Parallelism and concurrency

Lesson 2

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

Thanks for lots of pictures from Internet, credits coming soon

The door

The door

Assume a meeting of numerous people:

- Opener: always try to open the door, feels bad if it cannot be done
- Closer: always try to close the door, feels bad if it cannot be done
- Walker: absolutely wants to go to the other side of the door to see what is there

What are the processes?

The door

Assume a meeting of numerous people:

- Opener: always try to open the door, feels bad if it cannot be done
- Closer: always try to close the door, feels bad if it cannot be done
- Walker: absolutely wants to go to the other side of the door to see what is there

What is the specification of the monitor?

```
Process walker:
    Loop:
        walk(randomTime)
        goThroughTheDoor(randomTime)


Process opener:
    Loop:
        walk(randomTime)
        Open the door

Process closer:
    Loop:
        walk(randomTime)
        Close the door
```

The door

Assume a meeting of numerous people:

- Opener: always try to open the door, feels bad if it cannot be done
- Closer: always try to close the door, feels bad if it cannot be done
- Walker: absolutely wants to go to the other side of the door to see what is there

What are the blocking and unblocking conditions?

```
Monitor Door {
    open()
    close()
    start_walk_through()
    end_walk_through()
}
```

The door

Assume a meeting of numerous people:

- Opener: always try to open the door, feels bad if it cannot be done
- Closer: always try to close the door, feels bad if it cannot be done
- Walker: absolutely wants to go to the other side of the door to see what is there

Variables, conditions?

	Blocking	Unblocking
Opener	Door is opened	Door just closed
Closer	Door is closed OR a walker is inside	Door just opened OR last walker left
Walker	Door is closed	Door just opened or a walker started to walk

The door

Assume a meeting of numerous people:

- Opener: always try to open the door, feels bad if it cannot be done
- Closer: always try to close the door, feels bad if it cannot be done
- Walker: absolutely wants to go to the other side of the door to see what is there

```
Monitor Door {
    bool opened = False
    int nb_walkers = 0
    condition opener, closer, walker

    open()
    close()
    start_walk_through()
    end_walk_through()

}
```

The door

```
Monitor Door {
     Bool opened = False
     Int nb_walkers = 0
     Condition opener, closer, walker

     open()
     close()
     start_walk_through()
     end_walk_through()
}
```

```
Process walker:
     Loop:
          walk(randomTime)
          start_walk_through()
          goThroughTheDoor(randomTime)
          end_walk_through()

Process opener:
     Loop:
          walk(randomTime)
          open()

Process closer:
     Loop:
          walk(randomTime)
          close()
```

The door

1. In each group, everyone selects a role (opener, closer, walker). Each role must be present at least once
2. Use an object for each *process* i.e. yourself
3. Use an area (or a paper sheet) for storing processes blocked in each **condition** or in the monitor **mutex**, or **ready**
4. Use a paper to store the shared variables state

```
Monitor Door:
    bool opened = False
    int nb_walkers = 0
    condition opener, closer, walker

    open()
        while opened:
            opener.wait()
        opened = True
        if not walker.empty()
            walker.notify()
        else
            closer.notify()
    close()
        while not opened or nb_walkers>0
            closer.wait()
        opened = False
        opener.notify()
```

```
start_walk_through()
    while not opened
        walker.wait()
    nb_walkers++
    walker.notify()

end_walk_through()
    nb_walkers--
    if nb_walkers == 0
        closer.notify()
```

Readers - Writers

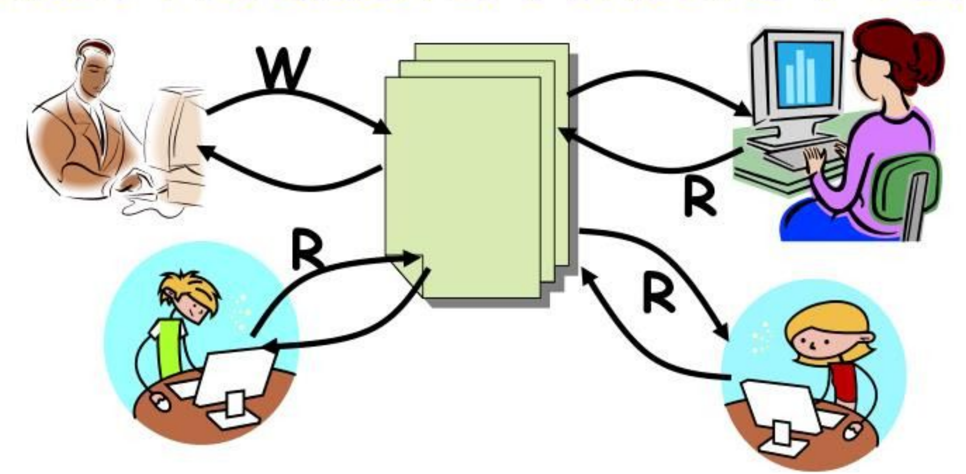
Readers - Writers

Classical usage

- Data Base
- Web Site
- Weather service

Two types of users

- Readers: never modify the data
- Writers: modify the data



Is a simple mutex can work? If no, are there corner cases where it works?

