Parallelism and concurrency

Lesson 1

Georges Da Costa
georges.da-costa@irit.fr

Thanks for lots of pictures from Internet, credits coming soon

Synchronization using Monitors

Left and Right

2 groups of people, some on left, some on right

Only one side can shout

Thread right

```
shout
```

Thread left

```
shout
```

Needs synchronization. How to do this with only two peoples? With more?

Thread right

```
mutex.lock
shout
mutex.unlock
```

Thread left

```
mutex.lock
shout
mutex.unlock
```

Left and Right

2 groups of people, some on left, some on right

Only one side can shout

Thread right

```
shout
```

Thread left

```
shout
```

Is it sufficient?

Thread right

```
If left is not shouting:
    mutex.lock
    shouting = right
    shout
    mutex.unlock
```

Thread left

```
If right is not shouting:
    mutex.lock
    shouting = left
    shout
    mutex.unlock
```

Left and Right

2 groups of people, some on left, some on right

Only one side can shout

Thread right

```
    shout
```

Thread left

```
    shout
```

Needs synchronization. How to do this with only two peoples? With more?

- Need a state: either left or right (or none)
- If the mode is the one of a thread
  - The thread can shout
- Otherwise, it needs to wait a condition
  - The test is on the mode value

Must be done without busy waiting

Hoare Monitors

Proposed in 1974 by Hoare et al.

Higher level abstraction than semaphores

- Shared state
- Manage operations on the state
- Operations can depend on the state

Hoare Monitors

Proposed in 1974 by Hoare et al.

Higher level abstraction than semaphores

- Shared state
- Manage operations on the state
- Operations can depend on the state

Thread right

```
If left is shouting:
    Wait for the left side to finish
shout
```

Thread left

```
If right is shouting:
    Wait for the right side to finish
shout
```

Need of synchronization, specific critical sections, ...

Hoare Monitors

Monitor: An “object” with methods, shared variables, and conditions. All methods are mutually exclusive.

```
Monitor m {
     int i;
     void inc() { i++; }
     void dec() {i--;}
}
```

```
i=0
```

Thread 1:

```
for i from 0 to 100:
     inc()
```

Thread 2:

```
for i from 0 to 100:
     dec()
```

```
print(i)
```

Conditions

Thread-safe concurrent condition

- Is there so data to process?
- Has another thread finished its processing?
- Do I have the right to be executed?

Global idea:

- A condition is linked to a particular behavior
- Allows to check the state (variables) in an exclusive way

Conditions

Thread-safe concurrent condition

- Is there so data to process?
- Has another thread finished its processing?
- Do I have the right to be executed?

Global idea:

- A condition is linked to a particular behavior
- Allows to check the state (variables) in an exclusive way

Condition: queue of threads linked to a mutex

Operations on **c** (condition) associated to **m** (mutex):

- **c.wait(mutex)**:
  - Unlock a mutex and put the current process in the queue in a sleep state. When exiting the wait, the mutex is locked again.
- **c.signal() or c.notify()**:
  - Wakes the first process in the queue and marks it as ready to be executed. Does nothing if no process is waiting
- **c.empty()**

Hoare Monitor

```
Monitor_ping_pong {
      Condition c_ping, c_pong;
      string next = “ping”

      void ping() {
            while next == “pong”:
                  c_ping.wait()
            next = “pong”
            c_pong.notify()
      }

      void pong() {
            while next == “ping”:
                  c_pong.wait()
            next = “ping”
            c_ping.notify()
      }

}
```

To use with:

Process 1:

```
      while(True):
            ping()
```

Process 2:

```
      while(True):
            pong()
```

Why putting the while outside of the monitor?

Hoare Monitor

```
Monitor_ping_pong {
      Condition c_ping, c_pong;
      string next = “ping”

      void ping() {
            while next == “pong”:
                  c_ping.wait()
            next = “pong”
            c_pong.notify()
      }

      void pong() {
            while next == “ping”:
                  c_pong.wait()
            next = “ping”
            c_ping.notify()
      }

}
```

To use with:

Process 1:

```
    while(True):
        ping()
```

Process 2:

```
    while(True):
        pong()
```

Process 3:

```
    while(True):
        pong()
```

What is changed ?

