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

MONITORS

3rd Year, Computer Science – ISIL

Dr. Hanene CHERAIT

2025-2026

SYNCHRONISATION

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

MONITORS

- Semaphores remain difficult to handle for complex problems.
  - A programming error in the use of the **P** and **V** primitives, or forgetting to use one of them, can have very serious consequences
    - **deadlocks**, **data inconsistencies**, and so on.
- The concept of the **Monitor** was proposed by **Hoare in 1974.**
  - to **make it easier to develop programs** that manage synchronization at a higher level
  - to **avoid programming errors** that can occur when using semaphores.

PRINCIPLE OF MONITORS

- Control synchronization through the use of a **unit** that encapsulates the definition of the **critical resource** and the **operations** that manipulate it.
- A monitor defines a set of **variables** that maintain the state of the resource, and a set of **procedures** that manipulate this resource.
- The use of a monitor is limited to **one process at a time**.
- Monitors are **fully managed by the compiler itself**; the programmer simply needs to indicate the critical sections.

DECLARATION OF A MONITOR

```
Monitor M;
Var ……..;   {declaration of shared variables}
Procedure P1(Parameters);
Begin
   …….
End;
……
……
Procedure Pn(Parameters);
Begin
   …….
End;
Begin
   Initialization of shared variables;
End;
```

A procedure is called in the usual way:

M.P1(actual parameters);

