

# PC3R

## Cours 03 - Passage de Message - Canaux

Romain Demangeon

PC3R MU1IN507 - STL S2

07/02/2022

# Plan du Cours 3

- ▶ Concurrence par passage de message.
- ▶ Canaux en *Go*.
- ▶ Canaux et évènements en *OCaml*.

# Jusqu'ici: Mémoire Partagée

- ▶ programmation à l'aide de plusieurs processus (concrètement, des threads) qui s'exécutent simultanément.
- ▶ la sémantique d'entrelacement décrit le comportement d'un système composite.
  - ▶ préemption: tous les entrelacements sont possibles.
  - ▶ coopération: l'ensemble des entrelacements possibles est restreint.
- ▶ la concurrence est obtenue par le partage de ressources:
  - ▶ une zone mémoire partagée est accessible aux processus
  - ▶ les communications sont réalisée par cette mémoire: lecture et écriture au même endroit.
  - ▶ la synchronisation (attente qu'une action soit réalisée par un autre processus) est réalisée:
    - ▶ par une attente active: boucle sur une condition lisant la mémoire,
    - ▶ par des mécanismes spécifiques au système (OS, langage, ...), variables de conditions ou événements.
- ▶ la programmation est rendue difficile:
  - ▶ par les nombreux écueils (compétitions, intebloquages, ...)
  - ▶ par l'exponentialisation de l'espace d'état.
  - ▶ par la complexification du code (mutex, condvar, cooperation)

# Mémoire Partagée vs. Passage de Messages

- ▶ l'utilisation de la mémoire partagée pour communiquer est **contingente**:
  - ▶ passage **naturel** depuis la programmation séquentielle
  - ▶ initialement, pas besoin de **mécanismes supplémentaires**
    - ▶ on répartit les **opérations** dans plusieurs processus.
  - ▶ au final, besoin de **mécanismes supplémentaires**.
- ▶ la communication par **messages** existe dans d'autres domaines:
  - ▶ **réseau**:
    - ▶ communications **distantes** et **asynchrones**
    - ▶ autre **problèmes**: ordre des messages, perte, ...
  - ▶ **web**, et plus généralement, modèles **client-serveur**:
    - ▶ communication **asymétriques asynchrones** suivant un **protocole** fixé à l'avance
  - ▶ **messagerie, mails, abonnement/diffusion**:
    - ▶ modèles de **haut-niveau**, construits sur d'autres systèmes (web, par exemple).
- ▶ par définition, la communication **distant**e se fait par messages.
- ▶ **messages** aussi utilisés pour des **communications locales**:
  - ▶ c'est le cas des **modèles de concurrence** par passage de messages

# Mémoire Partagée vs. Passage de Messages

- ▶ Jusqu'au années 2000, la programmation concurrente désignait principalement la mémoire partagée
  - ▶ puis, changement de paradigme (web, client-serveur locaux, ...),
- ▶ Domaines (plus ou moins) actifs en mémoire partagée:
  - ▶ mémoires transactionnelles: composition d'actions atomiques en *transactions*, implémentation de systèmes transactionnels (matériel, logiciel)
  - ▶ modèle mémoire faibles: étude du fonctionnement des processeurs modernes, création de modèles mathématiques réalistes (prise en compte du *Write Buffering*, entre autres).
- ▶ l'application de modèles par passage de messages à la programmation locale la rend plus claire.
  - ▶ "*don't communicate by sharing memory, share memory by communicating*" (motto du langage Go)
  - ▶ la difficulté reste exponentielle: interbloquages, ordre des messages, identité du récepteur, ...

# Messages

- ▶ une message  $m$  est une unité d'information qui peut être produite par un **envoi** et consommée par une **réception**.
  - ▶ un système (langage, OS) implémentant un **modèle concurrent par passage de message** doit fournir (au moins) ces **deux opérations** de base.
- ▶ le processus qui **envoi  $m$**  perd son contrôle sur le message.
  - ▶ après l'envoi, le message **ne peut pas être modifié**,
  - ▶ différence avec la communication par **mémoire partagée**,
- ▶ la communication est **directionnelle**:
  - ▶ l'information voyage de l'**émetteur** vers le **récepteur**.
- ▶ **implémentation**:
  - ▶ dans un cadre **distant**, les messages sont **encodés sur le medium**: bus, internet (TCP/IP) , web (HTTP), ...
  - ▶ dans un cadre **local**, les messages sont écrits et lus dans une **zone spécifique de la mémoire**.

# Messages Synchrones et Asynchrones

- ▶ on distingue deux types (principaux) de messages:
  - ▶ messages **synchrones**: l'envoi et la réception sont **bloquants**
    - ▶ un processus qui **envoie** reste bloqué sur l'opération d'envoi.
    - ▶ un processus qui **reçoit** reste bloqué sur l'opération de réception.
    - ▶ quand le système contient un processus bloqué sur l'envoi et un processus bloqué sur la réception, la **communication** a lieu, et les deux processus passent sont **débloqués**.
  - ▶ messages **asynchrones**: la réception seule est **bloquante**
    - ▶ un processus qui **envoie** passe directement à la suite.
    - ▶ le message est **stocké** par le système.
    - ▶ un processus qui **reçoit** reste bloqué sur l'opération de réception.
    - ▶ quand le système contient un processus bloqué sur la réception et (au moins) un message stocké, la **réception** a lieu.