Hoare Monitor

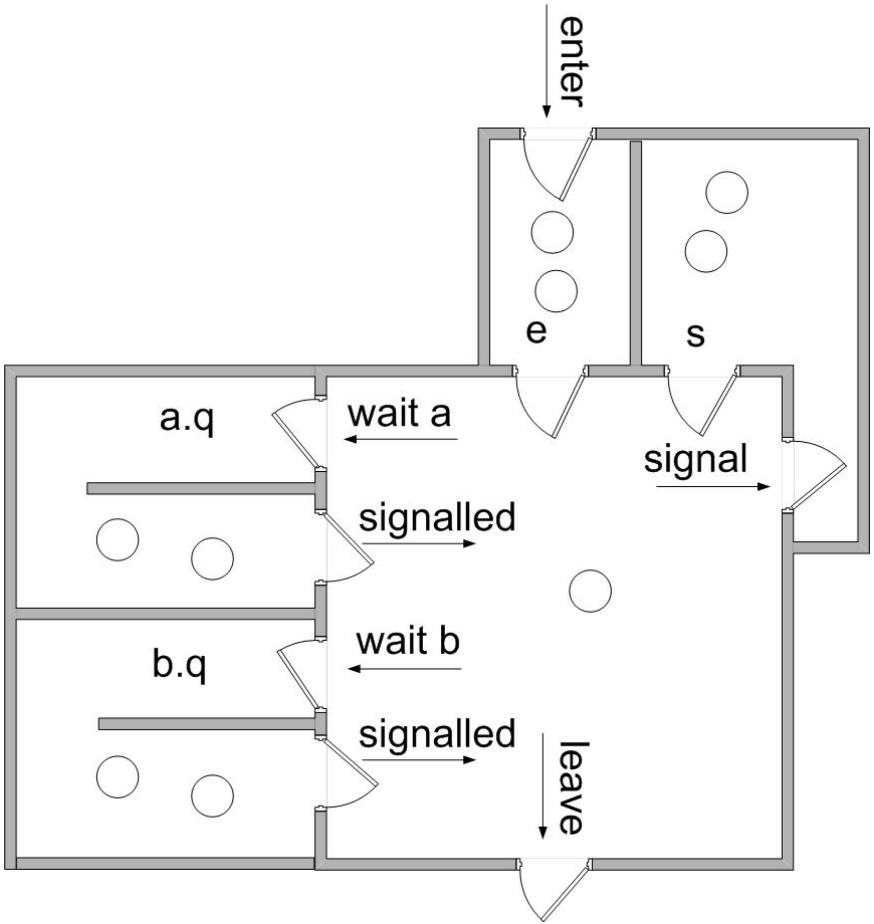
```
Monitor name {
     State variables
     Condition variables
     Functions
}
```

- Only one execution flow can use any of the functions at the same time
- Can be used for synchronizing threads or processes
- All conditions are independant
- `wait(0)` goes first

Hoare Monitor

```
Monitor name {
     State variables
     Condition variables
     Functions
}
```

- Only one execution flow can use any of the functions at the same time
- Can be used for synchronizing threads or processes
- All conditions are independant
- Signal stops the current flow (put it in URGENT) and starts directly the waiting one
- `wait(0)` goes first



Mutex using Hoare Monitors

```
Monitor mutexMonitor :
     bool locked=False
     Condition access

     lock()
          if locked:
               access.wait
          locked = True

     unlock()
          locked = False
          access.signal
```

Mutex using Hoare Monitors

```
Monitor mutexMonitor {
    bool locked=False
    Condition access

    lock()
        if locked:
            access.wait
        locked = True

    unlock()
        access.signal
        locked = False
```

is it the same?

Mutex using Hoare Monitors

```
Monitor mutexMonitor {
    bool locked=False
    Condition access

    lock()
        if locked:
            access.wait
        locked = True

    unlock()
        access.signal
        locked = False
```

is it the same?

1. P1: `lock()`
   a. `locked` changes to `True`
2. P2: `lock()`
   a. Waits because `locked` is `True`
3. P1: `unlock()`
   a. Starts with the signal
   b. Goes back to lock and changes `locked` to `True`
   c. Goes back to unlock and changes back `locked` to `False`

