

# Systèmes concurrents

2SN

20 septembre 2022

## 3 matières

- Systèmes concurrents : modèles, méthodes, outils pour le parallélisme « local »
- Intergiciels : mise en œuvre du parallélisme dans un environnement réparti (machines distantes)
- Projet données réparties : réalisation d'un service de support à la programmation concurrente, parallèle ou répartie.

## Evaluation de l'UE

- Examen Systèmes concurrents : écrit, sur la conception de systèmes concurrents
- (*Examen Intergiciels : écrit*)
- Projet commun : réalisation d'un service de support à la programmation concurrente, parallèle ou répartie.
  - présentation mi-octobre, rendu final mi janvier
  - travail en groupe, suivi + points d'étape réguliers

# Matière : systèmes concurrents – organisation

## Composition

- Cours (50%) : définitions, principes, modèles
- TD (25%) : conception et méthodologie
- TP (25%) : implémentation des schémas et principes

## Fonctionnement (si présentiel)

- Cours : classique, avec un soupçon de style classe inversée
  - version sonorisée disponible en ligne
  - pour les séances 6 et 7 : travail en amont de la séance, puis retour et séance en semi-autonomie
- TD : classique
- TP : classique

## Evaluation

- écrit + bonus (Quiz, QCM)

**Pages de l'enseignement** : <http://moodle-n7.inp-toulouse.fr>,  
<http://queinnec.perso.enseeiht.fr/Ens/sc.html>

**Contact** : [queinnec@enseeiht.fr](mailto:queinnec@enseeiht.fr), [mauran@enseeiht.fr](mailto:mauran@enseeiht.fr)

## Objectif

Être capable de comprendre et développer des applications parallèles (*concurrentes*)

- **modélisation** pour la conception de programmes parallèles
- connaissance des schémas (**patrons**) essentiels
- **raisonnement** sur les programmes parallèles : exécution, propriétés
- **pratique** de la programmation parallèle avec un environnement proposant les objets/outils de base

# Plan du cours

- ➊ Introduction : problématique
- ➋ Exclusion mutuelle
- ➌ Synchronisation à base de sémaphores
- ➍ Interblocage
- ➎ Synchronisation à base de moniteur
- ➏ API Java, Posix Threads
- ➐ Processus communicants – Go, Ada
- ➑ Transactions – mémoire transactionnelle
- ➒ Synchronisation non bloquante

# Première partie

## Introduction

# Contenu de cette partie

- Nature et particularités des programmes concurrents  
⇒ conception et raisonnement systématiques et rigoureux
- Modélisation des systèmes concurrents
- Points clés pour faciliter la conception des applications concurrentes
- Intérêt et limites de la programmation parallèle
- Mise en œuvre de la programmation concurrente sur les architectures existantes

# Plan

## 1 Activités concurrentes

- Le problème
- Un peu d'architecture
- Communication & activités

## 2 Conception

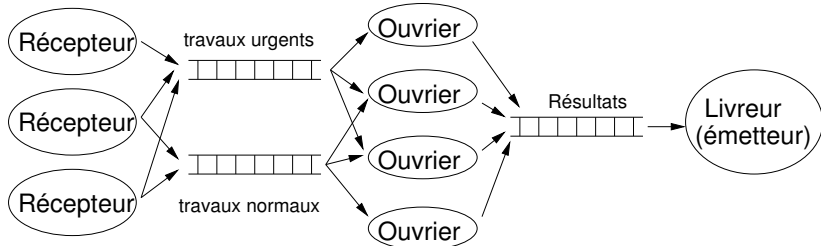
- Comment contrôler (réaliser) la composition ?
- Comment décrire ?
- Comment raisonner ?

