Unit 3.- Synchronization Primitives

Concurrency and Distributed Systems

Teaching Unit Objectives

- Explain the concurrent programming model provided by monitors.
- Solve the problem of conditional synchronization by means of using monitors
- Build monitors in concurrent programming languages, avoiding their potential problems.
- Evaluate existing monitor variants.

Content

- Concurrent Programming Languages
- Monitor: Concept
- Monitor Variants
- Nested calls

Concurrent Programming Languages

- Concurrent Programming must solve the needs of communication and synchronization between threads
- We can design solution strategies at different levels:

Without support of the Operating System

- Busy-waiting
- Disabling interruptions (in supervisor mode)
- TestAndSet, Swap primitives

Using Concurrent Programming Languages

- Monitors

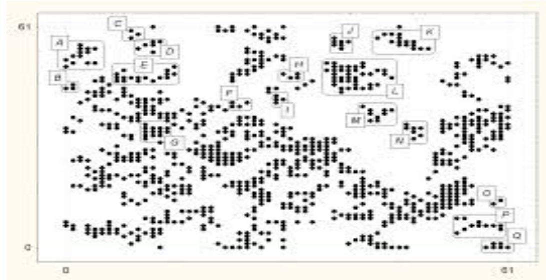
With support of the Operating System

- Semaphores
- Other tools

- Pros: Efficient and flexible
- Cons: Wrong usage cannot be detected when programming

Example.- Ants

- The ants share a territory
  - Matrix of cells, each with values free/occupied
  - Maximum one ant per cell
  - There is a *lock* that protects this territory



- Each ant is modeled by a thread

- When an ant moves from (x,y) to (x',y') executes

```
close lock;
// if occupied[x',y'] the ant must wait until the cell gets free
occupied[x',y']=true; occupied[x,y]=false; //updates matrix
open lock;
```

How can you implement this waiting?

Monitor.- Motivation

- Primitives ***open lock*** and ***close lock*** ensure secure access to shared variables
- You also need **other primitives** to safely wait **until certain logic condition is met** (synchronization)
  - A wait loop (i.e. an empty loop that repeatedly checks the condition) does not work. Why not?

```
close lock
while (occupied[x',y']) {}
      //This DOES NOT work
occupied[x',y']=true; occupied[x,y]=false
open lock
```

Monitor.- Motivation

- Most modern Programming Languages:
  - They are Object-Oriented Programming (OOP) Languages
    - Programmers can define types of data (classes)
    - A class allows defining variables (objects)
    - Distinction between interface/implementation
      - **Interface** (visible part) corresponds to its behavior (set of methods)
      - **Implementation** (hidden part).- set of attributes, methods code
  - They require concurrency

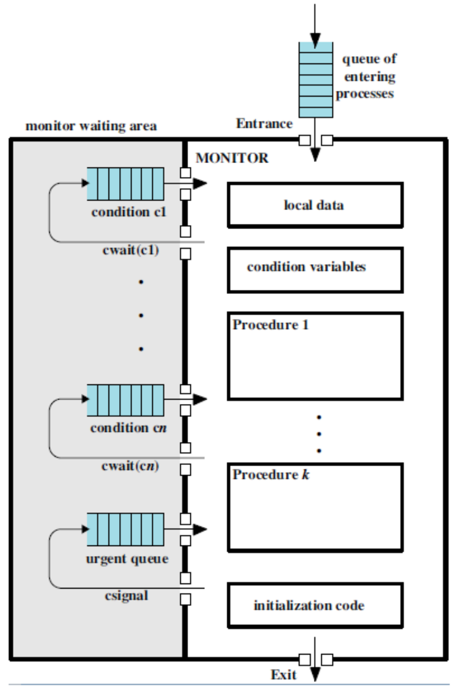
Monitor.- Motivation

- Idea.- Mixing OOP and Concurrent Programming
  - Threads are coordinated by means of **shared objects**.
  - Details of mutual exclusion and synchronization are hidden inside the classes that represent the shared objects.

