La Concurrence

Concepts de Langages de Programmation

Juan-Carlos Barros et Daniel Kessler

juin 2021

La Concurrence

Introduction

Définition

La **concurrence** en informatique consiste en l'exécution de plusieurs **tâches** simultanément (pendant des périodes qui se chevauchent) plutôt que séquentiellement (l'une démarrant après la fin de l'autre).

La concurrence est en nous

Le **cerveau** est intrisèquement **concurrent**.

La concurrence est en nous

Le **cerveau** est intrisèquement **concurrent**.

- traitement du stimulus visuel

Ce mot est rouge.

Ce mot est vert.

Ce mot est bleu.

La concurrence est en nous

Le **cerveau** est intrisèquement **concurrent**.

- traitement du stimulus visuel
- traitement du stimulus auditif

Ce mot est rouge.

Ce mot est vert.

Ce mot est bleu.

La concurrence est en nous

Le **cerveau** est intrisèquement **concurrent**.

- traitement du stimulus visuel
- traitement du stimulus auditif
- traitement de la syntaxe, grammaire puis sémantique

Ce mot est rouge.

Ce mot est vert.

Ce mot est bleu.

La concurrence est en nous

Le **cerveau** est intrisèquement **concurrent**.

- traitement du stimulus visuel
- traitement du stimulus auditif
- traitement de la syntaxe, grammaire puis sémantique
- analyse du sens

Ce mot est rouge.

Ce mot est vert.

Ce mot est bleu.

La concurrence est en nous

Le **cerveau** est intrisèquement **concurrent**.

- traitement du stimulus visuel
- traitement du stimulus auditif
- traitement de la syntaxe, grammaire puis sémantique
- analyse du sens
- comparaison avec les connaissances mémorisées

Ce mot est rouge.

Ce mot est vert.

Ce mot est bleu.

La concurrence est en nous

Le **cerveau** est intrisèquement **concurrent**.

- traitement du stimulus visuel
- traitement du stimulus auditif
- traitement de la syntaxe, grammaire puis sémantique
- analyse du sens
- comparaison avec les connaissances mémorisées
- critique

Ce mot est rouge.

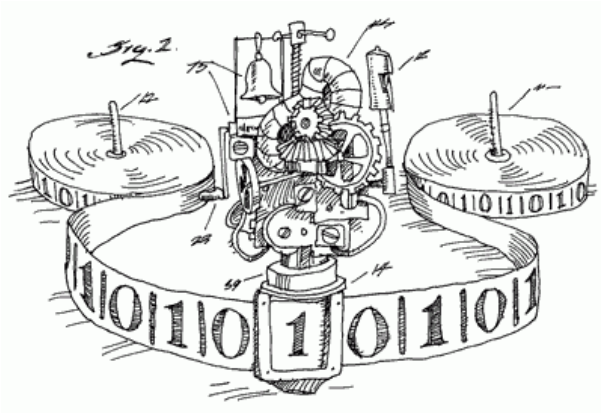
Ce mot est vert.

Ce mot est bleu.

La séquence est dans l'ordinateur

Le **processeur** est intrinsèquement **séquentiel**.

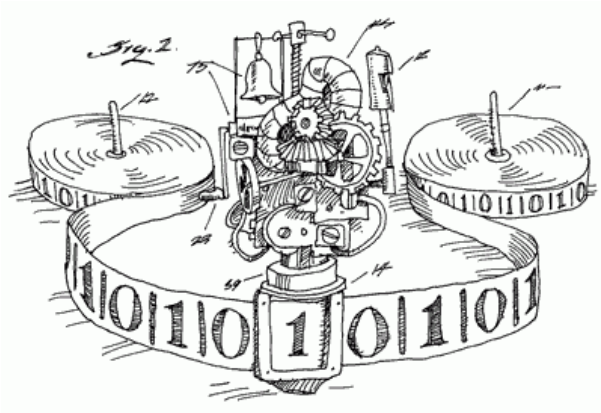
- Une instruction exécutée à la fois



La séquence est dans l'ordinateur

Le **processeur** est intrinsèquement **séquentiel**.