# Adresse directe

- ▶ En **adresse directe**, les mécanismes du systèmes permettent d'envoyer un message **directement à un processus**.
- ▶ **envoi**  $\bar{p}(m)$ : envoie le message  $m$  au processus  $p$ .
- ▶ **réception**  $m \leftarrow$ : **pas besoin d'argument**.
- ▶ Ce paradigme se base sur un mécanisme **d'identification des processus**.
  - ▶ à la création d'un processus, son **père** (le processus qui opère la création) récupère un **identifiant**,
  - ▶ l'identifiant peut ensuite être **passé dans un message** afin de **faire connaître** le nouveau processus.
  - ▶ par exemple:  
`cree_fils()`  
 $p \leftarrow$   
`n = spawn(code_fils)`  
 $\bar{p}(n)$ 
    - ▶ **ici**, le processus qui exécute `cree_fils` reçoit l'**identifiant** d'un futur père  $p$ , il crée un fils appelé  $n$  et envoie  $n$  à  $p$ .
    - ▶ ensuite,  $p$  pourra **communiquer** avec  $n$ .

# Adresse directe: Erlang

- ▶ Langage fonctionnel pour la programmation concurrente à **adresse directe**.
  - ▶ **modèle d'acteurs**:
    - ▶ les processus réagissent à une **réception** en **envoyant** des messages, en **préparant** une prochaine réception et/ou en **créant** de nouveaux acteurs.
    - ▶ l'envoi de message est **asynchrone**.
  - ▶ **boîtes aux lettres**:
    - ▶ les messages sont stockées dans la **boîte aux lettres** du processus receveur.
    - ▶ l'ordre entre messages d'un même processus est respecté,  $p \xrightarrow{m_1} q$  puis  $p \xrightarrow{m_2} q$  garantit que  $q$  reçoit  $m_1$  avant  $m_2$ .
    - ▶ attention à l'ordre entre messages de plusieurs émetteurs: faire  $p \xrightarrow{m_1} r$  puis  $q \xrightarrow{m_2} r$  ne donne aucune garantie sur l'ordre de réception de  $m_1$  et  $m_2$ .
    - ▶ la **boîte au lettre** dispose d'un mécanisme de **reconnaissance de motif** sur **toute la boîte**.
- ▶ Gestion du **temps réel** au coeur du système
  - ▶ héritage du domaine de la **téléphonie**.

# Adresse directe: Erlang (II)

```
ping(0, Pong_PID) ->
    Pong_PID ! finished.

ping(N, Pong_PID) ->
    Pong_PID ! {ping, self()},
    receive
        pong ->
            io:format(" Ping~n", [])
    end,
    ping(N - 1, Pong_PID).

pong() ->
    receive
        finished ->
            io:format(" Fin~n", []);
        {ping, Ping_PID} ->
            io:format("Pong~n", []),
            Ping_PID ! pong,
            pong()
    end.

start() ->
    Pong_PID = spawn(fun () -> pong() end),
    spawn(fun () -> ping(3, Pong_PID) end).
```

# Programmation en adresse directe: Bilan

## ► Avantages:

- ▶ Clarté du système: on sait, **pour chaque message**, qui l'envoie (place de `p!_m` dans le code) et qui le reçoit (c'est `p`).
- ▶ Flexibilité du traitement des messages: l'adresse directe permet la manipulation de **boîte à lettres** dans laquelle **tous les messages** sont déposés.
  - ▶ on peut réagir directement à **plusieurs types** de messages,
  - ▶ les messages sont stockés **localement**.

## ► Inconvénients:

- ▶ Mobilité difficile: la **création** d'un nouveau processus oblige à communiquer **explicitement** son identifiant à tous ceux qui devront communiquer avec lui.
- ▶ Concurrence peu naturelle: pour mettre en **concurrence** plusieurs processus (par exemple, plusieurs travailleurs), on doit construire explicitement un mécanisme qui envoie une **ressource** à un **unique** destinataire.
- ▶ comme souvent, tout est affaire d'**implicite** (l'opération se fait naturellement) vs. **explicite** (l'opération demande l'utilisation d'une bibliothèque ou la programmation du mécanisme).