## 3 Avantages/inconvénients



# Le problème

Concevoir une application concurrente qui reçoit des demandes de travaux, les régule, et fournit leur résultat



- coopération : les activités « se connaissent »
- compétition : les activités « s'ignorent »
- vitesse d'exécution arbitraire

# Intérêt des systèmes concurrents

- **Facilité de conception**

le parallélisme est naturel sur beaucoup de systèmes

- temps réel : systèmes embarqués, applications multimédia
- mode de fonctionnement : modélisation et simulation de systèmes physiques, d'organisations, systèmes d'exploitation

- **Pour accroître la puissance de calcul**

algorithmique parallèle et répartie

- **Pour faire des économies**

mutualisation de ressources coûteuses via un réseau

- **Parce que la technologie est mûre**

banalisation des systèmes multiprocesseurs, des stations de travail/ordinateurs en réseau, services répartis

# Nécessité des systèmes concurrents

- La puissance de calcul monoprocesseur atteint un plafond
  - l'augmentation des performances d'un processeur dépend directement de sa fréquence d'horloge  $f$
  - l'énergie consommée et dissipée augmente comme  $f^3$   
→ une limite physique est atteinte depuis quelques années
- Les gains de parallélisme au niveau mono-processeur sont limités : processeurs vectoriels, architectures pipeline, GPU conviennent mal à des calculs irréguliers/généraux
- La loi de Moore reste valide : la densité des transistors double tous les 18 à 24 mois

Les architectures multiprocesseurs sont (pour l'instant) le principal moyen d'accroître la puissance de calcul

# Concurrence vs parallélisme

## Parallélisme

Exécution simultanée de plusieurs codes.

## Concurrence

Structuration d'un programme en activités  $\pm$  indépendantes qui interagissent et se coordonnent.

	Pas de concurrence	Concurrence
Pas de parallélisme	prog. séquentiel	plusieurs activités sur un monoprocesseur
Parallélisme	parallélisation automatique / implicite	plusieurs activités sur un multiprocesseur

# Différence avec la programmation séquentielle

- Activités  $\pm$  simultanées  $\Rightarrow$  explosion de l'espace d'états

```
P1          ||          P2
for i := 1 to 10  for j := 1 to 10
print(i)          print(j)
```

- P1 seul  $\rightarrow$  10 états 😊
- P1 || P2  $\rightarrow$  10 x 10 = 100 états ☹
- P1 ; P2  $\rightarrow$  1 exécution 😊
- P1 || P2  $\rightarrow$  184756 exécutions ☹
- Interdépendance des activités
  - logique : production/utilisation de résultats intermédiaires
  - chronologique : disponibilité des résultats

$\Rightarrow$  non déterminisme

$\Rightarrow$  nécessité de méthodes et d'outils (conceptuels et logiciels) pour le raisonnement et le développement

# Composants matériels

Architecture d'un ordinateur :

- Processeurs
- Mécanisme d'interconnexion
- Mémoire et caches

# Processeur

Vision simpliste : à chaque cycle, le processeur exécute une instruction machine à partir d'un flot séquentiel (le code).

En pratique :

- pipeline : plusieurs instructions en cours dans un même cycle : obtention, décodage, exécution, écriture du résultat
- superscalaire : plusieurs unités d'exécution
- instructions vectorielles
- réordonnancement (out-of-order)
- exécution spéculative

# Interconnexion

- Bus unique
  - interconnecte des processeurs entre eux
  - interconnecte les processeurs et la mémoire
  - interconnecte les processeurs et des unités d'E/S
  - médium à diffusion : usage exclusif
- Plusieurs bus dédiés
- Mini réseaux locaux, network-on-chip (parallélisme massif)
- Réseaux locaux classiques (système réparti)



# Mémoire

La mémoire et le processeur sont éloignés : un accès mémoire est considérablement plus lent que l'exécution d'une instruction (facteur 10 à 1000 dans un ordinateur, 10 000 en réparti).

## Principe de localité

**temporelle** si on utilise une adresse, on l'utilisera probablement de nouveau dans peu de temps

**spatiale** si on utilise une adresse, on utilisera probablement une adresse proche dans peu de temps

⇒ conserver près du CPU les dernières cases mémoire accédées

⇒ **Cache** : mémoire rapide proche du processeur

Plusieurs niveaux de caches : de plus en plus gros, de moins en moins rapides (actuellement 3 niveaux).

