Systèmes concurrents

2SN

17 septembre 2020

UE Systèmes concurrents et communicants

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 de 4, 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

- TDs : classique (si présentiel)
- Cours, TP : à distance, style classe inversée
  travail en amont de la séance (avec retour), séance en semi-autonomie

Evaluation

- Si examen standard : ecrit + bonus (rendus TPs, Quiz)
- Si examen à distance contrôle continu (rendus TPs, quiz) + petit examen en ligne

Pages de l'enseignement : `http://moodle-n7.inp-toulouse.fr`
Contact : `mauran@enseeiht.fr`, `queinnec@enseeiht.fr`

Objectifs

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

1. Introduction : problématique
2. Exclusion mutuelle
3. Synchronisation à base de sémaphores
4. Interblocage
5. Synchronisation à base de moniteur
6. API Java, Posix Threads
7. Processus communicants – Go, Ada
8. Transactions – mémoire transactionnelle
9. Synchronisation non bloquante

Première partie

Introduction

Contenu de cette partie

- nature et particularités des programmes concurrents
  $\Rightarrow$ 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

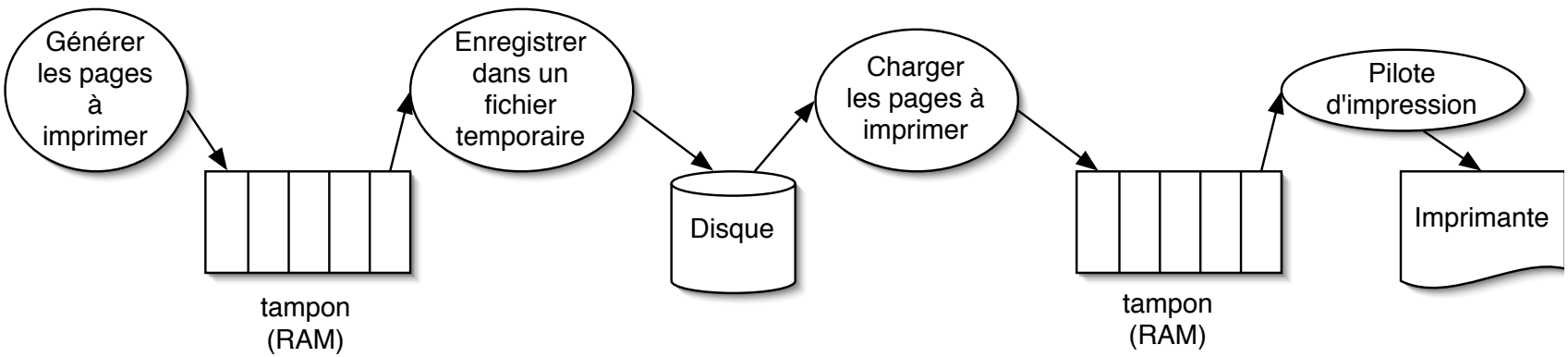
Le problème

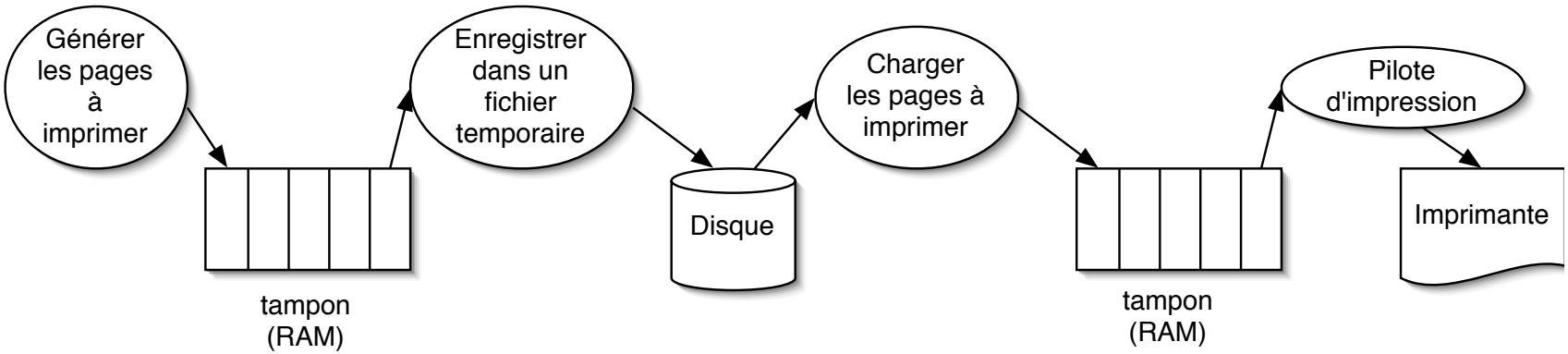
Système concurrent

Ensemble de processus s'exécutant simultanément

- en compétition pour l'utilisation de ressources partagées
- et/ou contribuant à l'obtention d'un résultat commun (global)

