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 – ISIL

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

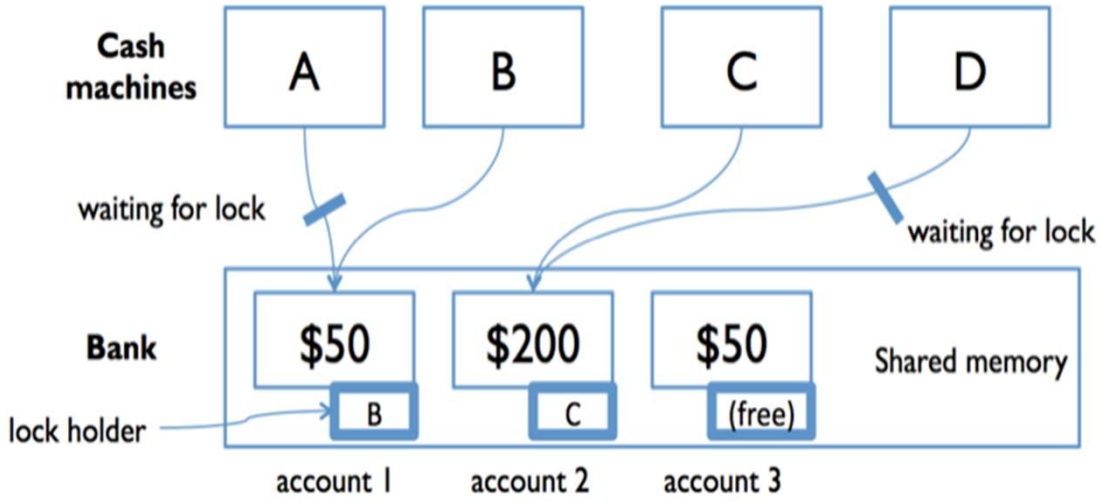
```
Entry primitive (lock);

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;
```