- Advantages:
  - Simplifies development, maintenance and understanding of code
  - Facilitates debugging (you can test each piece separately)
  - Facilitates reusing code
  - Improves documentation and readability

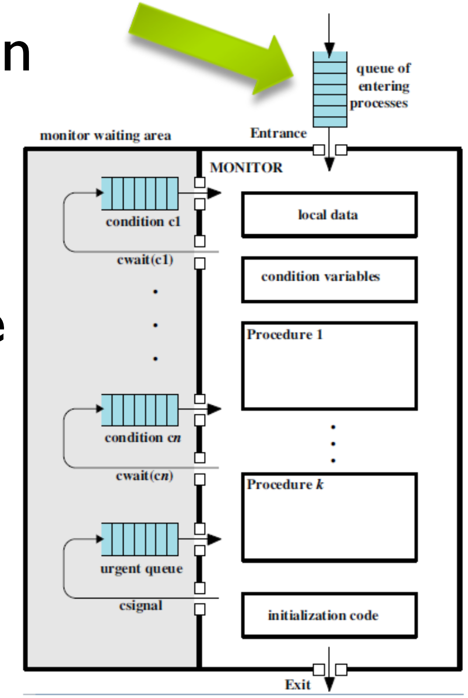
Monitor.- Concept

- **Monitor** = class to define objects that can be safely shared between different threads
  - Its methods are executed in Mutual Exclusion
  - Solves synchronization



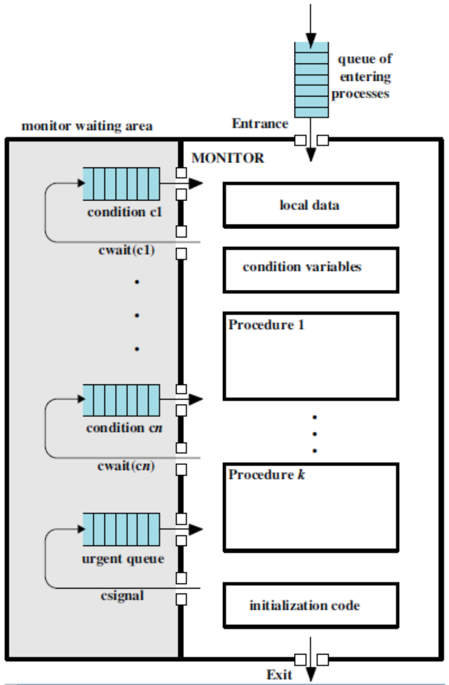
Monitor.- Concept

- **Monitor** = class to define objects that can be safely shared between different threads
  - Its methods are executed in **Mutual Exclusion**
    - It has an ***Entry queue*** for waiting there those threads that want to use the monitor when another thread is using it
    - **There are no race conditions inside the monitor**