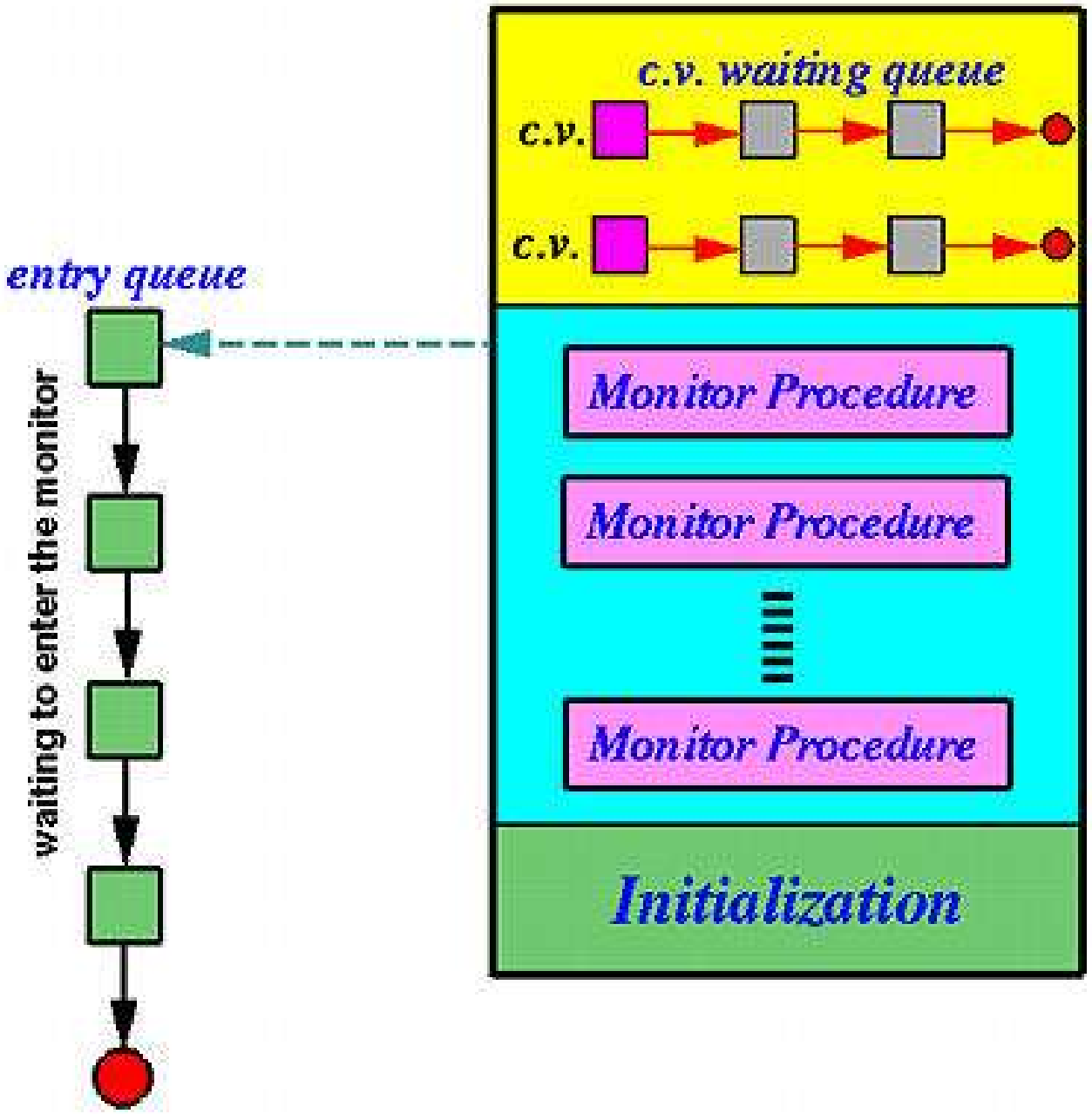
DECLARATION OF A MONITOR

- The procedures defined inside a monitor can access **only the variables declared locally within the monitor**.
- Only **one process can be active inside a monitor** at any given time, as **concurrent access to the monitor's procedures is prohibited**.
- The **initialization section** initializes the variables **before any operation on the resource is invoked**.
- The **values of the monitor's variables** can be accessed **only through the monitor's own procedures**.
- These procedures, in turn, may have **parameters and local variables**.

DECLARATION OF A MONITOR

- A monitor uses specific variable types called "Condition" variables:
  - **var C: condition;**
- Each condition C is associated with a queue containing the processes that are blocked on this condition.
- There are two primitives that act on a condition:
  - **C.wait:** causes the suspension (blocking) of the process that called it and places it in the queue associated with condition C.
    - The process will only be reactivated by a C.signal operation issued by another process.
  - **C.signal:** allows the resumption of execution of a process blocked on condition C.
    - If no process is suspended, this operation has no effect.

