# Concurrent programming

## Concurrency:

simultaneous process sharing ressources
$\Rightarrow$ mutual exclusion
$\Rightarrow$ synchronisation

## with Objective Caml

## Three modules

- `Thread`: to create, run and stop process involved in concurrent applications
- `Mutex`: to create, lock and release critical sections
- `Condition`: to create, wait and send synchronisation signals

Additonnal module

- `ThreadUnix`: non blocking Unix I/O

# Threads

"multiple threads of control (also called lightweight processes) that execute concurrently in the same memory space"

Creation: `val create : ('a -> 'b) -> 'a -> t`

```
Thread.create f x
```

1. creates a new thread to excecute `(f x)` *concurrently* with the other threads of the program.
   Note: "the program" itself is a thread.
2. returns the handle (`Thread.t`) of the created thread.
3. terminates when `(f x)` returns (or fails)
4. the result ofOCtext(f x) (or its failure) is discarded and not directly accessible to the parent thread (the one who created)

Suspend: `val delay : float -> unit`

```
Thread.delay d
```

1. suspends the execution of the calling thread for d seconds.

# Threads

## Let's play with

File `pingpong.ml`

```
let ping t =
  for i=0 to 10 do
    print_string "ping";
    flush stdout;
    Thread.delay t
  done ;;
let pong t =
  for i=0 to 10 do
    print_string "PONG";
    flush stdout;
    Thread.delay t
  done ;;

print_endline"ping-pong go:";
Thread.create ping 0.1;
Thread.create pong 0.05;
Thread.delay 3.0;
print_newline()
```

Threads are not in the standard library

```
ocamlc -thread -custom -o pingpong \
       unix.cma threads.cma pingpong.ml \
       -cclib -lunix -cclib -lthreads
```

# Let's play with threads

Run ping-pong game:

```
[unix-prompt] ./pingpong
ping-pong go:
pingPONGPONGpingPONGPONGpingPONGPONGpingPONGPONG
pingPONGpingPONGPONGpingpingpingpingping
[unix-prompt]
```

Delays:

- in `ping` ot `pong`, allow alternation
- in main expression, leave time for threads to execute

Changing delay parameters

```
Thread.create ping 0.01;
Thread.create pong 0.05;
```

changes the ditribution

```
[unix-prompt] ./pingpong
ping-pong go:
pingPONGpingpingpingPONGpingpingpingPONGpingping
pingPONGpingPONGPONGPONGPONGPONGPONGPONG
[unix-prompt]
```

# Mutual exclusion

## Critical section:

A piece of code that must not be interrupted
$\Rightarrow$ locks

## Module `Mutex`:

val create : unit -> t

*Return a new mutex.*

val lock : t -> unit

*Lock the given mutex. Only one thread can have the mutex locked at any time. A thread that attempts to lock a mutex already locked by another thread will suspend until the other thread unlocks the mutex.*

val unlock : t -> unit

*Unlock the given mutex. Other threads suspended trying to lock the mutex will restart.*

# Let's play with

Stamming players

```
let m = Mutex.create () ;;

let f s =
  for i=0 to 5 do
    Mutex.lock m;    (* begin critical section *)
    print_string s;
    Thread.delay 0.1;
    print_string s;
    flush stdout;
    Mutex.unlock m;  (* end critical section   *)
    Thread.delay (Random.float 0.3)
  done ;;

print_endline"ping-pong go:";
Thread.create f "ping";
Thread.create f "PONG";
Thread.delay 3.0;
print_newline()
```

Delays:

- between printing should allow the other thread to play but it will not, because of mutex
- randomized to introduce some perturbation in alternation

# Stamming play

Let's run

```
ping-pong go:
pingpingPONGPONGpingpingPONGPONGPONGPONGpingping
PONGPONGpingpingPONGPONGPONGPONGpingpingpingping
```

Note that `ping` and `PONG` are always displayed twice

Changing loop's body by adding one more display

```
    Mutex.lock m;    (* begin critical section *)
    print_string s;
    Thread.delay 0.1;
    print_string s;
    print_string s;
    flush stdout;
    Mutex.unlock m;  (* end critical section   *)
```

will give laternation of three consecutive `ping` and `PONG`

```
ping-pong go:
pingpingpingPONGPONGPONGpingpingpingPONGPONGPONG
PONGPONGPONGpingpingpingPONGPONGPONGpingpingping
PONGPONGPONGPONGPONGPONGpingpingpingpingpingping
```

# Waiting for a given condition

## Alternation on a boolean flag

- `ping` plays when flag is `true` and set it to `false`
- `pong` plays when flag is `false` and set it to `true`

## Wait and signal: module `Condition`

val create : unit -> t

*Return a new condition variable.*

val wait : t -> Mutex.t -> unit

`wait c m` *atomically unlocks the mutex* `m` *and suspends the calling process on the condition variable* `c`*. The process will restart after the condition variable* `c` *has been signalled. The mutex* `m` *is locked again before wait returns.*

val signal : t -> unit

`signal c` *restarts one of the processes waiting on the condition variable* `c`*.*

# Using conditions

## Fair and safe alternation

```
let m = Mutex.create () ;;
let c = Condition.create () ;;
let b = ref true ;;

let f (wait, s) =
  for i=0 to 10 do
    while wait () do Condition.wait c m done;
    print_string s; flush stdout;
    b := not !b;
    Condition.signal c;
    Mutex.unlock m;
  done ;;

print_endline"ping-pong go:";
Thread.create f ((fun () -> not !b), "ping");
Thread.create f ((fun () -> !b), "PONG");
Thread.delay 1.0;
print_newline()
```

Note: the mutex `c` is used both

- to protect the signal variable `c`
- to protect the modification of the flag `b`

# Using conditions (continued)

## Unfair but safe alternation

`ping` will play twice more than `pong`

Use an integer flag instead of a boolean

- `ping` plays when flag is more than zero, set substract 1 from the flag and do it one more
- `pong` plays when the flag is null and set it to 2

Partial code

```
[..]
let n = ref 2 ;;
let ping () =
  for i=1 to 10 do
    while !n = 0 do Condition.wait c m done;
    print_string "ping"; flush stdout;
    n := !n-1;
    Condition.signal c; Mutex.unlock m
  done ;;
let pong () =
  for i=1 to 5 do
    while !n > 0 do Condition.wait c m done;
    print_string "PONG"; flush stdout;
    n := 2;
    Condition.signal c; Mutex.unlock m
  done ;;
[..]
```