Exemple : service d'impression différée

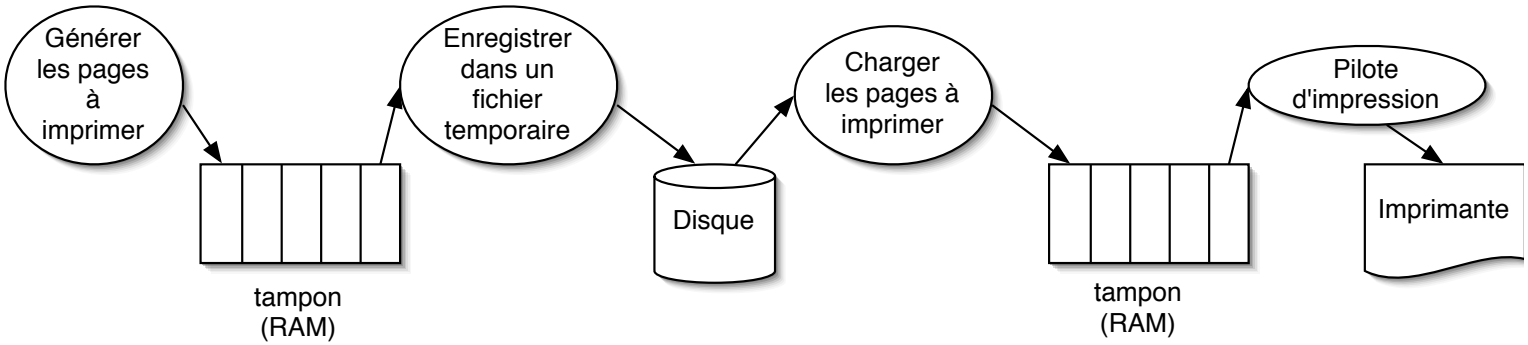




Conception : parallélisation d'un traitement

- décomposition en traitements séquentiels (*processus*)
- exécution simultanée (*concurrente*)
- les processus concurrents ne sont pas indépendants :
  ils partagent des objets (ressources, données)
  $\Rightarrow$ spécifier et contrôler les interactions entre processus

Relations entre activités composées



Chaque activité progresse à son rythme, avec une vitesse arbitraire

$\Rightarrow$ nécessité de réaliser un couplage des activités interdépendantes

- fort : arrêt/reprise des activités «en avance» (*synchronisation*)
- faible : stockage des données échangées et non encore utilisées (*schéma producteur/consommateur*)

Expression du contrôle des interactions : 2 niveaux d'abstraction

- *coopération* (dépôt/retrait sur le tampon) :
  les activités « se connaissent » (interactions explicites)
- *compétition* (accès au disque) :
  les activités « s'ignorent » (interactions transparentes)

Intérêt de la programmation concurrente

Intérêt de la programmation concurrente

- 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 multi-processeurs, des stations de travail/ordinateurs en réseau, services répartis

Nécessité de la programmation concurrente

- 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
      $\rightarrow$ une limite physique est atteinte depuis quelques années
  - les gains de parallélisme au niveau du processeur sont limités
    - processeurs vectoriels, architectures pipeline conviennent mal à des calculs irréguliers/généraux
  - coût excessif de l'augmentation de la taille des caches qui permettrait de compenser l'écart croissant de performances entre processeurs et mémoire
- La loi de Moore reste valide :
  la densité des transistors double tous les 18 à 24 mois

$\rightarrow$ les architectures multiprocesseurs sont pour l'instant le principal moyen d'accroître la puissance de calcul

Qu'est-ce qui fait que la programmation concurrente est différente de la programmation séquentielle ?

```
                variables globales : s, i
        P1                                  ||
s := 0                                      ||
pour i:= 1 à 10 pas 1                       ||
    s := s+i                                ||
fin_pour                                    ||
afficher(s,i)                               ||
```

- P1 seul → 12 états 

Qu'est-ce qui fait que la programmation concurrente est différente de la programmation séquentielle ?

- plusieurs activités simultanées ⇒ explosion de l'espace d'états

```
          variables globales : s, i
     P1                          ||      P2
s := 0                           ||   s := 0
pour i:= 1 à 10 pas 1            ||   pour i:= 1 à 10 pas 1
   s := s+i                      ||      s := s+i
fin_pour                         ||   fin_pour
afficher(s,i)                    ||   afficher(s,i)
```

  - P1 seul → 12 états 
  - P1 ‖ P2 → 12 x 12 = 144 états 

Qu'est-ce qui fait que la programmation concurrente est différente de la programmation séquentielle ?

- plusieurs activités simultanées ⇒ explosion de l'espace d'états

```
          variables globales : s, i
     P1                          ||      P2
s := 0                           ||   s := 0
pour i:= 1 à 10 pas 1            ||   pour i:= 1 à 10 pas 1
   s := s+i                      ||      s := s+i
fin_pour                         ||   fin_pour
afficher(s,i)                    ||   afficher(s,i)
```

  - P1 seul → 12 états 
  - P1 ‖ P2 → 12 x 12 = 144 états 

- interdépendance des activités
  - logique : production/utilisation de résultats intermédiaires
  - chronologique : disponibilité des résultats

  ⇒ non déterminisme (⇒ difficulté du raisonnement par scénarios)

Qu'est-ce qui fait que la programmation concurrente est différente de la programmation séquentielle ?