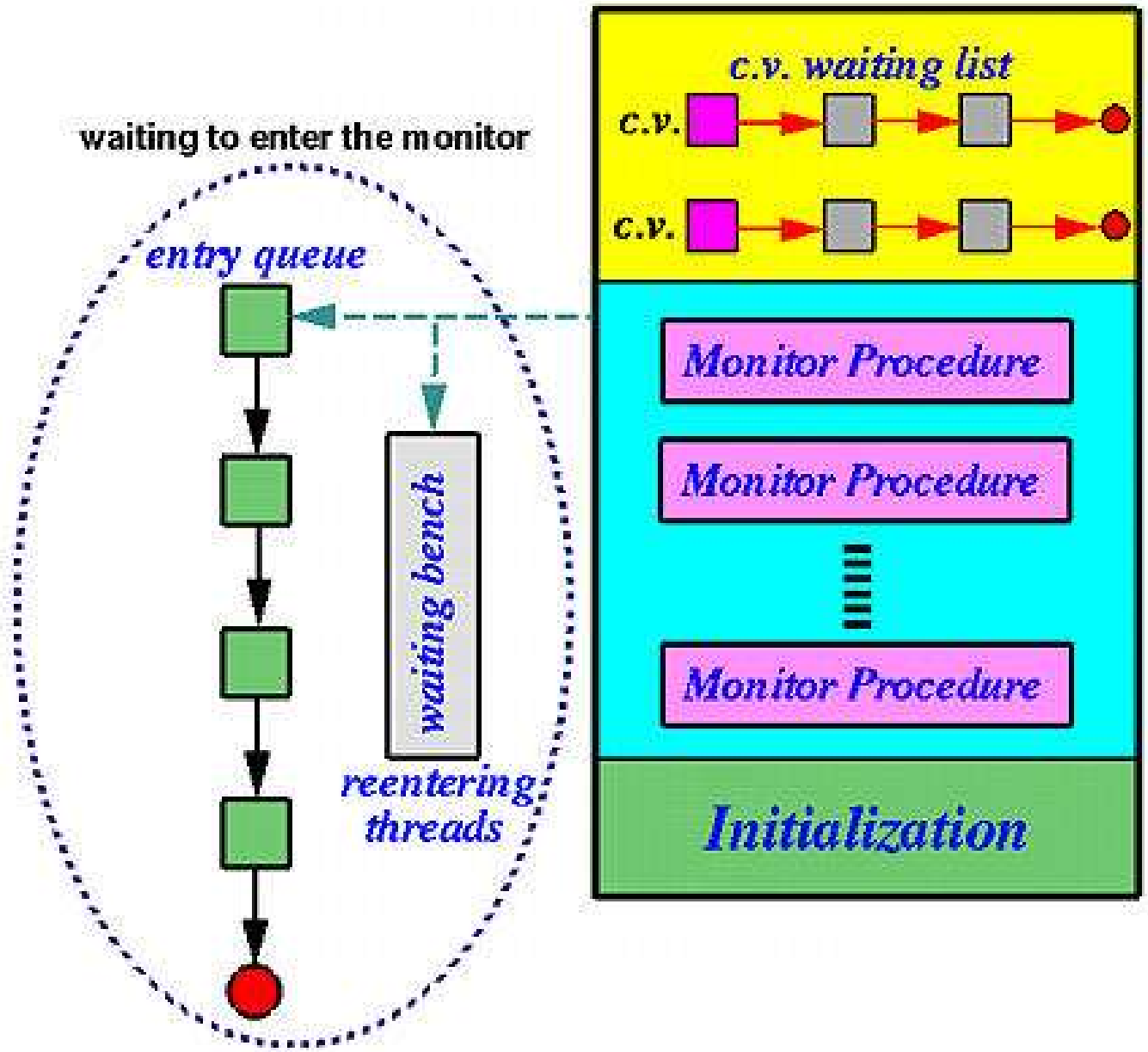
STRUCTURE OF A MONITOR



STRUCTURE OF A MONITOR

- Only **one process** can access the monitor at a time.
  - The others **wait in an entry queue**.
- The **structure of the monitor** ensures that only one process is **active** inside the monitor at any given time.
- When a process executes **C.signal**, it will be **suspended** (or ***re-entering***) until the process that was awakened **leaves the monitor**.
- Processes blocked by **signal** are **given priority** for access to the monitor **before a new process** can enter to execute a monitor procedure.

STRUCTURE OF A MONITOR



EXAMPLES OF SYNCHRONIZATION USING MONITORS

1. **Mutual Exclusion**
2. **Order Relations between Two Processes**
3. **Rendezvous Point**
4. **Producer/Consumer Problem**
5. **Readers/Writers Problem**
6. **Dijkstra's Dining Philosophers Problem**

EXAMPLES OF SYNCHRONIZATION USING MONITORS

1. **Mutual Exclusion**
2. **Order Relations between Two Processes**
3. **Rendezvous Point**
4. **Producer/Consumer Problem**
5. **Readers/Writers Problem**
6. **Dijkstra's Dining Philosophers Problem**

Common Context:

```
Var m: EM;

Process Pi;
Begin
    ……
    m.Enter();
    CRITICAL SECTION;
    m.Exit();
    ……
End;
```

```
Type EM = Monitor
Var a: condition;
       b: boolean;

Procedure Enter()
Begin
    If b then a.wait;
    b := true;
End;

Procedure Exit()
Begin
    b := false;
    a.signal;
End;

Begin
    b := false;  // Initialization
End;
```

EXAMPLES OF SYNCHRONIZATION USING MONITORS

1. Mutual Exclusion
2. Order Relations between Two Processes
3. Rendezvous Point
4. Producer/Consumer Problem
5. Readers/Writers Problem
6. Dijkstra's Dining Philosophers Problem

ORDER RELATIONS BETWEEN TWO PROCESSES

Process P0

```
begin
    A1; A2; A3;....An;
    wait for a signal from P1;
    ::::::::::::::::
end;
```

Process P1

```
begin
B1; B2; B3;....Bm;
send the activation signal to P0;
::::::::::::::::
end;
```

```
Var m : OrderMonitor;

Process P0
Begin
    A1; A2; A3; .... An;
    m.WaitForP1();
    ::::::::::::::
End

Process P1
Begin
    B1; B2; B3; .... Bm;
    m.SignalP0();
    ::::::::::::::
End
```

```
Monitor OrderMonitor
    var ready: Boolean;
    var cond: condition
Procedure SignalP0()
    Begin
        ready := true;
        cond.signal() ;
    End;
Procedure WaitForP1()
    Begin
        if not ready then
            cond.wait();
    End;
Begin
        ready:= false;
End;
```