# Canaux

- ▶ un canal est une entité abstraite du système (langage de programmation) utilisé pour communiquer
  - ▶ les primitives du système permettent l'envoi et la réception sur un canal,
  - ▶ un canal n'est, *a priori*, pas lié à un processus en particulier.
- ▶ envoi sur un canal:  $\bar{c}\langle m \rangle$ .
- ▶ réception depuis un canal:  $c(x)$ .
- ▶ création d'un canal:  $(\nu c)$
- ▶ capacité du canal:
  - ▶ moralement, nombre de messages qui peuvent exister dans le canal,
  - ▶ capacité 0: canaux synchrones
    - ▶ envoi et réception sont bloquant.
  - ▶ capacité  $n > 0$ : canaux asynchrones
    - ▶ envoi bloquant seulement si le canal est plein.
    - ▶ réception bloquante seulement si le canal est vide.
    - ▶ préservation de l'ordre des messages dans un même canal.
  - ▶ capacité  $\infty$ : canaux entièrement asynchrones.
    - ▶ envoi jamais bloquant, "fire and forget"
    - ▶ réception bloquante seulement si le canal est vide.
- ▶ ordre supérieur: on peut transmettre un canal sur canal.
  - ▶ système de types ?

# Canaux: Avantages

## ► Avantages:

- communication **anonymes**: pas besoin de publier l'**identifiant** d'un nouveau processus, il suffit de se passer un **canal existant**.

pere( $u$ ) :

$(\nu d)$

$\overline{u}\langle d \rangle$

$d(y)$

imprime("Mon fils dit: " +  $y$ )

fils( $x$ ) :

$\overline{x}\langle$ "Bonjour papa !"  $\rangle$

createur( $u$ ) :

$u(c)$

spawn(code\_fils( $c$ ))

$(\nu ch)$

spawn(createur( $ch$ ))

spawn(pere( $ch$ ))

Ici..

- $ch$  (ou  $u$ ) est "le canal de **contact**" du créateur,
- $d$  (ou  $c$ ) est "le canal de **communication** du père avec son nouveau fils",
- des canaux sont créés à la volée et **passés sur des canaux**
- **types**:  $d : \# str$ ,  $ch : \# \# str$

# Canaux: Avantages

## ► Avantage:

### ► concurrence implicite:

- plusieurs envois pour la même réception sont en concurrence,
- plusieurs réception pour le même envoi sont en concurrence,

`client( $u, n$ ) :`

$(\nu c)$   
 $\overline{u}\langle n, c \rangle$   
 $c(x)$   
`imprime( $x$ )`

`serv( $u$ ) :`

$u(y, z)$   
 $\overline{z}\langle "Gagné" + y \rangle$   
 $u(t, s)$   
 $\overline{s}\langle "Perdu" \rangle$

$(\nu ch)$

`spawn(serv(ch))`  
`spawn(client(ch, 1))`  
`spawn(client(ch, 2))`

- $ch$  (ou  $u$ ) est "le canal de contact" du serveur,
- deux comportements possibles.
- types:

# Canaux: Avantages

## ► Avantage:

### ► concurrence implicite:

- plusieurs **envois** pour la même **réception** sont en concurrence,
- plusieurs **réception** pour le même **envoi** sont en concurrence,

`client( $u, n$ ) :`

$(\nu c)$   
 $\overline{u}\langle n, c \rangle$   
 $c(x)$   
`imprime( $x$ )`

`serv( $u$ ) :`

$u(y, z)$   
 $\overline{z}\langle "Gagné" + y \rangle$   
 $u(t, s)$   
 $\overline{s}\langle "Perdu" \rangle$

$(\nu ch)$

`spawn(serv(ch))`  
`spawn(client(ch, 1))`  
`spawn(client(ch, 2))`

- $ch$  (ou  $u$ ) est "le canal de **contact**" du serveur,
- deux **comportements** possibles.
- **types**:  $ch : \#(\text{int}, \# \text{str})$

# Canaux: Concurrence

- ▶ Design d'un **serveur**:

- ▶ première idée:

**serveur**( $c_1, c_2$ ) :

loop

$c_1(x)$

$r := f(x)$

$\overline{c_2}\langle r \rangle$

**client**( $c_1, c_2, m$ ) :

$\overline{c_1}\langle m \rangle$

$c_2(y)$

$g(y)$

$(\nu in, out)$

spawn(**serveur**( $in, out$ )) :

spawn(**client**( $in, out, v_1$ )) :

spawn(**client**( $in, out, v_2$ )) :

- ▶ Problème:

# Canaux: Concurrence

- ▶ Design d'un **serveur**:

- ▶ première idée:

**serveur**( $c_1, c_2$ ) :

loop

$c_1(x)$

$r := f(x)$

$\overline{c_2}\langle r \rangle$

**client**( $c_1, c_2, m$ ) :

$\overline{c_1}\langle m \rangle$

$c_2(y)$

$g(y)$

$(\nu in, out)$

spawn(**serveur**( $in, out$ )) :

spawn(**client**( $in, out, v_1$ )) :

spawn(**client**( $in, out, v_2$ )) :

- ▶ **Problème**: le serveur n'accepte pas de requêtes concurrentes.

- ▶ une seule requête est traitée à la fois, on souhaiterait que le serveur redevienne disponible **directement après acceptation**.