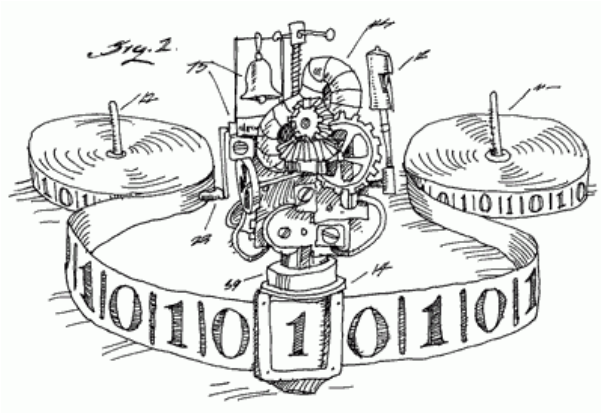
- Une instruction exécutée à la fois
- Pourquoi chercher la concurrence ?



La séquence est dans l'ordinateur

Le **processeur** est intrinsèquement **séquentiel**.

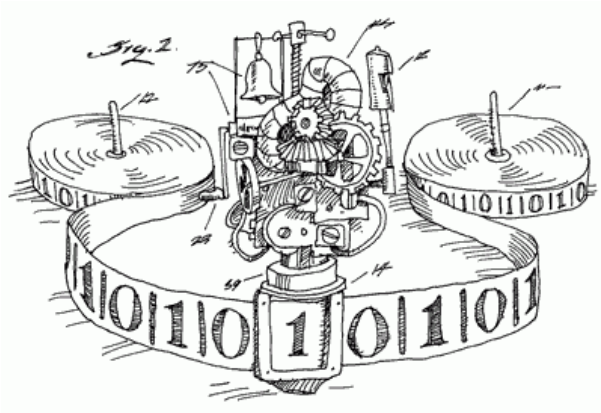
- Une instruction exécutée à la fois
- Pourquoi chercher la concurrence ?
  - Interruptions I/O



La séquence est dans l'ordinateur

Le **processeur** est intrinsèquement **séquentiel**.

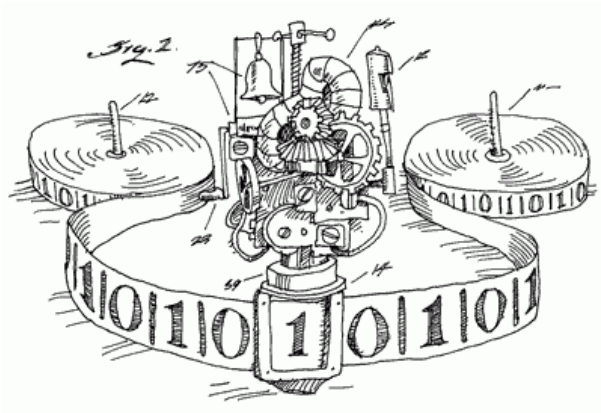
- Une instruction exécutée à la fois
- Pourquoi chercher la concurrence ?
  - Interruptions I/O
  - Exécuter plusieurs programmes "simultanément"



La séquence est dans l'ordinateur

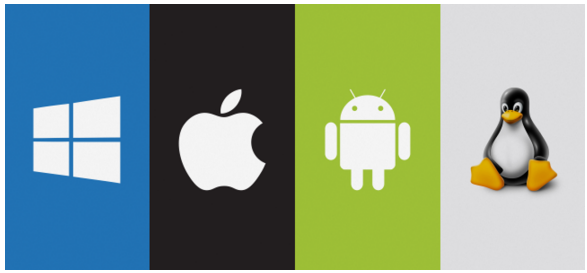
Le **processeur** est intrinsèquement **séquentiel**.

- Une instruction exécutée à la fois
- Pourquoi chercher la concurrence ?
  - Interruptions I/O
  - Exécuter plusieurs programmes "simultanément"
  - Systèmes distribués



OS : le maître de la concurrence

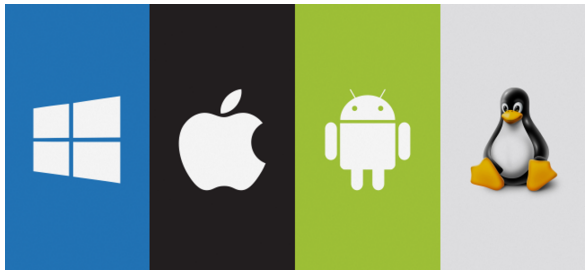
Le **système d'exploitation** est un "magicien".