- plusieurs activités simultanées ⇒ explosion de l'espace d'états

```
        variables globales : s, i
     P1                          ||     P2
s := 0                           ||  s := 0
pour i:= 1 à 10 pas 1            ||  pour i:= 1 à 10 pas 1
   s := s+i                      ||     s := s+i
fin_pour                         ||  fin_pour
afficher(s,i)                    ||  afficher(s,i)
```

  - P1 seul → 12 états
  - P1 ‖ P2 → 12 x 12 = 144 états

- interdépendance des activités
  - logique : production/utilisation de résultats intermédiaires
  - chronologique : disponibilité des résultats

  ⇒ non déterminisme (⇒ difficulté du raisonnement par scénarios)

⇒ nécessité d'outils (conceptuels et logiciels) pour assurer le raisonnement et le développement

Plan

Modèle d'exécution

Activité (ou : processus, processus léger, thread, tâche...)

- Représente l'activité d'exécution d'un programme séquentiel par un processeur
- Vision simple (simplifiée) : à chaque cycle, le processeur
  - extrait (lit et décode) une instruction machine à partir d'un flot séquentiel (le code exécutable),
  - exécute cette instruction,
  - puis écrit le résultat éventuel (registres, mémoire RAM).

$\rightarrow$ exécution d'un processus P

$=$ suite d'instructions effectuées $p_1; p_2; \ldots p_n$ (*histoire* de P)

Exécution concurrente

L'exécution concurrente (simultanée) d'un ensemble de processus $(P_i)_{i \in I}$ est représentée comme une exécution consistant en un *entrelacement arbitraire* des histoires de chacun des processus P_i

Exemple : 2 processus P=$p_1; p_2; p_3$ et Q=$q_1; q_2$
L'exécution concurrente de P et de Q sera vue comme (équivalente à) l'une des exécutions suivantes :

$p_1; p_2; p_3; q_1; q_2$ ou $p_1; p_2; q_1; p_3; q_2$ ou $p_1; p_2; q_1; q_2; p_3$ ou
$p_1; q_1; p_2; p_3; q_2$ ou $p_1; q_1; p_2; q_2; p_3$ ou $p_1; q_1; q_2; p_2; p_3$ ou
$q_1; p_1; p_2; p_3; q_2$ ou $q_1; p_1; p_2; q_2; p_3$ ou $q_1; p_1; q_2; p_2; p_3$ ou
$q_1; q_2; p_1; p_2; p_3$

Le modèle d'exécution par entrelacement est il réaliste ?

Le modèle d'exécution par entrelacement est il réaliste ?

Abstraction réalisée

Deux instructions a et b de deux processus différents ayant une période d'exécution commune donnent un résultat identique à celui de $a; b$ ou de $b; a$

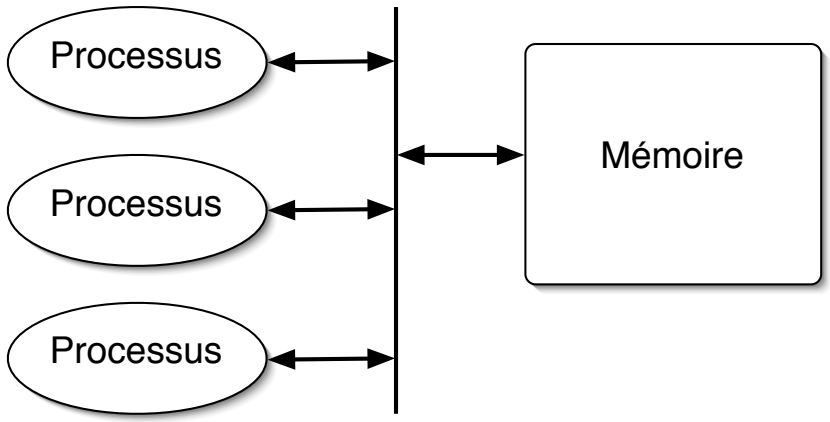
Motivation

- abstrait (ignore) les possibilités de chevauchement dans l'exécution des opérations
  $\Rightarrow$ on se ramène à un ensemble *discret* de possibilités (espace d'états/produit d'histoires)
- entrelacement *arbitraire* : pas d'hypothèse sur la vitesse relative de progression des activités
  $\Rightarrow$ modélise l'hétérogénéité et la charge des processeurs
- abstraction $\ll$ raisonnable $\gg$ au regard des architectures réelles (voir dernière section)