Readers - Writers

Actual constraints

- Readers can read at the same time
- Writers are incompatible with everything else (writers and readers)

Readers - Writers

Actual constraints

- Readers can read at the same time
- Writers are incompatible with everything else (writers and readers)

Is this a good specification?

```
Monitor RW {

    element_t Read()

    void Write(element_t)

}
```

Readers - Writers

Actual constraints

- Readers can read at the same time
- Writers are incompatible with everything else (writers and readers)

```
Monitor RW {
    start_read()
    end_read()

    start_write()
    end_write()
}
```

Thread reader:

```
    work()
    start_read()
    actually_read()
    end_read()
```

Readers - Writers

Actual constraints

- Readers can read at the same time
- Writers are incompatible with everything else (writers and readers)

How many processes? What are the blocking and unblocking condition?

Readers - Writers

Actual constraints

- Readers can read at the same time
- Writers are incompatible with everything else (writers and readers)

Reader unblocking condition

- A writer just finished

Writer unblocking condition

- A writer just finished

How to solve this problem?

Readers - Writers

Actual constraints

- Readers can read at the same time
- Writers are incompatible with everything else (writers and readers)

Reader unblocking condition

- A writer just finished

Writer unblocking condition

- A writer just finished and no reader is waiting

Monitors and semaphores

Semaphores implemented using Monitors

Methodology

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

Semaphores implemented using Monitors

Methodology

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

```
Monitor Semaphore {
    P()
    V()
}
```

Semaphores implemented using Monitors

Methodology

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

```
Monitor Semaphore {
    P()
    V()
}
```

- Blocking:
  - No token available
- Unblocking
  - A token has just been released

Semaphores implemented using Monitors

Methodology

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

```
Monitor Semaphore {
    P()
    V()
}
```

- Blocking:
  - No token available
- Unblocking
  - A token has just been released

Conditions: Only one behavior
State: number of tokens

Semaphores implemented using Monitors

Methodology

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

```
Monitor Semaphore {
    nb_tokens = N
    Condition cond
    P() {
        while nb_tokens == 0:
            cond.wait()
        nb_tokens--
    }
    V() {
        nb_tokens++
        cond.notify()
    }
}
```

Monitors implemented using Semaphores

Multiple elements to ensure

- Mutual exclusion of the methods

Monitors implemented using Semaphores

```
Monitor Counter {


     int c = 0

     void inc() {

          c++

     }
}
```

```
class Counter {
     semaphore entry = semaphore(1)

     int c = 0

     void inc() {
          entry.P()
          c++
          entry.V()
     }
}
```

Monitors implemented using Semaphores

Multiple elements to ensure

- Mutual exclusion of the methods
  - One global semaphore with one token
  - All methods use the semaphore at the beginning and at the end

Monitors implemented using Semaphores

Multiple elements to ensure

- Mutual exclusion of the methods
  - One global semaphore with one token
  - All methods use the semaphore at the beginning and at the end
- Management of the conditions

Monitors implemented using Semaphores

Multiple elements to ensure

- Mutual exclusion of the methods
  - One global semaphore with one token
  - All methods use the semaphore at the beginning and at the end
- Management of the conditions
  - Need a queue for the waiting threads
  - A wait must release the semaphore
  - The thread must be put on hold

```
Class condition {
    Queue waiting = Queue(threads)

    void wait() {
        waiting.add(current_thread)
        entry.V
        current_thread.state= BLOCKED
    }

    bool empty() {
        return waiting.is_empty()
    }
```

Monitors implemented using Semaphores

Multiple elements to ensure

- Mutual exclusion of the methods
  - One global semaphore with one token
  - All methods use the semaphore at the beginning and at the end
- Management of the conditions
  - Need a queue for the waiting threads
  - A wait must release the semaphore
  - The thread must be put on hold

```
void notify_continue() {
     if not waiting.empty:
          thread t = waiting.pop()
          t.state = WAITING
}
void signal() {
     if not waiting.empty:
          lock.add(current_thread)
          current_thread.state= BLOCKED
          t = waiting.pop()
          t.state = RUNNING
}
```

Monitors implemented using Semaphores

Multiple elements to ensure

- Mutual exclusion of the methods
  - One global semaphore with one token
  - All methods use the semaphore at the beginning and at the end
- Management of the conditions
  - Need a queue for the waiting threads
  - A wait must release the semaphore
  - The thread must be put on hold

How to modify this to take into account priorities?

```
void notify_continue() {
     if not waiting.empty:
          thread t = waiting.pop()
          t.state = WAITING
}
void signal() {
     if not waiting.empty:
          lock.add(current_thread)
          current_thread.state= BLOCKED
          t = waiting.pop()
          t.state = RUNNING
}
```

Monitors and semaphores

From a theoretical point of view:

- Monitors and semaphores are theoretically equivalent

From a practical point of view:

- Semaphores are easier to implement directly
- Monitors are more expressive and easy to use
- Re-implementing semaphores using monitors is overkill !