# Canaux: Concurrence (II)

- ▶ Design d'un **serveur**:
  - ▶ serveur **concurrent**:

**code\_serveur**( $c, z$ ) :

$r := f(z)$   
 $\bar{c}\langle r \rangle$

**serveur**( $c_1, c_2$ ) :

loop  
 $c_1(x)$   
spawn(**code\_serveur**( $c_2, x$ ))

**client**( $c_1, c_2, m$ ) :

$\bar{c}_1\langle m \rangle$   
 $c_2(y)$   
 $g(y)$

( $\nu in, out$ )

spawn(**serveur**( $in, out$ )) :  
spawn(**client**( $in, out, v_1$ )) :  
spawn(**client**( $in, out, v_2$ )) :

- ▶ à chaque **requête**, un sous-processus du serveur est créé pour la gérer. Plusieurs **requêtes** peuvent être traitées simultanément.
- ▶ **Problème**:

# Canaux: Concurrence (II)

- ▶ Design d'un **serveur**:
  - ▶ serveur **concurrent**:

`code_serveur( $c, z$ ) :`

$r := f(z)$

$\overline{c}\langle r \rangle$

`serveur( $c_1, c_2$ ) :`

`loop`

$c_1(x)$

`spawn(code_serveur( $c_2, x$ ))`

`client( $c_1, c_2, m$ ) :`

$\overline{c_1}\langle m \rangle$

$c_2(y)$

$g(y)$

$(\nu in, out)$

`spawn(serveur( $in, out$ )) :`

`spawn(client( $in, out, v_1$ )) :`

`spawn(client( $in, out, v_2$ )) :`

- ▶ à chaque **requête**, un sous-processus du serveur est créé pour la gérer. Plusieurs **requêtes** peuvent être traitées simultanément.
- ▶ **Problème:** les requêtes peuvent s'emmêler.

- ▶ le serveur lance **en parallèle** le code traitant les deux requêtes, et les réponses sont envoyées sur **le même canal**.
- ▶ le premier client peut recevoir  $f(v_2)$ .
- ▶ il faut un mécanisme qui **garantit** la **communication privée**.

# Canaux: Concurrence (III)

- ▶ Design d'un **serveur**:

- ▶ serveur concurrent **sûr**:

code\_serveur( $c, z$ ) :

$r := f(z)$

$\bar{c}\langle r \rangle$

serveur( $c_1$ ) :

loop

$c_1(x, c_2)$

spawn(code\_serveur( $c_2, x$ ))

client( $c_1, m$ ) :

$(\nu d)$

$\overline{c_1}\langle m, d \rangle$

$d(y)$

$g(y)$

$(\nu in, out)$

spawn(serveur( $in, out$ )) :

spawn(client( $in, out, v_1$ )) :

spawn(client( $in, out, v_2$ )) :

- ▶ chaque client crée un canal **privé**

- ▶ à la création, un canal n'est connu **que de son créateur**.

- ▶ **mobilité**: le client passe son canal au serveur **dans la requête**,

- ▶ le serveur **répond** sur le canal reçu.

- ▶ **transformation de types**:  $A \rightarrow B$  devient  $\sharp(A, \sharp B)$ .

# Canaux: Services Récursifs

```
add(a, x, y, r) :  
  if x = 0  
     $\bar{r}\langle y \rangle$   
  else  
    ( $\nu c$ )  
     $\bar{a}\langle x - 1, y, c \rangle$   
    c(z)  
    res := z + 1  
     $\bar{r}\langle res \rangle$ 
```

```
add_serveur(u) :  
  loop  
    u(x, y, r)  
    spawn(add(u, x, y, r))
```

```
( $\nu add, mult$ )  
spawn(add_serveur(add))  
spawn(mult_serveur(mult, add))
```

```
mult(m, a, x, y, r) :  
  if x = 0  
     $\bar{r}\langle 0 \rangle$   
  else  
    ( $\nu c_1, c_2$ )  
     $\bar{m}\langle x - 1, y, c_1 \rangle$   
    c_1(z_1)  
     $\bar{a}\langle y, z_1, c_2 \rangle$   
    c_2(z_2)  
     $\bar{r}\langle z_2 \rangle$ 
```

```
mult_serveur(u, a) :  
  loop  
    u(x, y, r)  
    spawn(mult(u, a, x, y, r))
```

- ▶ réseau de services: une requête au serveur de multiplication déclenche une requête au serveur d'addition.
- ▶ services récursifs: une requête au serveur d'addition déclenche une autre requête à ce même serveur.

# Canaux: Polyadicité

- ▶ On veut très souvent envoyer (recevoir) plusieurs choses à la fois (un  $p$ -uplet de valeurs):
  - ▶ par exemple, une valeur et un canal de retour.
- ▶ parfois, l'implémentation ne permet que les canaux monadiques.
  - ▶ souvent, on peut envoyer une structure.
- ▶ On ne peut pas utiliser deux canaux publics:
  - ▶ par exemple,
$$\begin{array}{c|c} \overline{c_1}\langle m_1 \rangle & c_1(x) \\ \overline{c_2}\langle m_2 \rangle & c_2(y) \end{array}$$
  - ▶ si plusieurs processus envoient et écoutent en même temps: risque d'emmêler les paires.
- ▶ On peut utiliser un canal privé:
  - ▶ par exemple,
$$\begin{array}{c|c} (\nu d) & \\ \overline{c}\langle d \rangle & c(z) \\ \overline{d}\langle m_1 \rangle & z(x) \\ \overline{d}\langle m_2 \rangle & z(y) \end{array}$$
  - ▶ envoi sûr.