Modèles d'interaction : interaction par mémoire partagée

Système centralisé multi-tâches

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



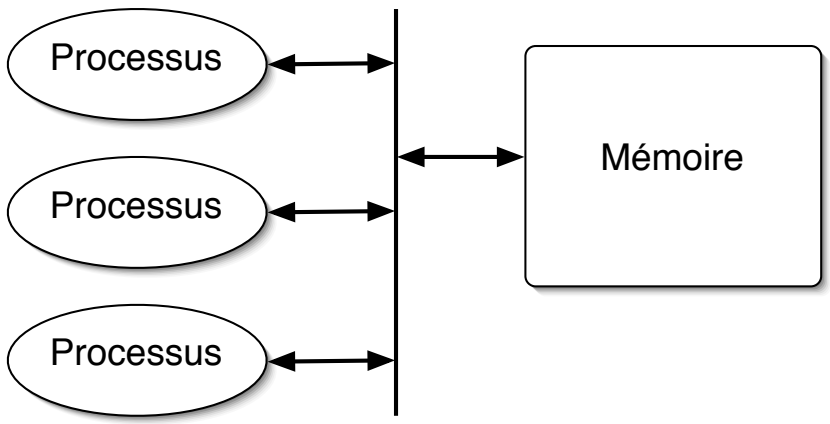
Exemples

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

Modèles d'interaction : interaction par mémoire partagée

Système centralisé multi-tâches

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



Exemples

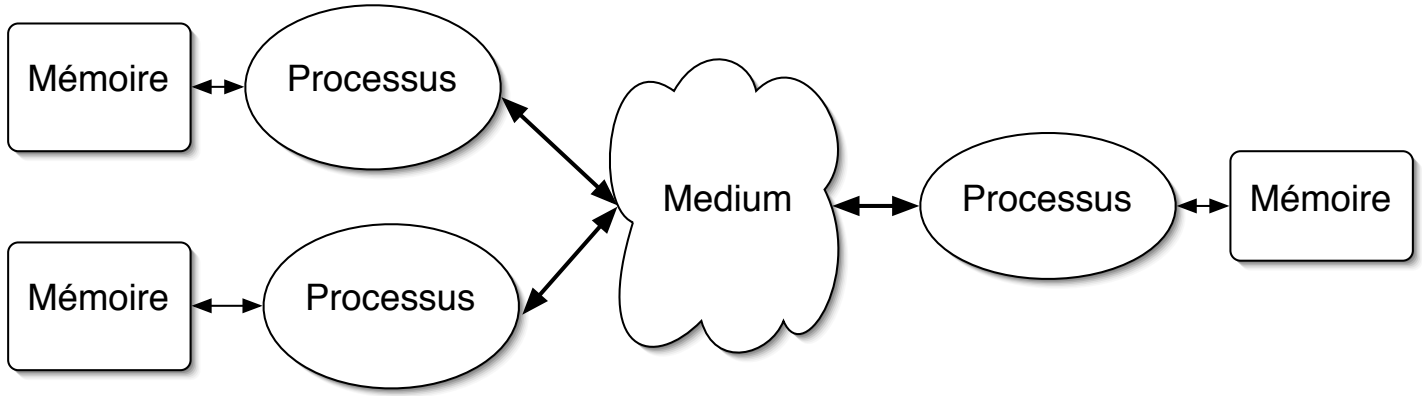
- multiprocesseurs à mémoire partagée,
- processus légers,
- Unix : couplage mémoire (mmap), fichiers

Modèles d'interaction : échange de messages

Processus communiquant par messages

Système réparti

- communication explicite par transfert de données (messages)
- désignation nécessaire du destinataire
- coordination implicite, découlant de la communication



Exemples

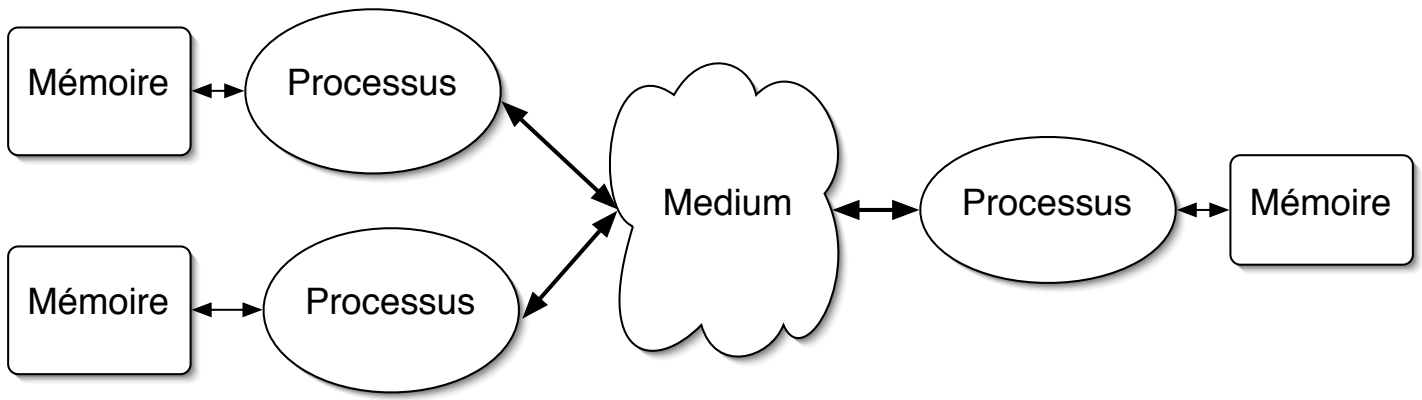
- processeurs en réseau,
- architectures logicielles réparties (client/serveur. . .),
- Unix : tubes, signaux

Modèles d'interaction : échange de messages

Processus communiquant par messages

Système réparti

- communication explicite par transfert de données (messages)
- désignation nécessaire du destinataire
- coordination implicite, découlant de la communication



Exemples

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

Spécifier un programme

Pourquoi ?

Difficulté à raisonner sur les systèmes concurrents
(explosion combinatoire de l'espace d'états/des histoires possibles)

Comment ?

Approche classique : donner les propriétés souhaitées du système, puis vérifier que ces propriétés sont valides lors des exécutions

Spécifier un programme

Pourquoi ?

Difficulté à raisonner sur les systèmes concurrents
(explosion combinatoire de l'espace d'états/des histoires possibles)

Comment ?