OS : le maître de la concurrence

Le **système d'exploitation** est un "magicien".

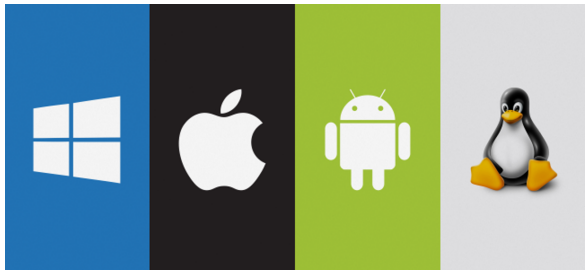
- gestion des ressources séquentielles



OS : le maître de la concurrence

Le **système d'exploitation** est un "magicien".

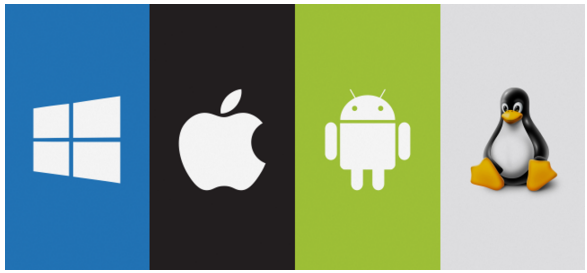
- gestion des ressources séquentielles
- couches d'abstraction



OS : le maître de la concurrence

Le **système d'exploitation** est un "magicien".

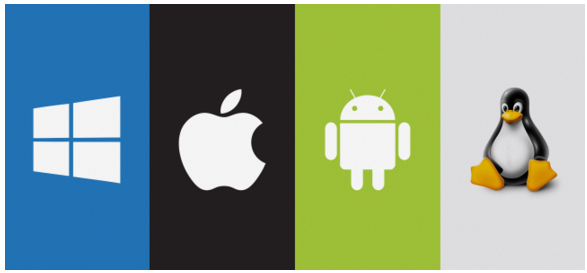
- gestion des ressources séquentielles
- couches d'abstraction
- partage des ressources :
  $\rightarrow$ **l'ordonnanceur**



OS : le maître de la concurrence

Le **système d'exploitation** est un "magicien".

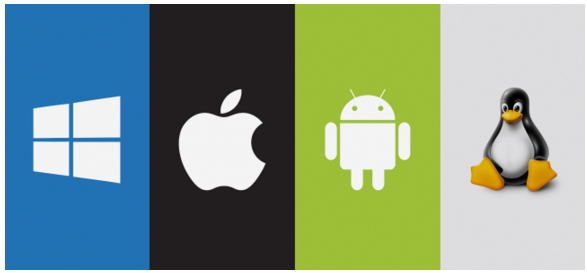
- gestion des ressources séquentielles
- couches d'abstraction
- partage des ressources :
  $\rightarrow$ **l'ordonnanceur**
  - gestion des priorités



OS : le maître de la concurrence

Le **système d'exploitation** est un "magicien".

- gestion des ressources séquentielles
- couches d'abstraction
- partage des ressources :
  $\rightarrow$ **l'ordonnanceur**
  - gestion des priorités
  - commutation des contextes



Mémoire partagée : un problème critique

Les fondamentaux de la concurrence

Comment se manifeste la concurrence ? (répartition des tâches)

- multiplicité des **processus**
- multiplicité des **fils d'exécution** (threads)

Comment communiquent les différents composants ?

- via des **ressources communes**

Problèmes et solutions

Problèmes

- Situation de compétition
- Inversion de priorité
- Entrelacement
- Interblocage
- Famine
- etc.

Solutions

- Verrous
- L'Atomicité
- Les Moniteurs
- Les Sémaphores
- Futures et Promises
- Et d'autres paradigmes

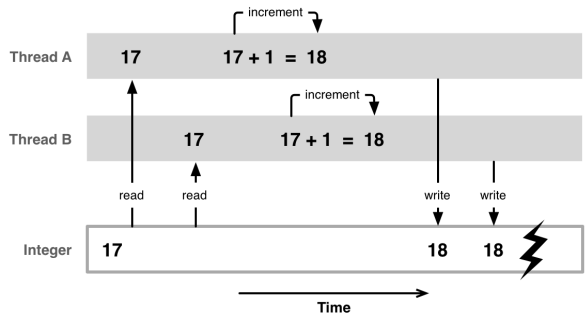
Problème 1 : **interleaving** (“entrelacement”)

Déterminisme ou indéterminisme ?