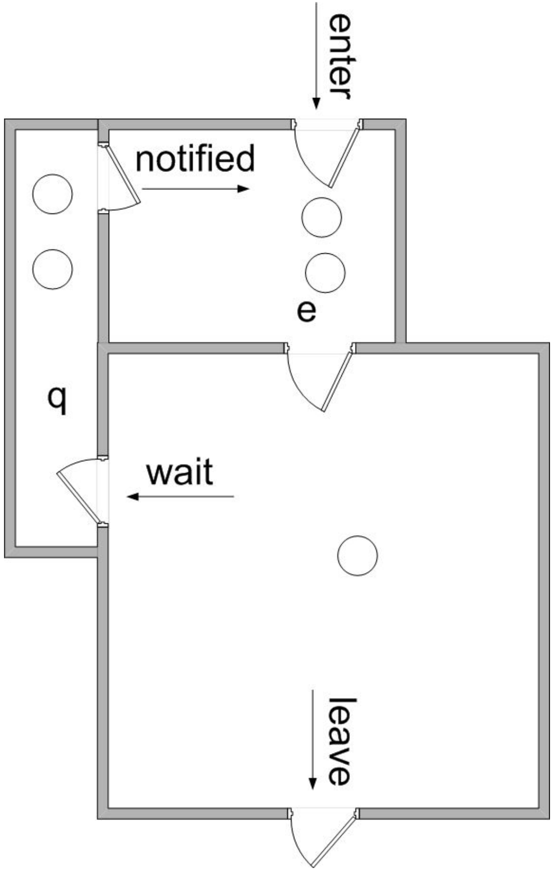
P2 should have the lock but `locked` is `False`

Mesa (Lampson / Redell) Monitors

Main difference:

- Replace **signal** by **notify**
- Notify moves a waiting thread to the ready pool
- The current threads finishes the function before returning

Does it changes anything in the code?



Mesa (Lampson / Redell) Monitors

Main difference:

- Replace **signal** by **notify**
- Notify moves a waiting thread to the ready pool
- The current threads finishes the function before returning

Does it changes anything in the code?

```
Monitor mutexHoareMonitor {
    bool locked=False
    Condition access

    lock()
        if locked:
            access.wait
        locked = True

    unlock()
        locked = False
        access.signal
```

Mesa (Lampson / Redell) Monitors

Main difference:

- Replace **signal** by **notify**
- Notify moves a waiting thread to the ready pool
- The current threads finishes the function before returning

Does it changes anything in the code?

```
Monitor mutexMesaMonitor {
    bool locked=False
    Condition access

    lock()
        while locked:
            access.wait
        locked = True

    unlock()
        locked = False
        access.notify
```

Philosophers using Monitors

```
Monitor table {
     available[5] = [True, True, ...]
     condition cond_phi[5]

     void put_chopsticks(pos):
          available[pos]=True
          available[pos+1%5]=True
          prev = (pos-1)%5
          next = (pos+1)%5
          if available[prev] and
             not cond_phi[prev].empty
               cond_phi[prev].notify
          elif available[next+1%5] and
               not cond_phi[next].empty
               cond_phi[next].notify

     void take_chopsticks(pos):
          while not available[pos] or
             not available[(pos+1)% 5]:
               cond_phi[pos].wait
          available[pos]=False
          available[(pos+1)%5]=False
}
```

Thread Philosopher

```
     While True:
          Think
          take_chopsticks
          Eat
          put_chopsticks
```

Philosophers using Hoare Monitors

Why Eat and Think are not in the monitor?

Can it leads to deadlock?

Can it leads to starvation?

Is it using busy waiting?

Philosophers using Hoare Monitors

Why Eat and Think are not in the monitor?	Can be done in parallel
Can it leads to deadlock?	No
Can it leads to starvation?	Yes: Your two neighbors can alternate and you can wait an infinite time
Is it using busy waiting?	No

Methodology

Steps to design a Monitor

4 steps:

1. Define the specification of the monitor
2. Blocking and unblocking conditions
3. Deduce Conditions, state variables
4. Implement

Steps to design a Monitor

4 steps:

1. **Define the specification of the monitor**
   a. List the methods in the monitor
2. Blocking and unblocking conditions
3. Deduce Conditions, state variables
4. Implement

```
Monitor PingPong {

    void ping()

    void pong()

}
```

Steps to design a Monitor

4 steps:

1. Define the specification of the monitor
    a. List the methods in the monitor
2. **Blocking and unblocking conditions**
    a. List the reasons for processes to be blocked or unblocked
    b. Blocking: State; Unblocking: event
3. Deduce Conditions, state variables
4. Implement

- Ping
    - Blocking
        - Ping or Pong running
        - We want the next to be Pong
    - Unblocking
        - A Pong just finished
- Pong
    - Blocking
        - Ping or Pong running
        - We want the next to be Ping
    - Unblocking
        - A Ping just finished

Steps to design a Monitor

4 steps:

1. Define the specification of the monitor
   a. List the methods in the monitor
2. Blocking and unblocking conditions
   a. List the reasons for processes to be blocked or unblocked
   b. Blocking: State; Unblocking: event
3. **Deduce Conditions, state variables**
   a. Variable: the state
   b. Conditions: One by similar behavior
4. Implement