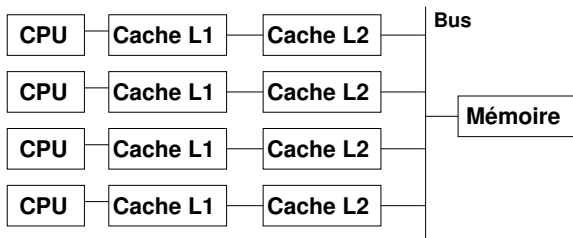


# Cache

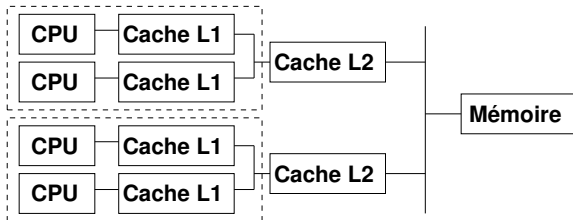
- Différents principes de placement dans le cache (associatif, mappé, k-associatif. . .)
- Différentes stratégies de remplacement (LRU - least recently used. . .)
- Différentes stratégies d'invalidation - cohérence mémoire

# Architecture multiprocesseur

Multiprocesseur « à l'ancienne » :

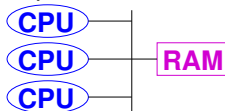


Multiprocesseur multicœur :

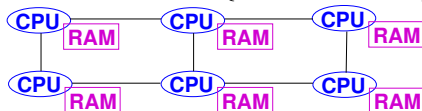


# Architecture multiprocesseur

**SMP** Symmetric multiprocessor : une mémoire + un ensemble de processeurs



**NUMA** Non-Uniform Memory Access : un graphe d'interconnexion de {CPU+mémoire}



**CC-NUMA** Cache-Coherent Non-Uniform Memory Access

# Écritures en mémoire

Comment fonctionne l'écriture d'une case mémoire en présence de caches ?

**Write-Through** diffusion sur le bus à chaque valeur écrite

- + visible par les autres processeurs  $\Rightarrow$  invalidation des valeurs passées
- + la mémoire et le cache sont cohérents
- trafic inutile : écritures répétées, écritures de variables privées au thread

**Write-Back** diffusion uniquement à l'éviction de la ligne

- + trafic minimal
- cohérence cache - mémoire - autres caches ?

# Cohérence mémoire

Si un processeur écrit la case d'adresse  $a_1$ , quand les autres processeurs verront-ils cette valeur ? Si plusieurs écritures consécutives en  $a_1$ ,  $a_2 \dots$ , sont-elles vues dans cet ordre ? Et les lectures indépendantes d'une écriture ?

## Règles de cohérence mémoire

**Cohérence séquentielle** le résultat d'une exécution parallèle est le même que celui d'une exécution séquentielle qui respecte l'ordre partiel de chacun des processeurs.

**Cohérence PRAM** (pipelined RAM ou fifo) les écritures d'un même processeur sont vues dans l'ordre où elles ont été effectuées ; des écritures de processeurs différents peuvent être vues dans des ordres différents.

**Cohérence « lente »** (slow consistency) : une lecture retourne *une* valeur précédemment écrite, sans remonter dans le temps.

# Cohérence Mémoire – exemple

Init :  $x = 0 \wedge y = 0$

Processeur P1		Processeur P2
(1) $x \leftarrow 1$		(a) $y \leftarrow 1$
(2) $r1 \leftarrow y$		(b) $r2 \leftarrow x$

Un résultat  $r1 = 0 \wedge r2 = 0$  est possible en cohérence PRAM et slow, impossible en cohérence séquentielle.

---

Init :  $x = 0 \wedge y = 0$

Processeur P1		Processeur P2
(1) $x \leftarrow 1$		(a) $r1 \leftarrow y$
(2) $y \leftarrow 1$		(b) $r2 \leftarrow x$

Un résultat  $r1 = 1 \wedge r2 = 0$  est possible en cohérence slow ou PSO (partial store order – réordonnement des écritures)

# Activité

Activité/processus/tâches/threads/processus légers/...