EXAMPLES OF SYNCHRONIZATION USING MONITORS

1. Mutual Exclusion
2. Order Relations between Two Processes
3. Rendezvous Point
4. Producer/Consumer Problem
5. Readers/Writers Problem
6. Dijkstra's Dining Philosophers Problem

RENDEZVOUS POINT

```
process Pi(id: integer);
begin
        ::::::::::::::::::::::
        Rendezvous.ProcessArrives;
        ::::::::::::::::::::::
end;
```

```
process RDV;
begin
        ::::::::::::::::::::::
        Rendezvous.RDVWait;
        ::::::::::::::::::::::
end;
```

```
monitor Rendezvous;
const
    N = 5;
var
    waitingCount: integer ;
    processCond, rdvCond: condition;

procedure ProcessArrives;
begin
    waitingCount := waitingCount + 1;
    if waitingCount = N then
        rdvCond.signal;
    else
        processCond.wait;
end;
```

```
procedure RDVWait;
begin
    if waitingCount < N then
        rdvCond.wait;
    while waitingCount > 1 do
    begin
        processCond.signal;
        waitingCount := waitingCount - 1;
    end;
 end;

  begin
        waitingCount:= 0;
    end;
```

EXAMPLES OF SYNCHRONIZATION USING MONITORS

1. Mutual Exclusion
2. Order Relations between Two Processes
3. Rendezvous Point
4. Producer/Consumer Problem
5. Readers/Writers Problem
6. Dijkstra's Dining Philosophers Problem

```
BufferManager:Monitor;
```

```
Process Producer-I;
Var
   producedObject: object;
Begin
   Repeat
      Produce(producedObject);
      BufferManager.Deposit(producedObject);
   Until true;
End;
```

```
Process Consumer-j;
Var
   consumedObject: object;
Begin
   Repeat
      BufferManager.Retrieve(consumedObject);
      Consume(consumedObject);
   Until true;
End;
```

Note : Production and consumption must be done outside the monitor so as not to block other processes that want to access the count variable.

```
Monitor BufferManager;
Const
   N = …;
Var
   Buffer: Array [0..N-1] of object;
   notEmpty, notFull: condition;
   in, out, count: integer;

Procedure Deposit(obj: object);
Begin
   If count = N then notFull.wait;
   Buffer[in] := obj;
   in := (in + 1) mod N;
   count := count + 1;
   notEmpty.signal;
End;
```

```
Procedure Retrieve(var obj: object);
Begin
   If count = 0 then notEmpty.wait;
   obj := Buffer[out];
   out := (out + 1) mod N;
   count := count - 1;
   notFull.signal;
End;


Begin
   count := 0;
   in := 0;
   out := 0;
End;
```

EXAMPLES OF SYNCHRONIZATION USING MONITORS

1. Mutual Exclusion
2. Order Relations between Two Processes
3. Rendezvous Point
4. Producer/Consumer Problem
5. Readers/Writers Problem
6. Dijkstra's Dining Philosophers Problem

READERS/WRITERS PROBLEM

- Readers' priority.

```
RW: Monitor;
```

```
Process Writer_j;
Begin
    RW.StartWrite;
    Write();
    RW.EndWrite;
End;
```

```
Process Reader_i;
Begin
    RW.StartRead;
    Read();
    RW.EndRead;
End;
```

```
Monitor RW;
Var
   writeCond, readCond: Condition;
   nbReaders: integer;
   writerActive: boolean;

  Procedure StartRead ();
  Begin
     nbReaders := nbReaders + 1;
     if writerActive then readCond.wait;
        readCond.signal;
  End;

  Procedure EndRead ();
  Begin
     nbReaders := nbReaders - 1;
     if nbReaders = 0 then writeCond.signal;
  End;
```

```
Procedure StartWrite;
Begin
   if writerActive or (nbReaders <> 0) then
      writeCond.wait;
   writerActive := true;
End;
 Procedure EndWrite;
 Begin
    writerActive := false;
    if nbReaders > 0 then  readCond.signal
    else writeCond.signal;
 End;
 Begin
    nbReaders := 0;
    writerActive := false;
 End;
```

EXAMPLES OF SYNCHRONIZATION USING MONITORS

1. Mutual Exclusion
2. Order Relations between Two Processes
3. Rendezvous Point
4. Producer/Consumer Problem
5. Readers/Writers Problem
6. Dijkstra's Dining Philosophers Problem