Approche classique : donner les propriétés souhaitées du système, puis vérifier que ces propriétés sont valides lors des exécutions

Particularité : calculs **interdépendants** et/ou **réactifs**

→ propriétés fonctionnelles (S=f(E)) insuffisantes/inappropriées
→ propriétés sur l'évolution des traitements, au fil du temps

Spécifier un programme

Pourquoi ?

Difficulté à raisonner sur les systèmes concurrents
(explosion combinatoire de l'espace d'états/des histoires possibles)

Comment ?

Approche classique : donner les propriétés souhaitées du système, puis vérifier que ces propriétés sont valides lors des exécutions

Particularité : calculs **interdépendants** et/ou **réactifs**

→ propriétés fonctionnelles (S=f(E)) insuffisantes/inappropriées
→ propriétés sur l'évolution des traitements, au fil du temps

- Un programme est caractérisé par l'ensemble de ses exécutions possibles
- exécution = histoire, suite d'instructions/d'états
  (état = valeur des variables)

→ propriétés d'un programme = propriétés de ses histoires possibles

Propriété d'une histoire (suite d'états)

Validité d'un prédicat d'état

- à chaque étape de l'exécution :
  propriété de sûreté (il n'arrive jamais rien de mal)
- après un nombre de pas fini :
  propriété de vivacité (une bonne chose finit par arriver)

Exemple

- Sûreté : *Deux serveurs ne prennent jamais le même travail.*
- Vivacité : *Un travail déposé finit par être pris par un serveur*

Remarque : les propriétés exprimées peuvent porter sur

- toutes les exécutions du programme (logique temporelle linéaire)
- ou seulement certaines exécutions du programme (LT arborescente)

Les propriétés que nous aurons à considérer se limiteront généralement au cadre (plus simple) de la LT linéaire.

Vérifier les propriétés : analyse des exécutions

Définition de l'effet d'une opération : triplets de Hoare

{précondition} Opération {postcondtion}

- précondition (hypothèse) :
  propriété devant être vérifiée avant l'exécution de l'opération
- postcondition (conclusion) :
  propriété garantie par l'exécution de l'opération

Exemple

$\{t = \text{nb requêtes en attente} \wedge t > 0 \wedge r = \text{nb résultats}\}$
le serveur traite une requête
$\{\text{nb requêtes en attente} = t - 1 \wedge \text{nb résultats} = r + 1\}$

Analyse d'une exécution

- partir d'une propriété (hypothèse) caractérisant l'état initial
- appliquer en séquence les opérations de l'histoire :
  propriété établie par l'exécution d'une op. = précondition de l'op. suivante

Vérifier les propriétés : analyse des exécutions

Définition de l'effet d'une opération : triplets de Hoare

{précondition} Opération {postcondtion}

- précondition (hypothèse) :
  propriété devant être vérifiée avant l'exécution de l'opération
- postcondition (conclusion) :
  propriété garantie par l'exécution de l'opération

Exemple

$\{t = \text{nb requêtes en attente} \wedge t > 0 \wedge r = \text{nb résultats}\}$
le serveur traite une requête
$\{\text{nb requêtes en attente} = t - 1 \wedge \text{nb résultats} = r + 1\}$

Analyse d'une exécution

- partir d'une propriété (hypothèse) caractérisant l'état initial
- appliquer en séquence les opérations de l'histoire :
  propriété établie par l'exécution d'une op. = précondition de l'op. suivante

Analyse des exécutions : propriétés d'actions concurrentes

Propriétés établies par la combinaison des actions (exemples)

Sérialisation (sémantique de l'entrelacement) :

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

Indépendance (des effets de calculs séparés) :

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

Plan

Conception des systèmes concurrents

Point clé :
contrôler les effets des interactions/interférences entre processus

- isoler (raisonner indépendamment) $\rightarrow$ modularité
- contrôler/spécifier l'interaction
  - définir les instants où l'interaction est possible
  - relier ces instants au flot d'exécution de chacun des processus

Modularité : pouvoir raisonner sur chaque activité séparément

Atomicité

mécanisme/protocole garantissant qu'une (série d')opération(s) est exécutée complètement et sans interférence (isolément)

- grain fin (instruction)
  - (modèle) utile pour le raisonnement : entrelacement
  - (matériel) utile pour déterminer un résultat en cas de conflit
- gros grain (bloc d'instructions) : utile pour la conception.

Réalisation directe :

Modularité : pouvoir raisonner sur chaque activité séparément

Atomicité

mécanisme/protocole garantissant qu'une (série d')opération(s) est exécutée complètement et sans interférence (isolément)

- grain fin (instruction)
  - (modèle) utile pour le raisonnement : entrelacement
  - (matériel) utile pour déterminer un résultat en cas de conflit
- gros grain (bloc d'instructions) : utile pour la conception.

Réalisation directe :

exclusion mutuelle (bloquer tous les processus sauf 1)

- verrous
- masquage des interruptions (sur un monoprocesseur)
- ...

Contrôle des interactions : synchronisation

Mise en œuvre : attente

Un processus prêt pour une interaction est mis en attente (bloqué), jusqu'à ce que **tous** les processus participants soient prêts.