- Variables
  - Is Pong or Ping running `next`
  - What was the last running
- Conditions
  - All Ping processes are similar
    - `c_ping`
  - All Pong processes are similar
    - `c_pong`

Steps to design a Monitor

4 steps:

1. Define the specification of the monitor
    a. List the methods in the monitor
2. Blocking and unblocking conditions
    a. List the reasons for processes to be blocked or unblocked
    b. Blocking: State; Unblocking: event
3. Deduce Conditions, state variables
    a. Variable: the state
    b. Conditions: One by similar behavior
4. **Implement**
    a. Check the whole list (2.a) is present !

```
Monitor ping_pong {
      Condition c_ping, c_pong;
      string next = “ping”

      void ping() {
            while next == “pong”:
                  c_ping.wait()
            next = “pong”
            c_pong.notify()
      }

      void pong() {
            while next == “ping”:
                  c_pong.wait()
            next = “ping”
            c_ping.notify()
      }

}
```

Design a monitor

Pizzeria

1. Clients who want to command a pizza
2. Waiter who takes only one of the command and pass it to the cook
3. The cook that takes the command and bring it himself to the client

What would be the specification ?

Waiter:

Loop:

    Walk

    Take command

    Walk

    Put command in kitchen

Design a monitor: specification of the monitor

Pizzeria

1. Clients who want to command a pizza
2. Waiter who takes only one of the command and pass it to the cook
3. The cook that takes the command and bring it himself to the client

What would be the blocking and unblocking conditions?

```
Monitor pizzeria {
     command(pizza)
     send_to_kitchen()
     exit()
}
```

Design a monitor: blocking and unblocking conditions

Pizzeria

1. Clients who want to command a pizza
2. Waiter who takes only one of the command and pass it to the cook
3. The cook that takes the command and bring it himself to the client

What would be the variables, conditions?

- Blocking:
  - Client: pizza is not finished or a client is waiting its pizza
  - Waiter: no pizza in command
  - Cook: no pizza sent to kitchen
- Unblocking
  - Client: pizza just finished or client just took its pizza
  - Waiter: a client just asked for a pizza
  - Cook: a new command arrives to the kitchen

Design a monitor: variables and conditions

Pizzeria

1. Clients who want to command a pizza
2. Waiter who takes only one of the command and pass it to the cook
3. The cook that takes the command and bring it himself to the client

What would be the code?

- Variable:
  - Command of a client
  - Message to the kitchen
  - Final pizza
- Conditions. Four different behaviors
  - Client before command, client after the command, waiter, cook
  - Four conditions

Design a monitor: specification of the monitor

Pizzeria

1. Clients who want to command a pizza
2. Waiter who takes only one of the command and pass it to the cook
3. The cook that takes the command and bring it himself to the client

```
Monitor pizzeria {
      String command, kitchen, result
      Condition client, waiter, cook, go
      command(pizza)
            while command != “none”:
                  client.wait
            command = pizza
            waiter.notify()
      send_to_kitchen():
            while command == “none”:
                  waiter.wait
            kitchen = command
            cook.signal
      cook():
            while kitchen == “none”:
                  cook.wait
            result=kitchen
            kitchen = “none”
            go.signal
      wait_pizza():
            while result == “none”:
                  go.wait
            result = “none”
            command = “none”
            client.signal
}
```

Post Box

Same questions:

- One postbox
- Two operations: post, take

1. Define the functions of the monitor
2. Blocking and unblocking conditions
3. Deduce Conditions, state variables
4. Implement

Producer - consumer

Producer - consumer

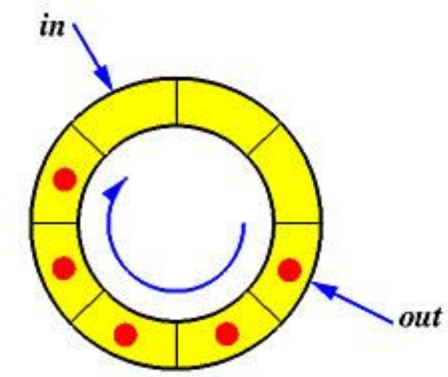
Producers and consumers share a bounded buffer of **N** elements. Assumption: most of the time is the actual production and consumption

Multiple producers

- Ensure production in free space

Multiple consumers

- Ensure consuming something existing



Producer thread:

```
val=produce_something
drop_off(val)
```

Consumer thread:

```
val = pick_up()
do_something_with(val)
```