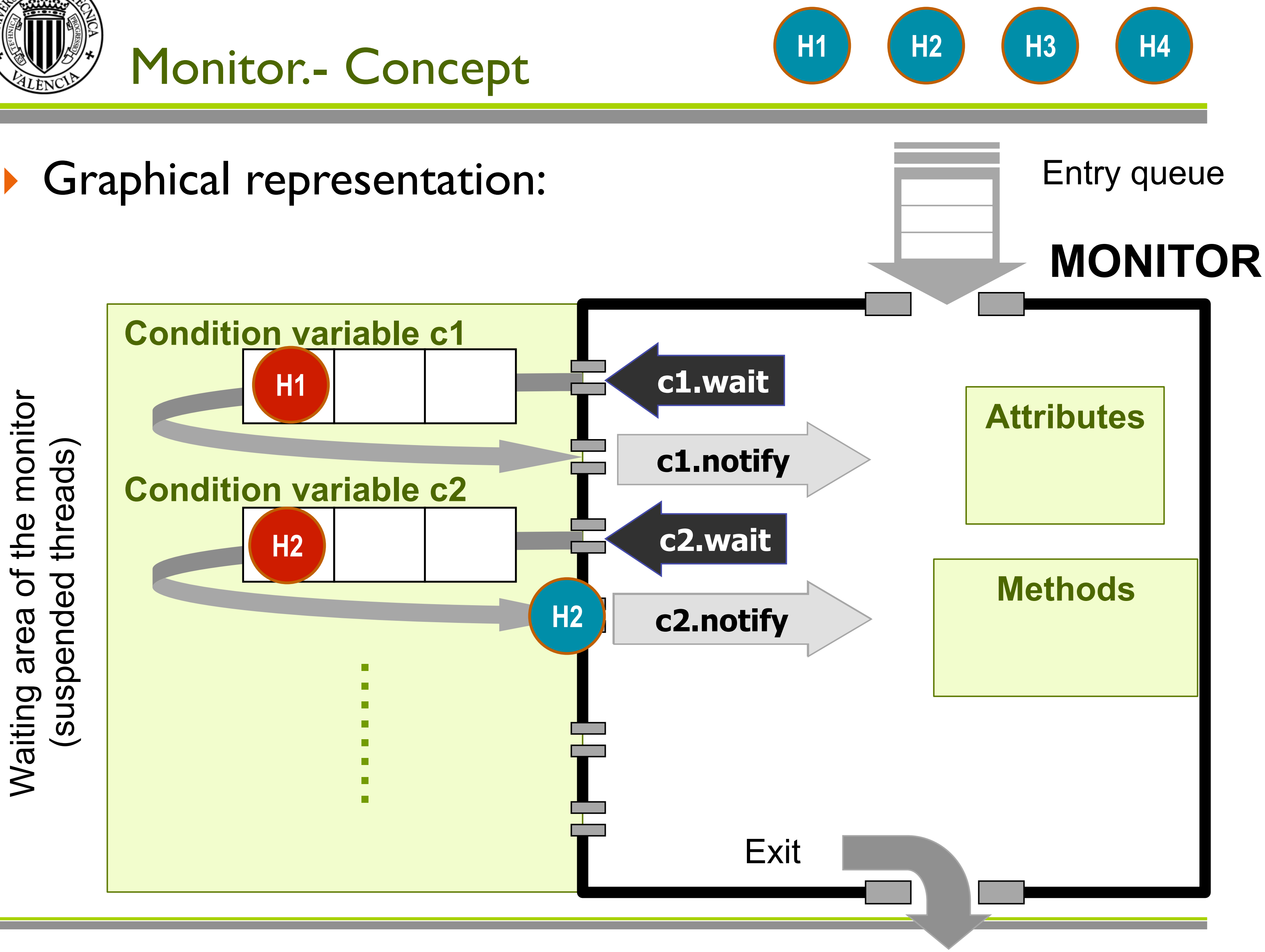
Monitor.- Concept

- **Monitor** = class to define objects that can be safely shared between different threads
  - Solves synchronization
    - You can define waiting queues (named '**condition variables**') inside the monitor
    - If a thread executes code inside the monitor it has to wait until a particular logic condition meets
      - □ Executes **c.wait()** → frees the monitor and waits at the condition c queue
    - When another thread **modifies the state** of the monitor
      - □ Executes **c.notify()** → reactivates a thread that is waiting in the condition *c* queue



Monitor.- Concept

- Graphical representation:



Monitor Example.- Producer/Consumer

- The shared object is the **buffer**
  - You design interface and implementation (attributes and method codes) like any other class

- You need to fill up a table:

Method	Waits when	Notifies to
int get()	Buffer empty	Who waits when buffer full
void put(int e)	Buffer full	Who waits when buffer empty
int numItems()	--	--

- You define a waiting queue (***condition variable***) for each case of waiting detailed in the table
  - **condition notFull, notEmpty**;

Monitor Example.- Producer/Consumer

//IMPORTANT.- THIS IS NOT JAVA, but pseudo-code

```
Monitor Buffer {
  .... //attributes for implementing the monitor
  condition notFull, notEmpty; //waiting queues
  int elems=0;

  public Buffer() {..} //initializing the attributes
  entry void put(int x) { // entry: public method with access in Mutual Exclusion
    if (elems==N) {notFull.wait();}     // waits in the notFull queue
      ... //code for inserting elements in the buffer
    elems++;
    notEmpty.notify();                  // reactivates someone from notEmpty queue
  }
  entry int get() {
    if (elems==0) {notEmpty.wait();} // waits in notEmpty queue
      ... // code to pick up elements from the buffer
    elems--;
    notFull.notify();                   // reactivates someone from notFull queue
  }
  entry int numItems() { return elems;}
}
Buffer b; //and from any thread we can invoke  b.numItems(), b.get() or b.put(x)
```