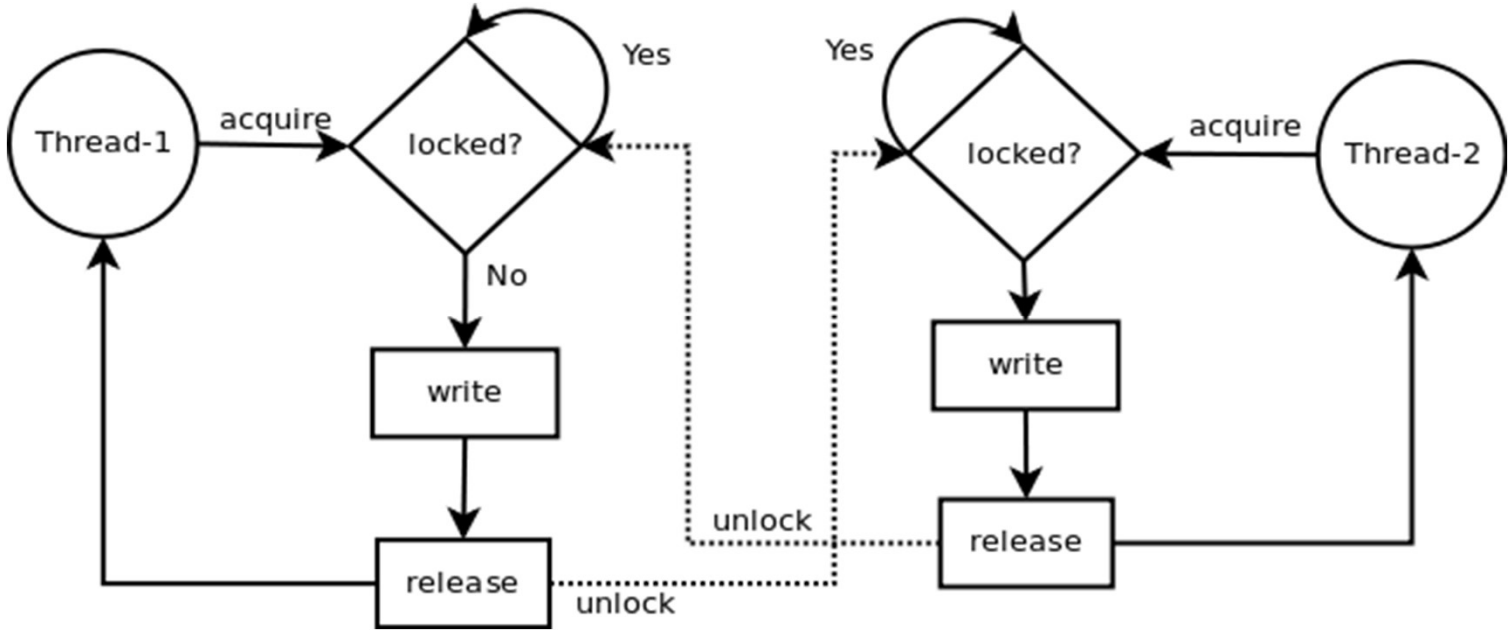
LOCKS

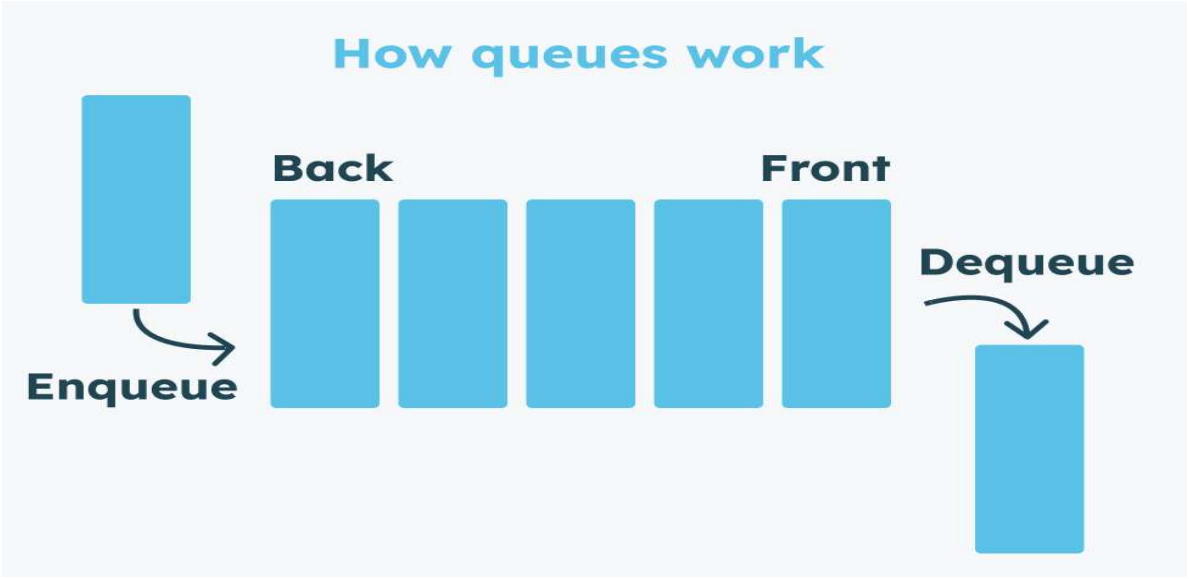



```
Process Pi;
Begin
    Repeat
        ::::
        Entry(lock);   // entry protocol
        CRITICAL SECTION;
        Exit(lock);    // exit protocol
        ::::
    Until false;
End;
```

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.

EVENTS VS LOCKS

- A lock has an owner
  - The process that calls lock becomes the owner.
  - The same process must call unlock.
  - If another process attempts to call unlock, a runtime error occurs.
- An event has no owner
  - Any process can call wait or signal.
  - There is no ownership restriction between the process that waits and the one that signals.