Expression

- en termes de
  - flot de contrôle : placer un *point de synchronisation commun* dans le code de chacun des processus d'un groupe de processus. Ce point de synchronisation définira un instant d'exécution commun à ces processus.
  - flot de données : définir les *échanges* de données entre processus (émission/réception de messages, ou d'événements). L'ordonnancement des processus suit la circulation de l'information.
- globale (barrière, événements, invariants) ou individuelle (rendez-vous, canaux)

Comment pouvoir raisonner sur chaque interaction séparément ? (1/3)

Principe

Définir les interactions permises, indépendamment des calculs

Première idée

Spécifier les suites d'interactions possibles (légales) pour les activités

→ grammaire définissant les suites d'opérations (interactions) permises (expressions de chemins)

→ moyen de vérifier de manière simple et indépendante du code des processus si 1 exécution (trace) globale est correcte (légale)

Exemple : interaction client/serveur

A tout moment, *nb d'appels à déposer_tâche* $\geq$ *nb d'appels à traiter_tâche*

Difficulté

Composition (ajout/retrait d'opérations $\Rightarrow$ redéfinir les suites)

Comment pouvoir raisonner sur chaque interaction séparément ? (2/3)

Deuxième étape

Définir les interactions permises, indépendamment des opérations

Idée

Les processus doivent se synchroniser parce qu'il **partagent** un objet

- à construire (coopération)
- à utiliser (concurrence)

→ spécifier un objet partagé, caractérisé par un ensemble d'états possibles (légaux) : invariant portant sur l'état de l'objet partagé

Exemple : *la file des travaux à traiter peut contenir de* 0 *à Max travaux*

→ indépendance par rapport aux opérations des processus
(Les interactions correctes sont celles qui maintiennent l'invariant)

Difficulté

Nécessite de connaître l'invariant (OK pour un système fermé)

Comment pouvoir raisonner sur chaque interaction séparément ? (3/3)

Systèmes ouverts

Situation : tous les processus ne sont pas connus à l'avance (au moment de la conception)

→ définition de critères de cohérence :

- proposer 1 interface d'accès aux objets partagés, permettant de
- contrôler (automatiquement) les accès pour garantir une propriété globale sur le résultat de l'exécution, indépendamment de l'ordre d'exécution réel

Exemples

- Equivalence à une exécution en exclusion mutuelle
  → maintien de tout invariant : mémoire transactionnelle
- Equivalence à une exécution entrelacée : cohérence mémoire

Plan

Bilan

+ modèle de programmation naturel
– 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 : débogage souvent délicat (pas de flot séquentiel à suivre, non déterminisme) ; effet d'interférence entre des activités, interblocage...
+ **parallélisme (répartition ou multiprocesseurs) = moyen actuel privilégié pour augmenter la puissance de calcul**

Parallélisme et performance

Idée naïve sur le parallélisme

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

Parallélisme et performance

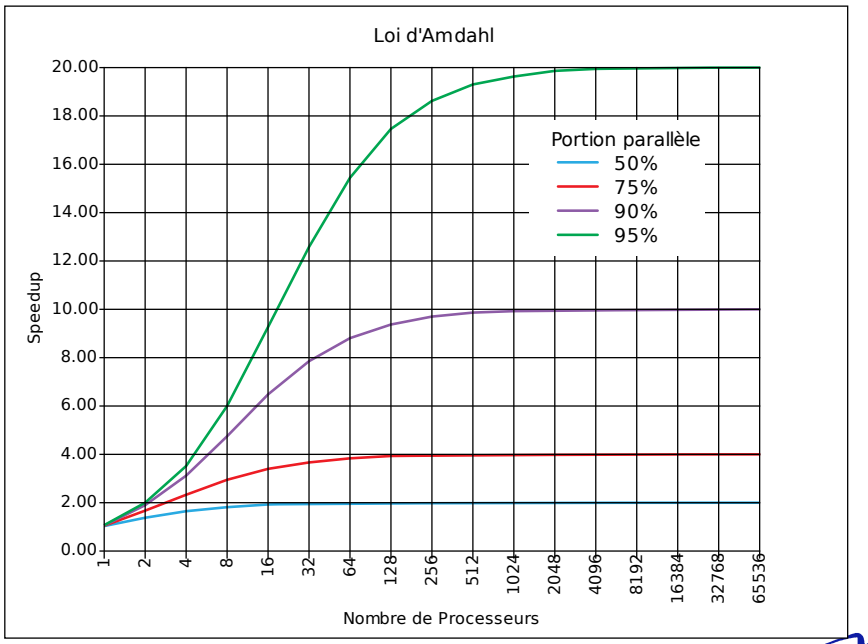
Idée naïve sur le parallélisme

« *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
∞	$0 + (1-p)$	60	20

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



(source : wikicommons)

Parallélisme et performance

Idée naïve sur 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 »

Parallélisme et performance

Idée naïve sur 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 !