Monitor.- Summary

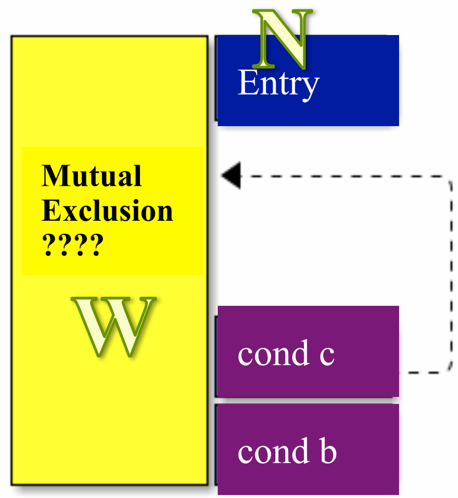
- **Monitor** = Class + Mutual Exclusion + Synchronization
- Hides details of Mutual Exclusion and Synchronization
  - Performing methods on the same monitor does not overlap (**mutual exclusion**)
  - For coordination, waiting queues are employed (named **condition variables**)
    - Primitives: ***wait**()* and ***notify**()*
      - When notifying, if there is nobody waiting, then this notification has no effect
    - These primitives can only be employed inside the monitor
    - The programmer is responsible of implementing the *wait/notify* calls in the right places.
- It provides **abstraction**
  - The programmer that invokes methods on the monitor might ignore how they have been implemented
  - The programmer that implements the monitor might ignore how its methods will be lately employed

Content

- Concurrent Programming Languages
- Monitor: Concept
- Monitor Variants
- Nested calls

Monitor.- Variants

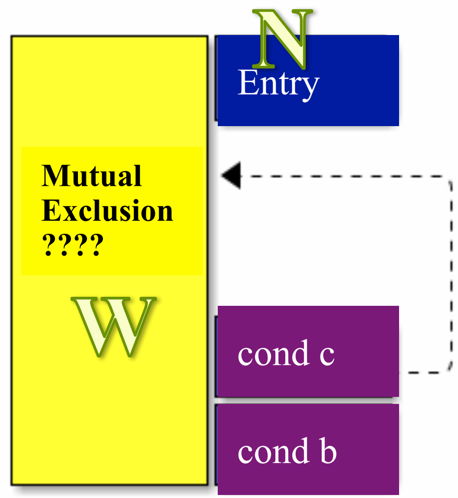
- Monitor ensures **Mutual Exclusion**
  - Only one thread executes code of the monitor at a given time
    - If another thread tries to execute code and the monitor is occupied, then this thread waits on the **Entry**.
    - When the method finishes, then the monitor gets free.
  - When the active thread (W) inside the monitor executes *c.wait()*, it goes to wait on condition c
    - **The monitor gets free** (waits outside the CS)
    - Another thread (N) waiting on the Entry becomes active inside the monitor



- Problem: if thread N executes c.notify()
    - Reactivates W
    - But only one thread (W or N) can be active inside the monitor. Which one?

Monitor.- Variants

- Monitor ensures **Mutual Exclusion**
  - Only one thread executes code of the monitor at a given time
  - When the active thread (W) inside the monitor executes ***c.wait()***, it goes to wait on condition c
    - **The monitor gets free** (waits outside the CS)
    - Another thread (N) waiting on the Entry becomes active inside the monitor
- **Problem:** if thread N executes ***c.notify()***
    - Reactivates W
    - But only one thread (W or N) can be active inside the monitor. Which one?



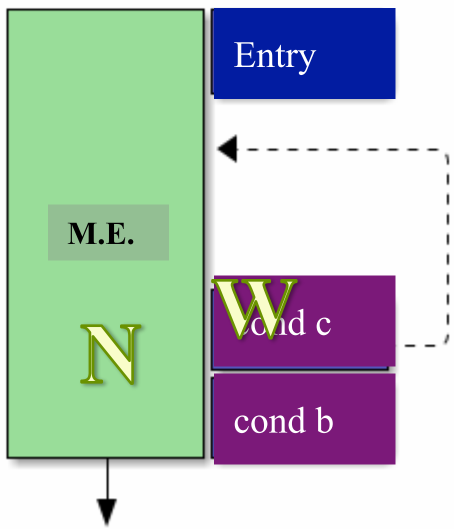
Monitor.- Variants

- Let us assume that:
  - W waits in cond c
  - N executes c.notify() and reactivates W