- programme séquentiel : toujours déterministe
- programme concurrent : peut être déterministe
  → toutes les possibilités d’entrelacement doivent été prises en compte !
  → prouver la validité d’un programme concurrent est particulièrement ardu !
- Si le résultat dépend de l’ordre d’exécution, le déterminisme est perdu

Problème 2 : **race condition**

Deux tâches pourraient tenter d'accéder à une même partie de leur mémoire partagée simultanément, ce qui dans la pratique pourra poser problème.



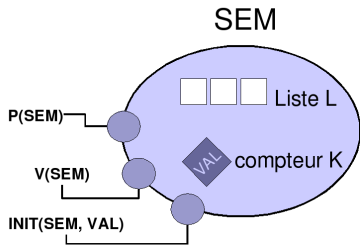
Solution 1 : **verrou** (lock)

Un **verrou** limite l'accès d'une ressource à une seule tâche.

- état fermé *vs* état ouvert
- si le verrou est fermé, les tâches sont bloquées
- sinon, une tâche qui accède à la ressource ferme le verrou
- quand la tâche finit, elle ouvre le verrou

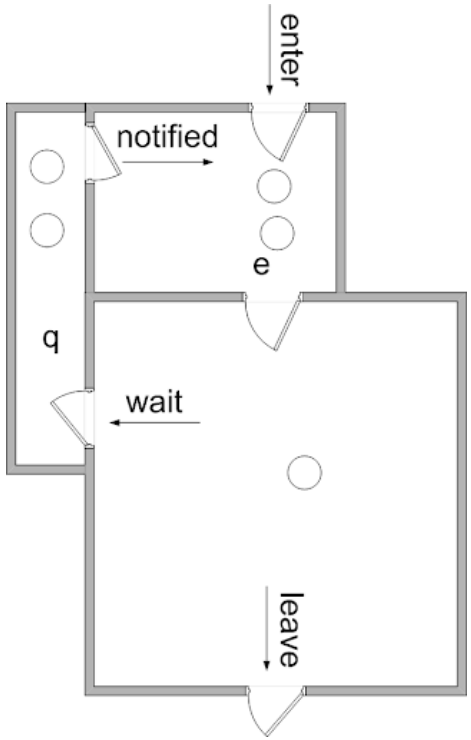
Solution 2 : **sémaphore** de Dijkstra

- Inventé pour l'OS "THE operating system"
- Compteur de tâches en attente de la ressource
- 3 primitives
  - `Init(sem, max)` définit nombres d'utilisateurs max
  - `P(sem)` pour accéder à la ressource
  - `V(sem)` pour quitter la ressource
- `P(sem)` bloque si max atteint, sinon incrémente le compteur
- `V(sem)` décrémente le compteur et notifie les tâches en attente
- cas particulier : **sémaphore binaire** ou **mutex** quand `max=1`.



Solution 3 : **moniteur** (Hoare / Brinch-Hansen)

- Inventé pour *Equivalent Pascal* et *Solo operating system*
- Verrou avec une ou plusieurs **conditions**
- Condition liée à **file d'attente** de processus voulant accéder à une ressource
- Équivalence avec sémaphores



Solution 3 : **moniteur** (Hoare / Brinch-Hansen)

- Inventé pour *Equivalent Pascal* et *Solo operating system*
- Verrou avec une ou plusieurs **conditions**
- Condition liée à **file d'attente** de processus voulant accéder à une ressource
- Équivalence avec sémaphores