Takeaway on monitors

Nested monitor calls

What is the behavior if a method in a monitor calls a method in another monitor?

Multiple approaches are possible:

1. Remove the capability
2. Release lock on first before taking the lock on the second
3. Keep both locks during the call
    a. If waiting in the second release both locks
    b. If waiting in the second release only the second lock

Nested monitor calls

What is the behavior if a method in a monitor calls a method in another monitor?

Multiple approaches are possible:

1. Remove the capability
2. Release lock on first before taking the lock on the second
3. Keep both locks during the call
   a. If waiting in the second release both locks
   b. If waiting in the second release only the second lock

3. leads to deadlocks, particularly 3.b.

Example: Java uses 3.b.

Pros & Cons of monitors

Pros

- Structured approach for concurrent programming
- Object-like approach
- High level of abstraction
- Separation of concerns
  - Mutual exclusion is implicitly managed by the monitor
  - Conditions is the explicit way to describe the synchronization

Cons

- Higher overhead
  - Performance, memory, invasiveness in the operating system
- Multiple `signal/notify` semantics
- In complex systems, nested call can occur

Monitors in python

Monitors:

- Monitors do not exist in python !
- But Mutex and Conditions exist !

Processes / Threads

- Threads do not exist in python3

Shared variables:

- Shared variable must be explicit
- Access to values is done through `value`

```
from multiprocessing import Process, Lock,
Condition, Value

lock = Lock()
cond_walker = Condition(lock)

door = Value('i', 0)

def walker():
     with lock:
          while door.value == 0:
               cond_walker.wait()
          door.value += 1
          cond_closer.notify()

for i in range(4):
     Process(target=walker).start()
```

Complex example

Model of a tennis court

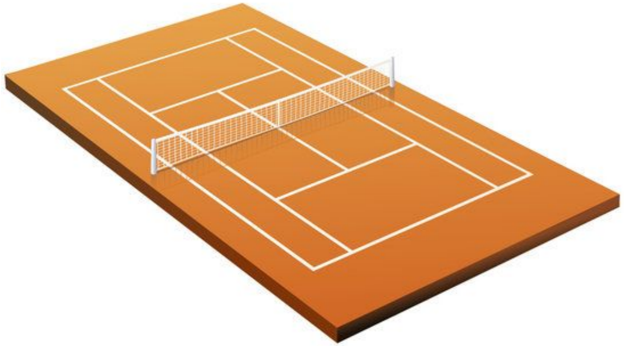
Multiple processes

- Players
  - Players do not choose with who to play
- Referees
  - Referees do not choose their game

First version: only one court, training (no referee), a game requires exactly 2 players.

```
Monitor Court {
    Play()
}
```

What is your opinion on this specification?



Model of a tennis court

Multiple processes

- Players
  - Players do not choose with who to play
- Referees
  - Referees do not choose their game

First version: only one court, training (no referee), a game requires exactly 2 players.

What are the blocking/unblocking conditions?

```
Monitor Court {
     AskCourt()
     FreeCourt()
}
```

Model of a tennis court

Multiple processes

- Players
  - Players do not choose with who to play
- Referees
  - Referees do not choose their game

First version: only one court, training (no referee), a game requires exactly 2 players.

Conditions and variables?

```
Monitor Court {
    AskCourt()
    FreeCourt()
}
```

- Blocking:
  - No player waiting on the court **or** two players already playing
- Unblocking
  - On the court and another player just arrived **or** (a game just finished and the last player left) **or** outside of the court and a player just entered the court

Model of a tennis court

Multiple processes

- Players
  - Players do not choose with who to play
- Referees
  - Referees do not choose their game

First version: only one court, training (no referee), a game requires exactly 2 players.

Code Python?

Model of a tennis court (Mesa/Python version)

```
from multiprocessing import Process, Lock,
Condition, Value

### Monitor start
lock = Lock()
court = Condition(lock)
outside = Condition(lock)
nb_players = Value('i', 0)

def askCourt():
      with lock:
            while nb_players.value == 2:
                  outside.wait()
            nb_players.value += 1
            print("On the court")
            if nb_players.value == 1:
                  outside.notify()
                  court.wait()
            else:
                  court.notify()

def freeCourt():
      with lock:
            nb_players.value -= 1
            if nb_players.value == 0:
                  outside.notify()
#### Monitor end

def player():
      print("waiting")
      askCourt()
      print("playing")
      freeCourt()
      print("finished")

for i in range(10):
      Process(target=player).start()
```

Model of a tennis court

Multiple processes

- Players
  - Players do not choose with who to play
- Referees
  - Referees do not choose their game

First version: only one court, training (no referee), a game requires exactly 2 players.

Second version: 1 referee is needed

Third version: multiple courts