- Solution alternatives (monitor variants)
  - Thread N leaves the monitor (**Brinch Hansen model**)
  - Thread N waits in a special queue (**Hoare model**)
  - Thread W waits in the entry (**Lampson-Redell model**)

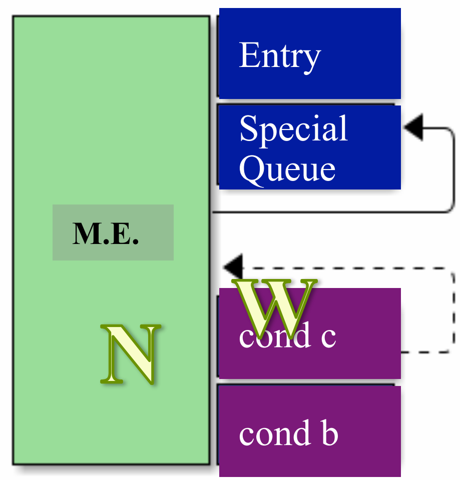
Monitor **Brinch Hansen** model

- Let us assume that:
  - W waits in cond c
  - N executes ***c.notify()*** and reactivates W
- The sentence ***notify*** must be the last sentence of the method (this is compulsory in this model)
  - N leaves the monitor and awakes thread W
- Ensures mutual exclusion
  - N leaves monitor
  - W stays active in the monitor
- It cannot be always applied
  - Some complex problems might require doing other actions after ***c.notify()***
  - "*Waterfall awake*" cannot be applied here



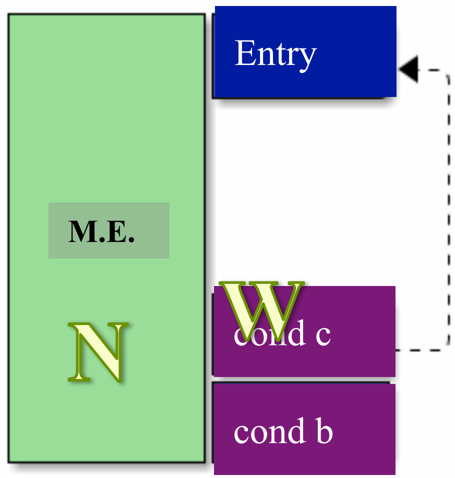
Monitor Hoare model

- Let us assume that:
  - W waits in cond c
  - N executes ***c.notify()*** and reactivates W
- Apart from entry, there is a **special queue**
  - To wait until monitor gets free
  - Priority over entry queue
- When N executes ***c.notify***
  - N goes to the special queue
  - W remains active in the monitor
- Ensures mutual exclusion
  - N waits outside the monitor
  - W stays active inside the monitor