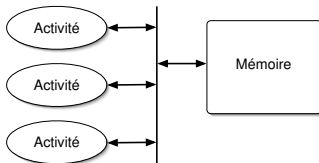
- exécution d'un programme séquentiel
- entité **logicielle**
- exécutable par un processeur
- interruptible et commutable



# Interaction par mémoire partagée

## Système centralisé multitâche

- communication implicite, résultant de l'accès par chaque activité à des variables partagées
- activités anonymes (interaction sans identification)
- coordination (synchronisation) nécessaire (pour déterminer l'instant où une interaction est possible)



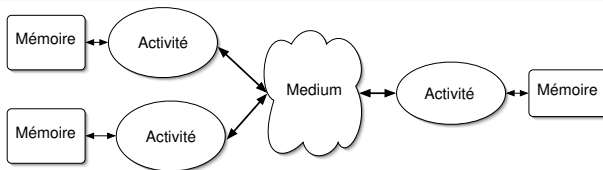
## Exemples

- multiprocesseurs à mémoire partagée,
- processus légers,
- Unix : couplage mémoire (mmap), fichiers

# Interaction par échange de messages

## Activités communiquant par messages

- communication explicite par transfert de données (messages)
- désignation nécessaire du destinataire (ou d'un canal de communication)
- coordination implicite, découlant de la communication



## Exemples

- processeurs en réseau,
- architectures logicielles réparties (client/serveur...),
- Unix : tubes, signaux

# Plan

- 1 Activités concurrentes
  - Le problème
  - Un peu d'architecture
  - Communication & activités
- 2 Conception
  - Comment contrôler (réaliser) la composition ?
  - Comment décrire ?
  - Comment raisonner ?
- 3 Avantages/inconvénients

# Contrôler

## Concevoir une application concurrente

- contrôler un ensemble d'activités concurrentes
- contrôler la **progression** et les interactions de chaque activité
- assurer leur **protection** réciproque

### Moyen

attente (par blocage – suspension – de l'activité)

→ déblocage nécessairement par une autre activité

# Contrôler

Hypothèses de bon comportement des activités : un **protocole** définit les interactions possibles :

- Opérations et paramètres autorisés.
- **Séquences d'actions autorisées.**

*Un ouvrier ne doit pas déposer plus de résultats qu'il n'a pris de travaux.*

# Décrire

## Compteurs d'événements

Compter les actions ou les changements d'états et les relier entre eux.

## Exemple

Invariant  $\#nb$  de travaux soumis =  $\#nb$  travaux effectués  
+  $\#nb$  travaux en cours  
+  $\#nb$  travaux en attente

# Décrire

## Triplets de Hoare

précondition/action/postcondition

## Exemple

$\{t = \text{nb travaux en attente} \wedge t > 0 \wedge r = \text{nb résultats}\}$

ouvrier effectue un travail

$\{\text{nb travaux en attente} = t - 1 \wedge \text{nb résultats} = r + 1\}$

Sérialisation : 
$$\frac{\{p\}A_1; A_2\{q_{12}\}, \{p\}A_2; A_1\{q_{21}\}}{\{p\}A_1 \parallel A_2\{q_{12} \vee q_{21}\}}$$

Indépendance :

$$\frac{A_1 \text{ et } A_2 \text{ sans interférence}, \{p\}A_1\{q_1\}, \{p\}A_2\{q_2\}}{\{p\}A_1 \parallel A_2\{q_1 \wedge q_2\}}$$

# Décrire

## Propriétés temporelles

Linéaires : pour **toutes** les exécutions possibles,  
à tout moment d'une exécution.

Arborescentes : pour **certaines** exécutions possibles,  
à tout moment d'une exécution.

## Exemple

- Sûreté : rien de mauvais ne se produit  
*Deux ouvriers ne peuvent **jamaïs** prendre le même travail.*
- Vivacité : quelque chose de bon finit par se produire  
*Un travail déposé **finit par** être pris par un ouvrier.*
- Possibilité : deux travaux déposés consécutivement **peuvent**  
être exécutés séquentiellement par le même ouvrier.



# Modèle : l'entrelacement

Raisonner sur tous les cas parallèles est trop complexe  
⇒ on raisonne sur des exécutions séquentielles obtenues par **entrelacement** des instructions des différentes activités.