Plan

Evaluation : architecture monoprocesseur

Modèle d'exécution abstrait : entrelacement

L'exécution concurrente (simultanée) d'un ensemble de processus $(P_i)_{i \in I}$ est représentée comme une exécution consistant en un *entrelacement arbitraire* des histoires de chacun des processus P_i

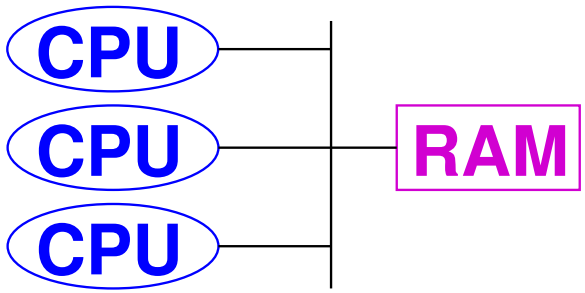
Réalisation sur un monoprocesseur

Pseudo parallélisme (ou parallélisme virtuel)

- le processeur est alloué à tour de rôle à chacun des processus par l'ordonnanceur du système d'exploitation
- le modèle reflète la réalité
- le parallélisme garde tout son intérêt comme
  - outil de conception et d'organisation des traitements,
  - et pour assurer une indépendance par rapport au matériel.

Evaluation : multiprocesseurs SMP (vrai parallélisme)

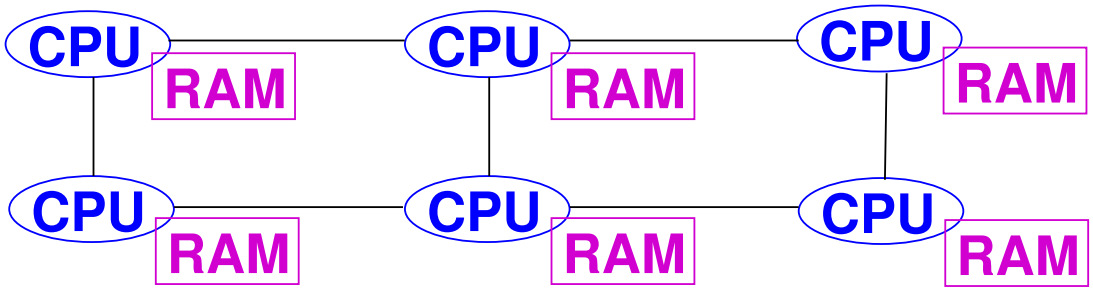
[SMP] Symmetric MultiProcessor :
une mémoire + un ensemble de processeurs



- tant que les processus travaillent sur des zones mémoires distinctes $a; b$ ou $b; a$ ou encore une exécution réellement simultanée de a et b donnent le même résultat
- si a et b opèrent simultanément sur une même zone mémoire, le résultat serait imprévisible, *mais* les requêtes d'accès à la mémoire sont (en général) traitées en séquence par le matériel, pour une taille de bloc donnée.
  Le résultat sera donc le même que celui de $a; b$ ou de $b; a$
- le modèle reflète donc la réalité

Evaluation : multiprocesseurs NUMA (vrai parallélisme)

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



- chaque nœud/site opère sur sa mémoire locale, et traite en séquence les requêtes d'accès à sa mémoire locale provenant d'autres sites/nœuds
- le modèle reflète donc la réalité

Modèle et réalité : un bémol

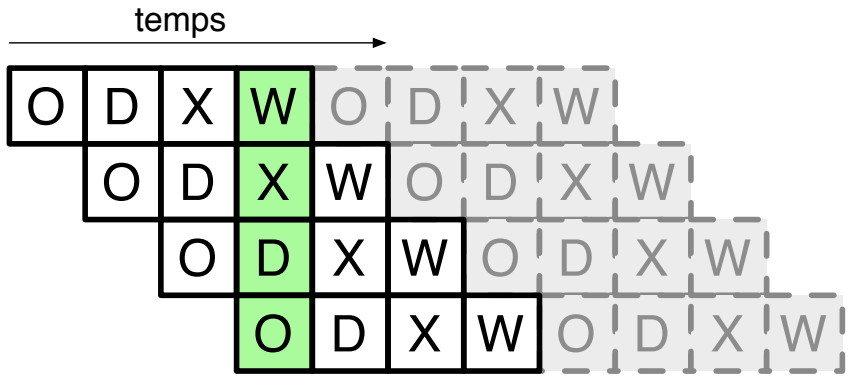
Les architectures récentes éloignent le modèle de la réalité :

- au niveau du processeur : fragmentation et concurrence à grain fin
  - pipeline : plusieurs instructions en cours dans un même cycle : obtention, décodage, exécution, écriture du résultat
  - superscalaire : plusieurs unités d'exécution (et pipeline)
  - instructions vectorielles
  - réordonnancement (out-of-order)
- au niveau de la mémoire : utilisation de caches

Concurrence à grain fin : pipeline

Principe

- chaque instruction comporte une série d'étapes :
  obtention (O)/décodage (D)/exécution (X)/écriture du résultat (W)
- chaque étape est traitée par un circuit à part
- le pipeline permet de charger plusieurs instructions et ainsi d'utiliser simultanément les circuits dédiés, chacun opérant sur une instruction



Difficulté

dépendances entre données utilisées par des instructions proches

```
ADD R1, R1,1     # R1++
SUB R2, R1, 10   # R2 := R1 - 10
```

Remèdes

- insertion de NOP (bulles) pour limiter le traitement parallèle
- réordonnancement (éloignement) des instructions dépendantes

Caches

La mémoire et le processeur sont éloignés : un accès mémoire est considérablement plus lent que l'exécution d'une instruction (peut atteindre un facteur 100 dans un ordinateur, 10000 en réparti).
Principe de localité :