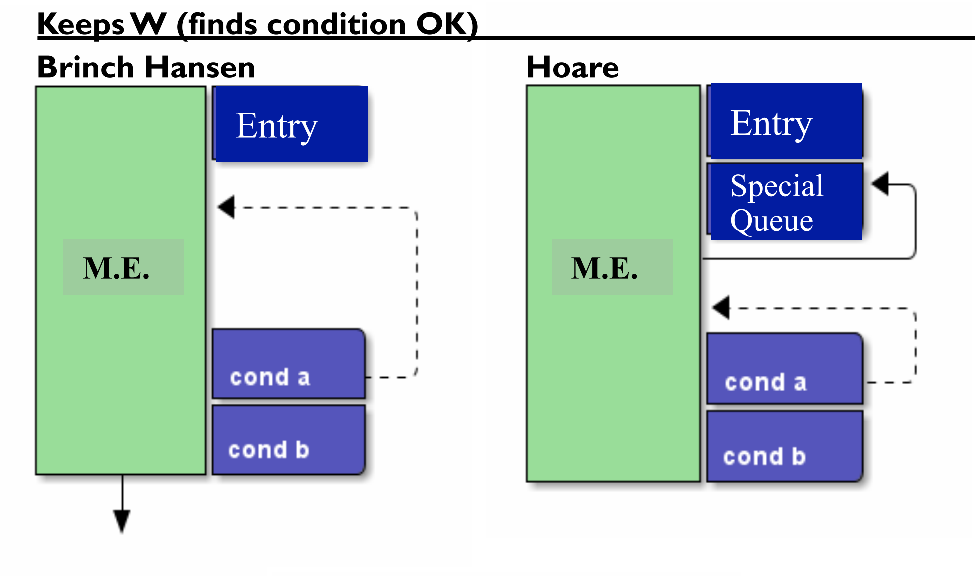


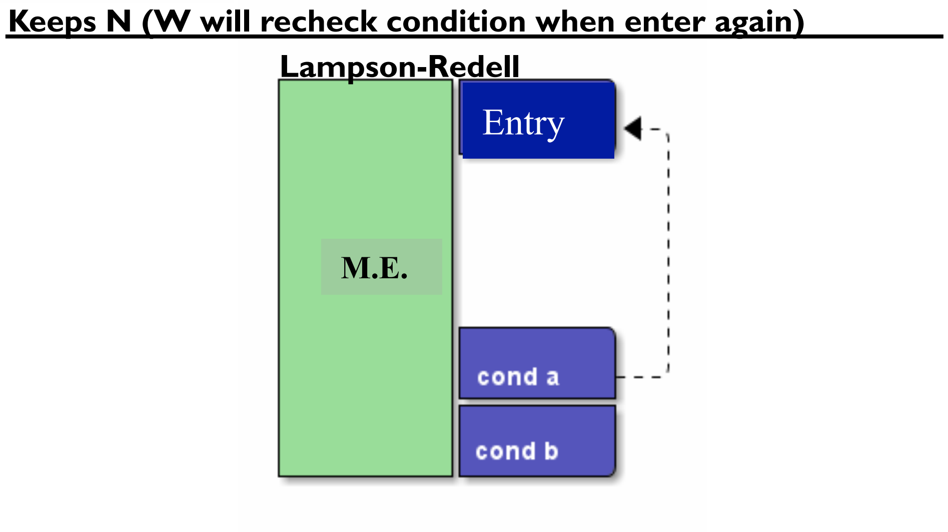
Monitor **Lampson-Redell** model

- Let us assume that:
  - W waits in cond c
  - N executes c.notify() and reactivates W
- When N executes *c.notify*
  - W goes to the entry queue
  - N remains active inside the monitor
- Ensures mutual exclusion
  - W waits outside the monitor (in the **Entry**)
    - When it enters again, the state might have changed: the thread must recheck the condition
  - N stays active inside the monitor



Monitor Variants: Summary





Monitor.- Ant example

- The territory is modeled by a monitor
  - Attributes:
    - A matrix of logical values, which indicate whether a cell is free/occupied
      - `boolean[N][N] occupied;   // This is NOT Java`
  - Methods:
    - *entry void moves(x,y,x',y')*   The ant moves from cell (x,y) to cell (x',y')

- Two solution alternatives:
  1. A waiting queue for each cell to wait till this cell gets free
     - `condition[N][N] free;   // This is NOT Java`
  2. A single waiting queue named *free*
     - `condition free;          // This is NOT Java`

Monitor.- Ant example

Alternative 1

```
Monitor Territory {
  boolean[N][N] occupied;
  condition[N][N] free;

  entry void moves(int x,y,x',y') {
    if (occupied[x',y'])
        free[x',y'].wait();
    //updates matrix
    occupied[x',y']=true;
    occupied[x,y]=false;
    //notifies who wants to go to x,y
    free[x,y].notify();
  }
}
```

Alternative 2

```
Monitor Territory {
  boolean[N][N] occupied;
  condition free;

  entry void moves(int x,y,x',y') {
    while (occupied[x',y']) {
      free.wait();
      free.notify();
    }
    //updates matrix
    occupied[x',y']=true;
    occupied[x,y]=false;
    //notifies who wants to go to x,y
    free.notify();
  }
}
```

Monitor.- Ant example

- Alternative 1:
  - It is much more efficient
    - Only reactivates an ant if the cell it was waiting for has become free.
  - *if* clause only possible in some monitor variants (to be discussed later). The other variants require a *while* clause.
- Alternative 2:
  - It is less efficient
    - Employs "**waterfall awake**". At worst case, all suspended ants can be reactivated when a cell gets free.
  - It cannot be applied in the Brinch Hansen monitor variant.
  - It is the only option if you cannot define multiple waiting queues (i.e. multiple condition variables).
- Whenever possible, you must choose alternative 1.