Deux activités  $P = p_1; p_2$  et  $Q = q_1; q_2$ . L'exécution concurrente  $P \parallel Q$  est vue comme (équivalente à) l'une des exécutions :  
 $p_1; p_2; q_1; q_2$  ou  $p_1; q_1; p_2; q_2$  ou  $p_1; q_1; q_2; p_2$  ou  $q_1; p_1; p_2; q_2$  ou  
 $q_1; p_1; q_2; p_2$  ou  $q_1; q_2; p_1; p_2$

Nombre d'entrelacements :  $\frac{(p+q)!}{p! q!}$

## Attention

- Ne pas oublier que c'est un modèle simplificateur (vraie concurrence, cohérence mémoire. . .)
- Il peut ne pas exister de code séquentiel équivalent au code parallèle.

# Raisonner

## Contrôler les effets des interactions

- isoler (raisonner indépendamment)  $\Rightarrow$  modularité
- spécifier/contrôler l'interaction
- schémas connus d'interaction (design patterns)

# Plan

- 1 Activités concurrentes
  - Le problème
  - Un peu d'architecture
  - Communication & activités
- 2 Conception
  - Comment contrôler (réaliser) la composition ?
  - Comment décrire ?
  - Comment raisonner ?
- 3 Avantages/inconvénients

# Avantages/inconvénients

- + utilisation d'un système multiprocesseur.
- + utilisation de la concurrence naturelle d'un programme.
- + modèle de programmation naturel, en explicitant la synchronisation nécessaire.
- surcoût d'exécution (synchronisation, implantation du pseudo-parallélisme).
- surcoût de développement : nécessité d'expliciter la synchronisation, vérifier la réentrance des bibliothèques, danger des variables partagées.
- surcoût de mise-au-point : debuggage souvent délicat (pas de flot séquentiel à suivre) ; effet d'interférence entre des activités, interblocage. . .

# Parallélisme et performance

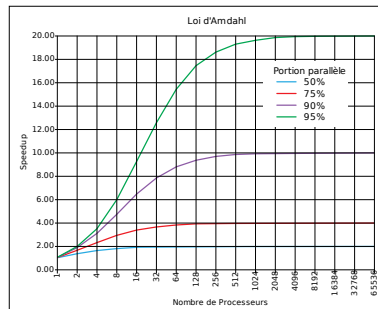
## Mythe du parallélisme

« Si je remplace ma machine mono-processeur par une machine à  $N$  processeurs, mon programme ira  $N$  fois plus vite »

Soit un système composé par une partie  $p$  parallélisable + une partie  $1 - p$  séquentielle.

CPU	durée	$p = 40\%$	$p = 80\%$
1	$p + (1 - p)$	100	100
4	$\frac{p}{4} + (1 - p)$	70	40
8	$\frac{p}{8} + (1 - p)$	65	30
16	$\frac{p}{16} + (1 - p)$	62,5	25
$\infty$		60	20

Loi d'Amdahl : maximal speedup =  $\frac{1}{1-p}$



(source : wikicommons)

# Parallélisme et performance

## Mythe de la performance

« Si je remplace ma machine par une machine  $N$  fois plus rapide, mon programme traitera des problèmes  $N$  fois plus grands dans le même temps »

Pour un problème de taille  $n$  soluble en temps  $T$ , taille de problème soluble dans le même temps sur une machine  $N$  fois plus rapide :

complexité	$N = 4$	$N = 16$	$N = 1024$
$O(n)$	$4n$	$16n$	$1024n$
$O(n^2)$	$\sqrt{4}n = 2n$	$\sqrt{16}n = 4n$	$\sqrt{1024}n = 32n$
$O(n^3)$	$\sqrt[3]{4}n \approx 1.6n$	$\sqrt[3]{16}n \approx 2.5n$	$\sqrt[3]{1024}n \approx 10n$
$O(e^n)$	$\ln(4)n \approx 1.4n$	$\ln(16)n \approx 2.8n$	$\ln(1024)n \approx 6.9n$

En supposant en outre que tout est 100% est parallélisable et qu'il n'y a aucune interférence !