## Canaux: Asynchronie

- ▶ si on dispose de canaux totalement asynchrones, on peut quand même encoder l'[envoi bloquant](#)

- par exemple dans:

$$\begin{array}{c|c} \overline{c}\langle m \rangle & c(x) \\ \text{suite\_envoi}() & \text{suite\_reception}(x) \end{array}$$

- ▶ `suite_envoi` peut être exécuté avant la réception de  $m$  (asynchronie).

- on l'encode en:

$(\nu d)$	
$\overline{c}\langle m, d \rangle$	$c(x, y)$
$d()$	$\overline{y} \langle \rangle$
suite_envoi()	suite_reception( $x$ )

- ici, on est sûr que  $m$  a été reçu quand on exécute `suite_envoi`.

# Canaux: Bilan

- ▶ grande **expressivité**: les messages permettent d'encoder des mécanismes de **communication** ou de **synchronisation** complexes.
- ▶ **mobilité**: **création dynamique** de canaux et leur diffusion grâce à l'**ordre supérieur**
- ▶ **facilité** d'écriture de systèmes concurrents: la **communication** est **primitive**.
- ▶ **courant** dans la programmation contemporaine:
  - ▶ canaux de *Go*, **mécanisme premier** de concurrence,
  - ▶ canaux synchrones d'*OCaml*, modèle **typé** de communication par passage de message, encapsulée dans les **événements**,
  - ▶ canaux **dirigés, synchrones/asynchrones**, de *Rust*
  - ▶ canaux de *Promela* pour la **vérification**
  - ▶ **Message-Oriented Middleware**: par exemple l'interface *MessageConsumer* de *Java Message Service*, file de messages pour la communication entre applications.

# Go: Langage

- ▶ Langage
  - ▶ compilé,
  - ▶ multiparadigme: syntaxe impérative, fonctions comme citoyens de première classe, héritage implicite entre interfaces et polymorphisme de rangée.
  - ▶ facile d'accès: compromis entre l'accessibilité (*Javascript*) et la sûreté du code produit (*Rust*) ; syntaxe proche de *Pascal*, *C*, ... avec des particularités
  - ▶ concurrent: développé dans le but d'être utilisé pour des applications concurrentes, mécanismes primitifs de manipulation de processus (*goroutines*)
  - ▶ par passage de message: mécanismes primitifs de manipulation de canaux, désignés comme *la bonne manière de programmer*
  - ▶ efficace: légèreté des goroutines, directement développé pour l'usage des processeurs multi-coeurs.
  - ▶ typage fort, statique inféré (annotations possibles), structurel (pour les interfaces).
- ▶ Utilisations:
  - ▶ applications systèmes: *Docker*, *Netflix*, *dl.google* ...
  - ▶ remplace Python: *Dropbox*, *Twitch*, ...

# Go: Syntaxe séquentielle

- ▶ **variables**: := initialisation, = affectation, inférence de types

```
func f() {  
    var s1 string = "saucisse"  
    s2 := "aspirateur"  
    fmt.Println(s1 + s2)  
}
```

- ▶ **boucles**: for est while

```
func f() {  
    continu := true  
    c := 1024  
    for continu {  
        c = c / 2  
        if c == 0 {  
            continu = false  
        }  
    }  
}
```

# Go: Syntaxe séquentielle (II)

- ▶ boucles: `for` est `for`

```
func f() {
    for i := 0; i < 100; i++ {
        fmt.Println("tour", i)
    }
}
```

- ▶ structures: enregistrements

```
type paquet struct {
    arrivee string
    depart   string
    arret    int
}

func f() {
    p := paquet{arrivee : "lun", depart : "jeu", arret : 180}
    ...
}
```

# Go: Syntaxe séquentielle (III)

## ► interfaces:

```
type annuaire interface {
    enregistre(service func(i int) int, nom string)
    cherche(nom string) bool
    trouve(nom string) func(i int) int
}
```

## ► fermetures dynamiques:

```
func imprime(i int) {
    fmt.Println("valeur", i)
}

func f() {
    table := [100]func(){}
    for i := 0; i < 100; i++ {
        table[i] = func() { imprime(i) }
    }
    for i := 0; i < 100; i++ {
        table[i]()
    }
}
```

# Go: Goroutines

- ▶ Modèle concurrent: un **thread système** par cœur du processeur, les goroutines sont réparties sur les **threads systèmes**.
- ▶ l'ordonnancement est (plutôt) **coopératif**: une goroutine n'est préemptée que si:
  - ▶ elle atteint une **primitive de message**,
  - ▶ elle **appelle une fonction**,
  - ▶ elle fait une **entrée/sortie**,
  - ▶ elle **dure** trop longtemps.
- ▶ elles sont créées avec une **petite pile** (quelques kilooctets) de taille **variable**.
- ▶ elles sont manipulées à la volée par l'**environnement d'exécution** de Go et **redistribuées** sur les threads systèmes si nécessaire (entrée/sortie bloquantes).
- ▶ plusieurs **dizaines de milliers** de coroutines peuvent coexister dans une application standard.