Java.- Monitor

- Two possible levels:
  - Basic support in language
  - Extended support (by means of *java.util.concurrent* library)

- In this unit we focus on the basic support
  - *Java.util.concurrent* is detailed in Unit 5

Java supports the monitor concept

- Every object has in an **implicit way** (you do not need to declare):
  - A lock
    - If we label a method with the word ***synchronized***, this ensures execution with mutual exclusion
      - This is equivalent to call *close lock()* before its first instruction and call *open lock()* after its last instruction
  - A waiting queue with primitives:
    - ***wait***() wait on the *waiting queue*
    - ***notify***() reactivates one of the threads that waits on the *waiting queue*
    - ***notifyAll***() notifies all threads that are waiting
- But here you **cannot** declare other locks nor other waiting queues.

Java.- How to define a Monitor

- A class that defines objects to be shared among threads should:
  - Define all its attributes as **private**
  - Synchronize all its non-private methods (using *synchronized*)
  - In the implementation of each method, access only to class attributes and to local variables (defined in this method)
  - Use *wait(), notify(), notifyAll()* inside synchronized methods

- IMPORTANT: The compiler does not check anything (it is all the responsibility of the programmer)
  - There is no warning or error if there are non-private attributes, or public methods without the synchronized label, or we employ wait(), notify(), notifyAll() inside a non-synchronized method.

Java.- How to define a Monitor (cont.)

- In an ideal monitor, threads that wait for **different** logical conditions have to wait in **different queues**
  - E.g.- in Producer/Consumer you can have *notEmpty*, *notFull* queues.
    - Producers that find the buffer full have to wait in the *notFull* queue
    - Consumers that wait because the buffer is empty stay in the *notEmpty* queue
- But Java only employs **one condition variable** per monitor
  - Threads that wait for different logical conditions **have to wait in a unique queue** (thus, in the same queue) and not in different queues
  - So when reactivating a thread you do not know if this thread was waiting for one condition or another
    - Except in very simple cases, it is recommended to wake up all threads and each of them has to recheck its condition
  - *java.util.concurrent* library (see Unit 5) solves this limitation

Java.- How to define a Monitor (cont.)

- The typical scheme of a method in a shared object in Java is:

```
while (logicalCondition){ // has to wait while the condition is true
        wait();       // try { wait } catch (InterruptedException e) {..}
};
... // normal code
notifyAll(); // if we have modified the state of the object, we have to notify all threads that are waiting
```

Java.- Ant example

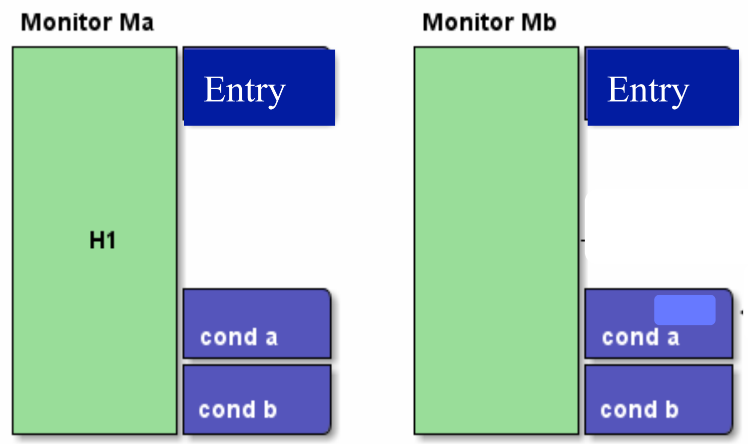
```
public class Territory{
   private boolean[][] occupied;

   public Territory(int N) {
         occupied=new boolean[N][N];
         for (int i=0; i<N; i++)
            for (int j=0; j<N; j++)
                  occupied [i][j]=false; //free
   }

   public synchronized void moves(int x0, int y0, int x, int y) {
         while (occupied[x][y])
                  try { wait(); } catch (InterruptedException e) {};
         occupied [x0][y0]=false; occupied[x][y]=true;
         notifyAll();
   }
}
```