EVENTS VS LOCKS

- If multiple unlock operations are executed consecutively, only the first unlock has an effect
  - it releases the lock and wakes one waiting process.
  - The additional unlock calls have no effect.
- For events multiple signal calls can trigger multiple wake-ups
  - depending on how many processes are waiting.
  - Broadcast events also trigger multiple wake-ups

EVENTS VS LOCKS

- Locks are mainly used to protect critical sections and ensure mutual exclusion.
  - enforce ownership and mutual exclusion.
- Events are primarily used for synchronization and notification, not for protecting critical sections.
  - provide coordination between processes without ownership constraints.

EVENTS VS LOCKS

Aspect	Locks	Events
Ownership	Has an owner	No owner
Who can release / signal?	Only the owning process can call unlock	Any process can call signal
Error on misuse	unlock by non-owner → runtime error	No ownership error
Effect of multiple release/signal calls	Multiple unlock calls → only the first has effect	Multiple signal calls → multiple wake-ups
Main purpose	Protect critical sections (mutual exclusion)	Synchronization and notification
Blocking operation	lock blocks if already held	wait blocks until signaled
Typical use case	Prevent race conditions on shared data	Coordinate execution order between processes

EXERCISE: SYNCHRONIZATION BARRIER

- Context:
  - N threads each perform Phase 1 work independently .
  - Threads must wait at a barrier before starting Phase 2.
  - After the barrier, threads continue to Phase 2.

- Task:
  - Design an algorithm that ensures no thread starts Phase 2 until all N threads have reached the barrier.

EXERCISE: SYNCHRONIZATION BARRIER

- Principle
  - Each thread waits when it reaches the barrier.
  - Keep a counter of threads that have arrived.
  - The last thread to arrive releases all waiting threads.
  - Threads waiting for the barrier block efficiently (don’t busy-wait).

- Locks or events to resolve this problem?

EXERCISE: SYNCHRONIZATION BARRIER

- Using only locks
  - Each thread could increment a counter safely with a lock.
  - But how does a thread wait for the others efficiently?
  - Locks don’t have a “wait here until condition is true” feature.
  - threads would have to keep trying to acquire/release the lock repeatedly (busy-wait), which wastes CPU cycles.
  - So locks alone cannot make threads sleep until all arrive.

EXERCISE: SYNCHRONIZATION BARRIER

- Using only events
  - Cannot count arrivals safely: Multiple threads updating a counter without a lock causes race conditions.
  - Missed signals: Threads that start waiting after others set the event may never wake.
  - No “all arrived” check: Event cannot ensure every thread has reached the barrier.
  - So Events alone cannot make threads wait reliably until all arrive.

A correct barrier requires both a lock to protect shared counter and an event to manage sleep/wake up efficiently. Neither alone is enough.

EXERCISE: SYNCHRONIZATION BARRIER

- Lock
  - Protects shared data (counter) from race conditions.
  - Only one thread can modify the counter at a time.
- Event
  - Threads wait on an event until it is set.
  - When the last thread reaches the barrier, the event is set, releasing all waiting threads
- Flow:
  - Thread acquires the lock.
  - Increments the shared counter.
  - Checks if it's the last thread:
    - Yes: sets the event → all threads are released.
    - No: releases the lock and waits on the event.
  - All threads proceed past the barrier.

```
Const N=….; //Total number of threads participating
var
lock: Lock := unlocked;    // Mutual exclusion for counter
event: Event := non-signaled;  //Signals threads at the barrier
counter: integer := 0; //Counts threads that have reached the barrier
```

```
Process  Pi;
Begin
    PHASE 1;
    Acquire(lock);     // Safely access shared counter
    counter := counter + 1;    // Increment counter
    if counter = N then
    begin
        signal(event);   // Last thread reached barrier → release all (broadcast)
        Unlock(lock);
    end
    else
    begin
        Unlock(lock);   // Allow other threads to increment
        Wait(event);    // Wait until last thread signals event
    end;
    PHASE 2;
end;
```

THANK YOU FOR YOUR ATTENTION