**Objectif:** **Transparence** pour le programmeur: l'environnement d'exécution s'occupe de l'efficacité.

## Go: Goroutines (II)

- ▶ le mot-clef go **crée** une nouvelle goroutine.

```
for i := 0; i < 100; i++ {  
    go imprime(i)  
}
```

- ▶ il est d'**usage** de passer une **application de fonction anonyme** à go

```
func f() {  
    for i := 0; i < 100; i++ {  
        go func() { imprime(i) }()  
    }  
}
```

# Go: Goroutines (II)

- ▶ le mot-clef go **crée** une nouvelle goroutine.

```
for i := 0; i < 100; i++ {  
    go imprime(i)  
}
```

- ▶ il est d'**usage** de passer une **application de fonction anonyme** à go

```
func f() {  
    for i := 0; i < 100; i++ {  
        go func() { imprime(i) }()  
    }  
}
```

- ▶ **attention** aux fermetures dynamiques !

```
func f(fini chan int) {  
    for i := 0; i < 100; i++ {  
        go func(k int) { imprime(k) }(i)  
    }  
}
```

# Go: Messages

- ▶ les canaux synchrones sont primitifs:

```
c := new(chan int)
d := new(chan chan int)
```

- ▶ les canaux asynchrones aussi:

```
c2 := new(chan int, 10)
```

- ▶ envoi:

```
d <- c
c <- 42
```

- ▶ réception (on peut jeter la valeur reçue):

```
x := <- d
<- x
```

# Go: Messages

- ▶ les canaux synchrones sont primitifs:

```
c := new(chan int)
d := new(chan chan int)
```

- ▶ les canaux asynchrones aussi:

```
c2 := new(chan int, 10)
```

- ▶ envoi:

```
d <- c
c <- 42
```

- ▶ réception (on peut jeter la valeur reçue):

```
x := <- d
<- x
```

- ▶ envoi asynchrone:

```
go func (){<- x}()
```

## Go: Messages (II)

- ▶ les canaux partagent de l'information

```
x := <- d  
x <- "saucisse"
```

- ▶ les canaux servent de mécanismes de synchronisation.

```
func imprime(i int, f chan int) {  
    fmt.Println("tour", i)  
    f <- 0  
}  
  
func f(fini chan int) {  
    f := make(chan int)  
    for i := 0; i < 100; i++ {  
        go func(k int) { imprime(k, f) }(i)  
    }  
    for i := 0; i < 100; i++ {  
        <-f  
    }  
    fini <- 0  
}  
  
func main() {  
    fin := make(chan int)  
    go f(fin)  
    <-fin  
}
```

## Go: Serveur

```
type requete struct {
    arg      int
    retour   chan int
}

func client(u chan requete, id int, f chan int) {
    n := rand.Intn(100)
    r := make(chan int)
    u <- requete{arg: n, retour: r}
    res := <-r
    fmt.Println("envoyé:", n, "reçu:", res)
    f <- 0
}
```

- ▶ **structure** pour encoder la paire (valeur, canal de retour).
- ▶ création d'un canal **privé** `r`.

## Go: Serveur (II)

```
func serveur(u chan requete) {
    for {
        req := <-u
        go func(r requete) {
            res := r.arg * r.arg
            r.retour <- res
        }(req)
    }
}

func main() {
    url := make(chan requete)
    fin := make(chan int)
    go func() { client(url, 1, fin) }()
    go func() { client(url, 2, fin) }()
    go func() { serveur(url) }()
    <-fin
    <-fin
}
```

- ▶ les résultats ne s'emmêle pas.
- ▶ l'exécution est déterministe.

## Go: Sélection

- ▶ la primitive `select` permet de proposer plusieurs comportements liés à des opérations de synchronisation différentes.

```
select {
    case n := <- c1 :
        f()
    case c2 <- 42 :
        go g()
    case <-fin :
        return
}
```

- ▶ les opérations peuvent être différentes: envoi / réception de types différents.
- ▶ une unique opération, parmi celles disponibles est choisie (non déterministiquement).
- ▶ si aucune opération n'est disponible, `select` est bloquant (jusqu'à ce que l'une d'elles le devienne).

# Go: Polymorphismes

```
func client(u chan chan interface{}, id int, f chan int) {  
    n := rand.Intn(100)  
    l := make(chan interface{})  
    r := make(chan int)  
    u <- l  
    l <- n  
    l <- r  
    res := <-r  
    fmt.Println("envoyé:", n, "reçu:", res)  
    f <- 0  
}
```

- ▶ polymorphisme d'interface:
  - ▶ plusieurs interfaces plus précises peuvent instancier une interface plus grossière.
  - ▶ `interface{}` est l'interface la plus grossière.
- ▶ le polymorphisme s'applique aux canaux.