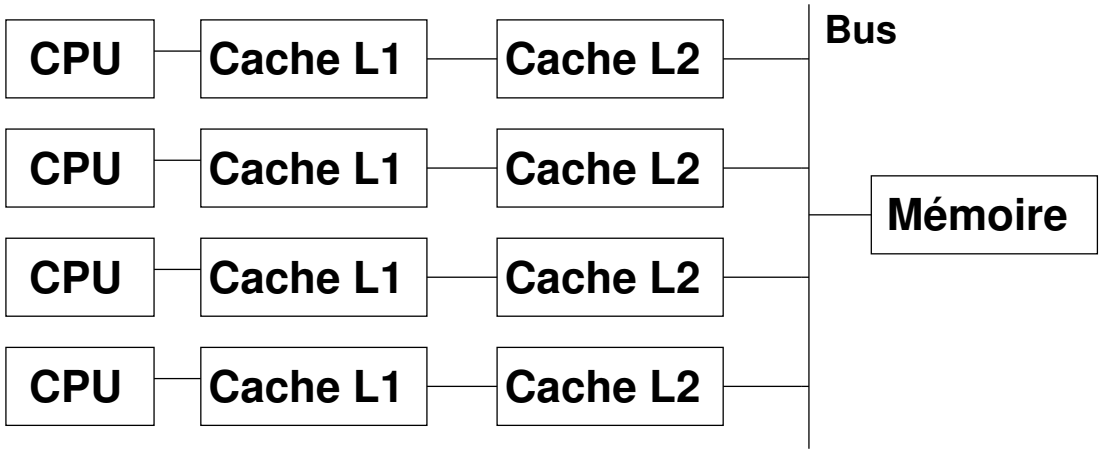
- **temporelle** si on utilise une adresse, on l'utilisera probablement de nouveau dans peu de temps
- **spatiale** si on utilise une adresse, on utilisera probablement une adresse proche dans peu de temps

⇒ conserver près du CPU les dernières cases mémoire accédées
⇒ Cache : mémoire rapide proche du processeur

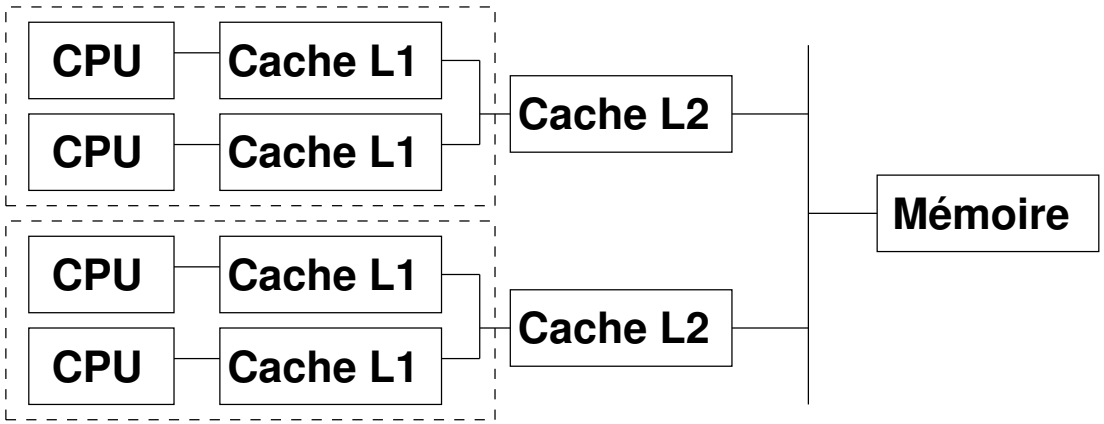
Plusieurs niveaux de caches : de plus en plus gros, de moins en moins rapides (couramment 3 niveaux).

Caches sur les architectures à multi-processeurs

Multi-processeurs « à l'ancienne » :



Multi-processeurs multi-cœurs :



Problème :

cohérence/arbitrage si plusieurs copies en cache d'un même mot mémoire

Comment fonctionne l'écriture d'une case mémoire avec les caches ?

Write-Through diffusion sur le bus à chaque valeur écrite

- \+ visible par les autres processeurs ⇒ 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, $a_2 \ldots$, sont-elles vues dans cet ordre ?

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 : $\texttt{x} = 0 \wedge \texttt{y} = 0$

Processeur P1	Processeur P2
(1) $\texttt{x} \leftarrow 1$	(a) $\texttt{y} \leftarrow 1$
(2) $\texttt{t1} \leftarrow \texttt{y}$	(b) $\texttt{t2} \leftarrow \texttt{x}$

Un résultat $\texttt{t1} = 0 \wedge \texttt{t2} = 0$ est possible en cohérence PRAM et slow, impossible en cohérence séquentielle.

Le mot de la fin

Les mécanismes disponibles sur les architectures actuelles permettent d'accélérer l'exécution de traitements indépendants, mais n'offrent pas de garanties sur la cohérence du résultat de l'exécution d'activités coordonnées/interdépendantes

- contrôler/débrayer ces mécanismes
  - vidage des caches
  - inhibition des caches ($\approx$ variables `volatile` en Java)
  - remplissage des pipeline
  - choix de protocoles de cohérence mémoire
- préciser les hypothèses faites sur le matériel par les différents protocoles de synchronisation
  Exemple : accès séquentiels sur les variables partagées