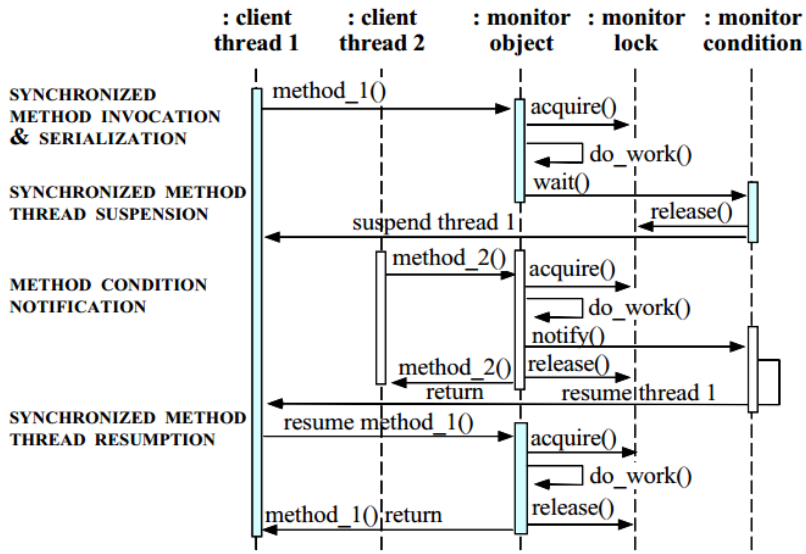


Fig. 3. Sequence diagram of the *Monitor Object* concurrency pattern [Schm00].

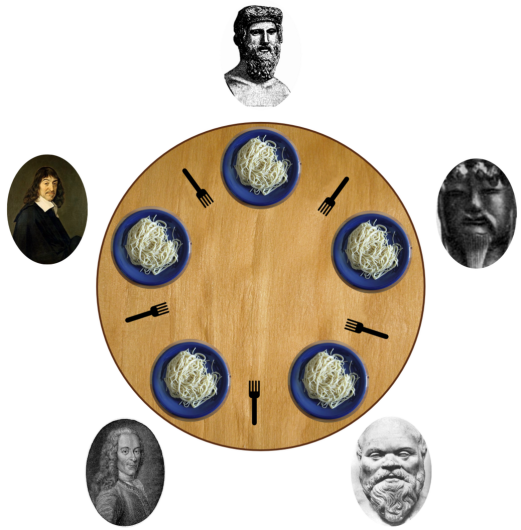
Solution 3 : **moniteur** (Hoare / Brinch-Hansen)

```
public class SynchronizedCounter {
   private int c = 0;

   public synchronized void increment() {
      c++;
   }

   public synchronized void decrement() {
      c--;
   }

   public synchronized int value() {
      return c;
   }
}
```

Problème 3 : **interblocage** et **famine**

Verrous, etc. $\Rightarrow$ nouveaux problèmes !. . .

- Deux tâches pourraient chacune verrouiller une ressource dont l'autre à besoin
  $\rightarrow$ **interblocage**
- Une tâche attend indéfiniment une ressouce bloquée
  $\rightarrow$ **famine**



Solution 4 : Futures et Promises (Hewitt / Liskov)

Les **Futures** et **Promises** sont des abstractions de plus haut niveau que les verrous, sémaphores ou moniteurs.

Un Future/Promise est une valeur qui peut déjà être disponible, ou sinon, devrait devenir disponible plus tard.

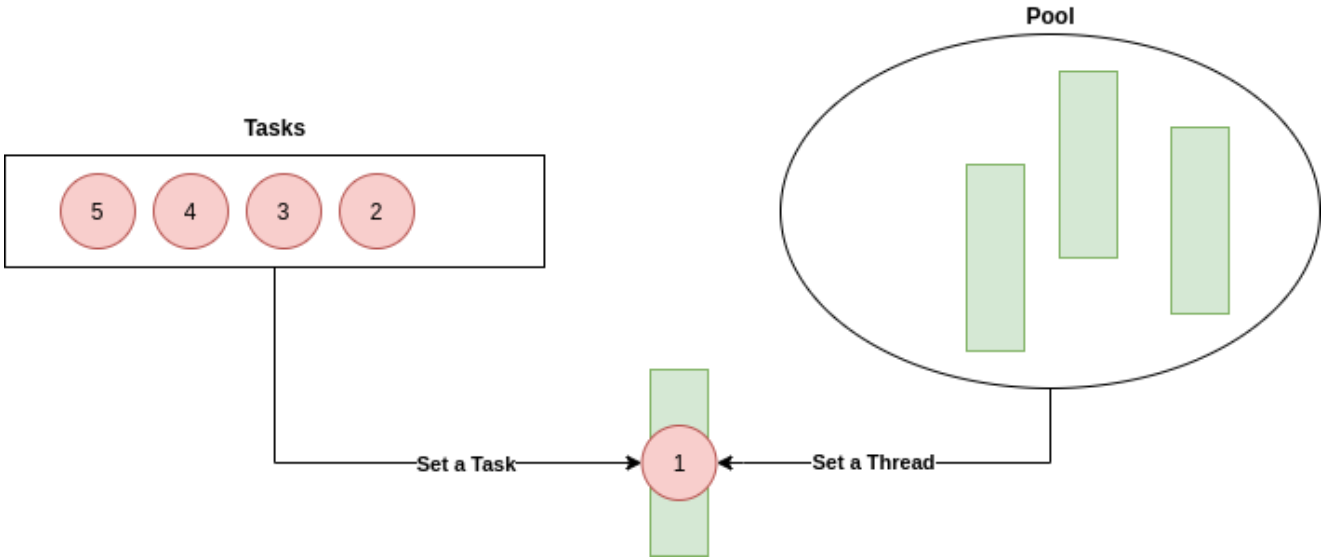
La distinction entre les deux est floue et dépend du langage de programmation.
Si distinction il y a, le Future apparaît comme un objet “en lecture seule”.