# Go: Polymorphismes (II)

```
func serveur(u chan chan interface{}) {
    for {
        cli := (<-u)
        go func(cl chan chan interface{}) {
            r := (<-cl).(int)
            ret := (<-cl).(chan int)
            res := r * r
            ret <- res
        }(cli)
    }
}

func main() {
    url := make(chan chan interface{})
    fin := make(chan int)
    go func() { client(url, 1, fin) }()
    go func() { client(url, 2, fin) }()
    go func() { serveur(url) }()
    <-fin
    <-fin
}
```

- ▶ **assertion** de type `exp.(T)` nécessaire pour diriger l'utilisation.
- ▶ **limites** de l'analyse statique: *mismatch* de types à l'`exécution`

# *OCaml*: Concurrence

- ▶ *OCaml* possède des mécanismes élégants (fortement typé, encapsulé dans des modules) de programmation concurrente:
  - ▶ threads préemptifs avec mutexes et variables de conditions,
  - ▶ concurrence asynchrone Async,
  - ▶ monade de concurrence coopérative Lwt,
  - ▶ événements et canaux synchrones Event.
- ▶ *OCaml* ne possède pas de modèle concurrent efficace:
  - ▶ l'environnement d'exécution utilise un verrou global sur les threads systèmes créés par le programme,
- ▶ Ca reste un bon langage pédagogique pour la concurrence
- ▶ *Multicore OCaml* par *OCaml Labs* devrait bientôt arriver (2021?).

# OCaml: Canaux synchrones

- ▶ le module Event offre des mécanismes de communication et synchronisation par évènements.
- ▶ les canaux sont des objets manipulables, avec un type spécifique,
- ▶ les évènements sont des objets manipulables, avec un type spécifique,
- ▶ les opérations élémentaires sur les canaux (envoi/réception) renvoient des évènements.
- ▶ une primitive de synchronisation permet d'attendre qu'un évènement se produise.
- ▶ les canaux sont fortement et statiquement typés
  - ▶ pas de polymorphismes douteux avec chan interface{}

# OCaml: Événements

- ▶ deux types abstraits : `'a channel` et `'a event`
- ▶ `new_channel` : `unit -> 'a channel` : création d'un canal
- ▶ `send` : `'a channel -> 'a -> unit event` : envoi
  - ▶ renvoie un évènement de type `unit`.
- ▶ `receive` : `'a channel -> 'a event`: réception
- ▶ ni `send` ni `receive` ne sont bloquants.
  - ▶ ils renvoient un évènement qu'on peut utiliser directement (le passer à une fonction, par exemple)
- ▶ `sync` : `'a event -> 'a` : fonction de synchronisation
  - ▶ bloquante jusqu'à ce que l'évènement se produise.

# OCaml: Événements (exemple)

```
let ch = Event.new_channel () ;;
let v = ref 0;;

let reader () = Event.sync (Event.receive ch);;
let writer () = Event.sync (Event.send ch ("S" ^ (string_of_int !v)));;

let loop_reader s d () =
  for i=1 to 10 do
    let r = reader() in
    print_string (s ^ " " ^ r); print_newline();
    Thread.delay d
  done ;;

let loop_writer d () =
  for i=1 to 10 do incr v; writer(); Thread.delay d
done ;;

Thread.create (loop_reader "A" 1.1) ();
Thread.create (loop_reader "B" 1.5) ();
Thread.create (loop_reader "C" 1.9) ();
Thread.delay 2.0;;
loop_writer 1. ();
```

- ▶ souvent, on **enchaîne** la synchronisation aux opérations d'envoi/réception.

# OCaml: Enveloppe

- ▶ `poll: 'a event -> 'a option`: non bloquant, retourne `Some v` si un événement l'évènement est disponible, `None` sinon,
- ▶ `always : 'a -> 'a event`: crée un évènement disponible
  - ▶ c'est un *return*
- ▶ `wrap : 'a event -> ('a -> 'b) -> 'b event`: enveloppe l'évènement d'un futur.
  - ▶ `wrap f e` est disponible quand `e` est disponible de valeur `r` et `f r` est disponible,
  - ▶ la valeur de `wrap f e` est la valeur de `f r` si `r` est la valeur de `e`.
  - ▶ c'est un *foncteur*

# OCaml: Enveloppe

- ▶ `poll: 'a event -> 'a option`: non bloquant, retourne `Some v` si un événement l'évènement est disponible, `None` sinon,
- ▶ `always : 'a -> 'a event`: crée un évènement disponible
  - ▶ c'est un *return*
- ▶ `wrap : 'a event -> ('a -> 'b) -> 'b event`: enveloppe l'évènement d'un futur.
  - ▶ `wrap f e` est disponible quand `e` est disponible de valeur `r` et `f r` est disponible,
  - ▶ la valeur de `wrap f e` est la valeur de `f r` si `r` est la valeur de `e`.
  - ▶ c'est un *foncteur*
  - ▶ du coup on pourrait écrire:

```
bind : 'a event -> ('a -> 'b event) -> 'b event
let bind e f = wrap (fun x -> Event.Sync (f x)) e
```