Content

- Concurrent Programming Languages
- Monitor: Concept
- Monitor Variants
- Nested calls

Monitor.- nested calls

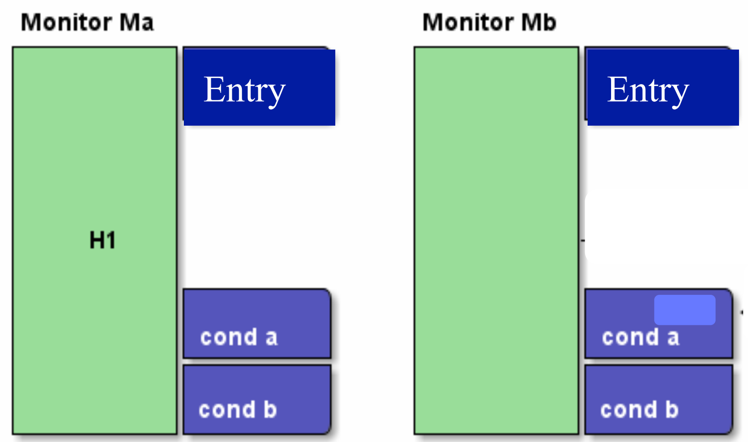
- Calling from one monitor to a method of another monitor can:
  - reduce concurrency
  - and even cause deadlocks.



- Example: Let us suppose…
  - 2 monitors Ma and Mb:
    - from one method of Ma we invoke a method of Mb and vice versa
  - 2 threads H1 and H2

Monitor.- nested calls

- H1 active in Ma, invokes a method of Mb, in which it executes ***a.wait()***
  - Goes to the waiting queue "a" of monitor Mb
  - Releases monitor Mb, but not Ma
    - Nobody else can use Ma.- we reduce concurrency



- If H2 enters in Mb (which was free) and invokes a method of monitor Ma
  - Waits in the entry queue of Ma (since Ma is occupied)
  - It does not release monitor Mb
  - We have reached a **deadlock**

Nested calls.- 1st Example of deadlock

```java
public class Problem {
  public synchronized void hello() {...}
  public synchronized void test (Problem x) { x.hello(); }
}
```

- Two threads H1, H2
- Two variables `Problem p1, p2`

```
H1 { p1.test(p2) }
H2 { p2.test(p1) }
```

What happens here?

Nested calls.- 2nd Example of deadlock

- We define two monitors (p, q) of type Bcell
- We assume 2 concurrent threads H1 and H2
  - H1 invokes **p.swap(q)**, obtains access to monitor p and starts execution of *p.swap*
  - H2 invokes **q.swap(p)**, obtains access to monitor q and starts execution of *q.swap*

What happens here?

```
class BCell {
  int value;
  public synchronized void getValue() {
    return value;
  }
  public synchronized void setValue(int i)
{
    value=i;
  }
  public synchronized void swap(BCell x)
{
    int temp= getValue();
      setValue(x.getValue());
      x.setValue(temp);
  }
}
```

Nested calls.- 2nd Example of deadlock

- **<u>Threre is a deadlock:</u>**
  - Inside ***p.swap***, H1 invokes ***q.getValue()***, but it must wait because monitor q is not free
  - Inside ***q.swap***, H2 invokes ***p.getValue()***, but it must wait because monitor p is not free
  - Both are waiting to each other, and the situation cannot evolve → **<u>DEADLOCK</u>**

```
class BCell {
  int value;
  public synchronized void getValue() {
    return value;
  }
  public synchronized void setValue(int i)
{
    value=i;
  }
  public synchronized void swap(BCell x)
{
    int temp= getValue();
      setValue(x.getValue());
      x.setValue(temp);
  }
}
```

Learning results of this Teaching Unit

- At the end of this unit, the student should be able to:
  - Program efficient solutions to the conditional synchronization problem, using monitors.
  - Design a new monitor suitably, attending to the conditions to be managed.
  - Compare the existing monitor variants.
    - Classify concurrent programming languages depending on the monitor variant that they support.