Côté pratique

Implémenation naïve d'un Future minimal en Python

```
class FutureNaif:
    """Future implémenté "à la main", avec thread privé."""

    def __init__(self, task):
        """Initialiser un Future."""
        self.done = False
        self.result = None
        self._task = task
        # pas besoin de lock vu que le thread est créé
        # sur place et n'est référencé nulle part ailleurs
        self.__thread = threading.Thread(target=self._target)
        self.__thread.start()

    def _target(self):
        """Travail à effectuer par le thread dédié."""
        self.result = self._task()
        self.done = True
```

Thread/Process Pools et Executors



Tests de performance

Comparaison de deux situations. . .

- Calcul intensif
- I/O avec latence

Tests de performance

Comparaison de deux situations. . .

- Calcul intensif
- I/O avec latence

. . . en utilisant trois approches :

- séquentielle
- concurrente par *threads* (garantis non parallélisés)
- concurrente par *process* (garantis parallélisés)

Changement de paradigme

Future avec **callback** associé (exemple en Scala)

```scala
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global
import scala.util.{Failure, Success}

object ScalaFuture extends App {

  val finiraPlusTard = Future {
    ... // tâche longue
  }

  finiraPlusTard.onComplete {
    case Success(valeur) => traiterValeurDeRetour(valeur)
    case Failure(exception) => gererException(exception)
  }

  Thread.sleep(2000)  // "garder en vie" le runtime
}
```

pipelining de Promise (exemple en javascript)

```
function faireQuelqueChose() {
    // ...
      return new Promise(...)
    }

faireQuelqueChose()
   .then(x => faireAutreChose(x))
   .then(x => faireUneTroisiemeChose(x))
   .then(x => {
         console.log("Résultat final reçu: ${x}");
     })
   .catch(gererEchec);
```

Les fondamentaux de la concurrence (2)

Comment se manifeste la concurrence ? (répartition des tâches)

- multiplicité des **fils d'exécution** (threads)
- multiplicité des **processus**

Comment communiquent les différents composants ?

- via des **ressources communes**

Les fondamentaux de la concurrence (2)

Comment se manifeste la concurrence ? (répartition des tâches)

- multiplicité des **fils d'exécution** (threads)
- multiplicité des **processus**
- multiplicité des processeurs
- multiplicité des machines

Comment communiquent les différents composants ?

- via des **ressources communes**

Les fondamentaux de la concurrence (2)

Comment se manifeste la concurrence ? (répartition des tâches)

- multiplicité des **fils d'exécution** (threads)
- multiplicité des **processus**
- multiplicité des processeurs
- multiplicité des machines

Comment communiquent les différents composants ?

- via des **ressources communes**
- de manière **synchrone** : transmission d'information directe
- de manière **asynchrone** : transmission d'information indirecte

Éviter la mémoire partagée ?

- La mémoire partagée était la source de la plupart des problèmes

  $\Rightarrow$ l'éviter ou la cacher dans une couche d'abstraction qui la gère de manière automatique et efficace

Éviter la mémoire partagée ?

- La mémoire partagée était la source de la plupart des problèmes

  $\Rightarrow$ l'éviter ou la cacher dans une couche d'abstraction qui la gère de manière automatique et efficace
- Les communications ne devraient se faire que par **messages** ou **événements**

Éviter la mémoire partagée ?