# OCaml: Enveloppe (exemple)

- ▶ wrap est particulièrement utile avec les primitives de choix:
  - ▶ choose: 'a event list -> 'a event
    - ▶ renvoie un évènement correspondant au premier évènement disponible de la liste.
  - ▶ select: 'a event list -> 'a
    - ▶ c'est choose + sync

```
let rec accum sum =
  print_int sum; print_newline ();
  Event.sync (
    Event.choose [
      wrap (receive addCh) (fun x -> accum(sum + x));
      wrap (receive subCh) ( fun x -> accum(sum - x));
      wrap (send readCh sum) ( fun x -> accum(sum))
    ]
  )
```

# Exercice CS de l'examen PC2R de 2016

- ▶ table d'association clefs (chaînes de caractères) / valeurs (entiers) partagée
- ▶ implémentation séquentielle:

```
type 'a option = Some of 'a | None
type assoc
cr_assoc : unit -> assoc
setv: assoc * string * int -> assoc
getv: assoc * string -> int option
```

Un thread "serveur" `table_p` qui maintient une `assoc` et est joignable sur deux canaux publics `s` (mise à jour) et `g` (récupération). Les clients ont un canal personnel (différent), et connaissent `s` et `g` du serveur:

- ▶ pour mettre à jour, ils envoient sur `s` une paire (`k`, `v`) à inclure dans la table ;
- ▶ pour récupérer une valeur, ils envoient sur `g` une paire (`cp`, `k`), canal personnel et clef de la table. Ils attendent ensuite sur `cp` qu'on leur envoie la valeur entière de l'association avec `k`. Si la clef n'existe pas dans la table, ils attendent indéfiniment.

# Exercice CS de l'examen PC2R de 2016 (II)

```
type assoc = string * int list
type 'a option = Some of 'a | None

let cr_assoc () = []

let setv ass k v = (k,v)::ass

let rec getv ass k = match ass with
  [] -> None
  | (k',v)::q -> if (k = k') then Some(v) else (getv q k)

let g = Event.new_channel ()
let s = Event.new_channel ()
let c2 = Event.new_channel ()
```

- ▶ en-tête du système
  - ▶ code de la [table d'association](#),
  - ▶ création des [canaux](#),

# Exercice CS de l'examen PC2R de 2016 (III)

```
let rec table_p ass =
  let conts (k, v) =
    (print_endline "Serveur: Mise a jour.");(table_p (setv ass k v))
  in
  let contg (cp, k) =
    match (getv ass k) with
      None -> (print_endline "Serveur: Pas trouve.");(table_p ass)
    | Some(v) ->
        let _ = (Event.sync (Event.send cp v)) in
        (print_endline "Serveur: Trouve.");(table_p ass)
  in
  Event.select [(Event.wrap (Event.receive s) conts);
                (Event.wrap (Event.receive g) contg)]
```

► code du **serveur**:

- ▶ il se **passe récursivement** une table ass,
- ▶ il **sélectionne sur les réceptions des deux canaux**,
- ▶ les **continuations** (**wrap**) sont définies comme **fonctions internes**.

# Exercice CS de l'examen PC2R de 2016 (III)

```
let client1 () =
  let _ = Event.sync (Event.send s ("brouette", 28)) in
  let _ = Thread.delay (Random.float 2.0) in
  Event.sync (Event.send s ("brouette", 15))

let client2 () =
  let _ = Thread.delay (Random.float 1.0) in
  let _ = Event.sync (Event.send g (c2, "brouette")) in
  let a = Event.sync (Event.receive c2) in
  print_string "Client: Valeur"; print_int a; print_endline ""

let main =
  let _ = Thread.create table_p (cr_assoc ()) in
  let c1 = Thread.create client1 () in
  let c2 = Thread.create client2 () in
  Thread.join c1;
  Thread.join c2
```

## ► code des clients:

- ▶ le premier **associe** "brouette"
- ▶ seul le deuxième **cherche** "brouette"
- ▶ en fonction des **valeurs** des delay, le 2ème client peut **bloquer** **indéfiniment ou non.**

# Conclusion

- ▶ Résumé:
  - ▶ passage de message: paradigme avec des mécanismes de synchronisation et communication basés sur l'envoi et la réception de données.
  - ▶ l'ordre supérieur permet la mobilité: la topologie du système évolue à l'exécution.
  - ▶ Go propose une implémentation efficace de systèmes composés de processus (*goroutines*) communiquant avec des canaux synchrones (ou non).
  - ▶ OCaml propose une implémentation inefficace mais élégante (typage fort, évènements de première classe) de processus (*threads*) communiquant avec des canaux synchrones.
- ▶ TD / TME:
  - ▶ TD: canaux synchrones Go et OCaml
  - ▶ TME: MapReduce en Go
- ▶ Séance prochaine:
  - ▶ passage de messages: utilisations.