;
        Exit(lock);    // exit protocol
        ::::
    Until false;
End;
```

On many machines, locks are handled directly by two **indivisible primitives**, **"Lock(lock)"** and **"unLock(lock)"**, which respectively simulate the behavior of the two primitives **"Entry(lock)"** and **"Exit(lock)"**.

LOCKS


Thread-1
acquire
locked?
Yes
No
write
release
unlock
unlock
Yes
locked?
acquire
Thread-2
write
release


How queues work
Back
Front
Dequeue
Enqueue

ADVANTAGES OF LOCKS

- **Avoid busy waiting**
  - Processes that cannot acquire the lock are blocked and removed from the CPU, saving CPU cycles.
- **Efficient CPU utilization**
  - Other processes can continue executing while blocked processes wait for the lock.
- **Mutual exclusion**
  - Guarantees that only one process enters the critical section at a time.
- **Fairness possible**
  - With a queue (like Q(R)), processes can be served in FIFO order, reducing starvation.
- **Deterministic behavior**
  - Provides predictable access to shared resources when correctly implemented.

DISADVANTAGES OF LOCKS

- **Resource underutilization**
  - Simple locks do not handle multiple copies of a resource efficiently — some resources may remain idle while processes wait.
    - Ex: A database server might allow **N simultaneous connections.**
- **Deadlock risk**
  - If multiple locks are acquired in the wrong order, processes can wait indefinitely for each other.
- **Priority inversion**
  - A low-priority process holding a lock can block a higher-priority process, delaying time-critical tasks.
- **Context-switch overhead**
  - Blocking and waking up processes requires OS intervention, which adds overhead.
- **Implementation complexity**
  - Fairness, queues, and multi-resource management make locks more complex to implement correctly.

THANK YOU FOR YOUR ATTENTION