DIJKSTRA'S DINING PHILOSOPHERS PROBLEM

```
Process Philosopher_i;  {i = process number}
Begin
   repeat
      Think();
      DiningPhilosophers.PickUpForks(i);
      Eat();
      DiningPhilosophers.PutDownForks(i);
   Until false;
End;
```

```
Monitor DiningPhilosophers;
Var
   state: array[0..4] of (Thinking, Hungry, Eating);
   i: integer;
   SelfCond: array[0..4] of Condition;
```

```
Procedure PickUpForks(i: byte);
Begin
   state[i] := Hungry;
   If (state[(i+1) mod 5] <> Eating) and (state[(i+4) mod 5] <> Eating) then
      state[i] := Eating
   Else
      SelfCond[i].Wait;
End;
```

```
Begin  {Initializations}
   For i := 0 to 4 do
      state[i] := Thinking;
End;
```

```
Procedure PutDownForks(i: byte);
Begin
   state[i] := Thinking;
   // wake up possible neighbors
   If (state[(i+1) mod 5] = Hungry) and (state[(i+2) mod 5] <> Eating) then
   Begin
      state[(i+1) mod 5] := Eating;
      SelfCond[(i+1) mod 5].Signal;
   End;

   If (state[(i+4) mod 5] = Hungry) and (state[(i+3) mod 5] <> Eating) then
   Begin
      state[(i+4) mod 5] := Eating;
      SelfCond[(i+4) mod 5].Signal;
   End;
End;
```

SEMAPHORE VS MONITOR ACROSS OS LAYERS

1 Hardware Layer

Aspect	Semaphore	Monitor
Physical existence	Not a physical device — implemented in memory using atomic hardware instructions.	Same — not a hardware object; implemented using atomic locking mechanisms.
Hardware primitives used	Uses instructions like test-and-set, compare-and-swap, xchg, LDREX/STREX.	Uses the same instructions to acquire/release the lock protecting the monitor.
Atomicity guarantee	Ensures atomic increment /decrement of a counter (the semaphore value).	Ensures atomic entry and exit to a critical section (the monitor lock).
Hardware role	Provides atomic operations to prevent race conditions on shared variables.	Provides atomic operations to prevent simultaneous access to the same monitor.
Synchronization granularity	Can manage multiple permits through atomic counter updates.	Usually one-at-a-time access — lock ensures only one thread inside the monitor.

SEMAPHORE VS MONITOR ACROSS OS LAYERS

2 Kernel Layer

Aspect	Semaphore	Monitor
Implementation level	Low-level synchronization primitive inside the OS kernel.	Higher-level abstraction built using kernel primitives like mutexes and condition variables.
Data structure	Typically: { int value; queue waiting; }	Typically: { mutex lock; condition queues; shared data; }
Thread blocking / waking	Kernel maintains a queue of processes waiting for the semaphore.	Kernel (or runtime) maintains waiting queues for threads on conditions inside the monitor.
Operation behavior	wait (P) decrements counter or blocks process; signal (V) increments and wakes a blocked process.	wait() releases lock and blocks thread; signal() wakes a waiting thread and reacquires the lock automatically.
Control granularity	Explicit control — programmer decides when to call wait() or signal().	Implicit control — monitor enforces mutual exclusion automatically.
Error handling	Misuse can cause deadlocks or starvation if not balanced correctly.	Safer — kernel or runtime enforces correct locking behavior.

SEMAPHORE VS MONITOR ACROSS OS LAYERS

3 User / Language Layer

Aspect	Semaphore	Monitor
Visibility to programmer	Exposed via system calls or libraries (e.g., POSIX sem_wait, sem_post).	Usually provided by the programming language runtime (e.g., Java synchronized, C# lock) or built manually in C.
Abstraction level	Low-level primitive — requires explicit calls to manage synchronization.	High-level abstraction — combines data, operations, and synchronization rules.
Ease of use	More error-prone (manual acquire/release).	Easier and safer (automatic mutual exclusion).
Language support	Provided as libraries (POSIX, Win32 APIs, java.util.concurrent.Semaphore).	Provided as part of the language syntax (e.g., Java, C#, but not in C).
Encapsulation	Synchronization separate from the data.	Synchronization encapsulated with the shared data inside the monitor.
Typical use	Controlling access to limited shared resources (printers, buffers, etc.).	Coordinating threads working on shared state within a module or object.

THANK YOU FOR YOUR ATTENTION