- La mémoire partagée était la source de la plupart des problèmes

  $\Rightarrow$ l'éviter ou la cacher dans une couche d'abstraction qui la gère de manière automatique et efficace
- Les communications ne devraient se faire que par **messages** ou **événements**
- Les messages ou événements ne devraient être constitué que de données **immuables**

Modèle d'Acteurs de Hewitt

Un **acteur** peut :

- Envoyer des **messages** à d'autres acteurs
- Créer d'autres acteurs
- Décider de son comportement à la prochaine réception de message

Modèle d'Acteurs en Erlang

Le langage **Erlang** exploite bien ce modèle.

```
-module(acteur).
-export([start/0, alice/0]).

alice() ->
    receive
        hello ->
            io:format("Alice dit bonjour.~n", [])
    end.

start() ->
    Alice_PID = spawn(acteur, alice, []),
    Alice_PID ! hello.
```

Plus récemment, le langage **Scala** utilise ce même modèle, à travers le "toolkit" **Akka**.

Du modèle d'Acteurs au π-calcul de Milner

Le modèle d'acteurs a influencé le π-**calculus** (algèbre de processus)

> *Now, the pure* λ-**calculus is built with just two kinds of thing : terms and variables**. *Can we achieve the same economy for a process calculus ? Carl Hewitt, with his actors model, responded to this challenge long ago ; he declared that a value, an operator on values, and a process should all be the same kind of thing : an actor.*
>
> *(. . .)*
>
> *So, in the spirit of Hewitt, our first step is to demand that all things denoted by terms or accessed by names—values, registers, operators, processes, objects—are all of the same kind of thing ;* **they should all be processes**.

Robin Milner (*Turing Lecture* 1993)

Du modèle d'Acteurs au π-calcul de Milner

Le modèle d'acteurs a influencé le π-**calculus** (algèbre de processus)

$$
\begin{array}{rll}
P, Q ::= & x(y).\, P & \text{Receive on channel } x, \text{ bind the result to } y, \text{ then run } P \\
| & \overline{x}\langle y \rangle.\, P & \text{Send the value } y \text{ over channel } x, \text{ then run } P \\
| & P|Q & \text{Run } P \text{ and } Q \text{ simultaneously} \\
| & (\nu x)P & \text{Create a new channel } x \text{ and run } P \\
| & !P & \text{Repeatedly spawn copies of } P \\
| & 0 & \text{Terminate the process}
\end{array}
$$

Équivalence des paradigmes

Lauer et Needham on montré que les modèles de concurrence basé sur procédures et les modèles basés sur messages sont équivalents (“duaux”).

$\rightarrow$ cf. *On the duality of operating system structures* H. Lauer, R. Needham 1978

- orienté procédures :
  - beaucoup de processus, souvent créés ou détruits
  - communication par mémoire partagée et par appels de fonction
  - synchronisation via verrous, sémaphores ou moniteurs
- orienté messages :
  - peu de processus, en nombre fixe et isolés
  - communication par messages ou événements
  - traitement efficace des files d'attente de messages

Conclusion

Autres approches de la concurrence

Autres approches de la concurrence

- Transactions
  - idée inspirée des Bases de Donnée (par exemple langage **SQL**)
  - rendre “atomique” la section critique
  - idée exploitée par exemple dans le langage **Clojure**

Autres approches de la concurrence

- Transactions
  - idée inspirée des Bases de Donnée (par exemple langage **SQL**)
  - rendre “atomique” la section critique
  - idée exploitée par exemple dans le langage **Clojure**
- Coroutines
  - coroutines : des routines concurrentes pouvant “vivre” dans le même thread
  - idée exploitée par exemple dans le langage **Go** (“goroutines”)

Autres approches de la concurrence

- Transactions
  - idée inspirée des Bases de Donnée (par exemple langage **SQL**)
  - rendre “atomique” la section critique
  - idée exploitée par exemple dans le langage **Clojure**
- Coroutines
  - coroutines : des routines concurrentes pouvant “vivre” dans le même thread
  - idée exploitée par exemple dans le langage **Go** (“goroutines”)
- Modélisation de la concurrence par diagrammes
  - Réseaux de Petri
  - Diagrammes UML de séquence, d’activité et d’état

L'avenir de la concurrence