Systèmes concurrents

Philippe Quéinnec

ENSEEIHT
Département Sciences du Numérique

13 septembre 2024

Première partie

Introduction

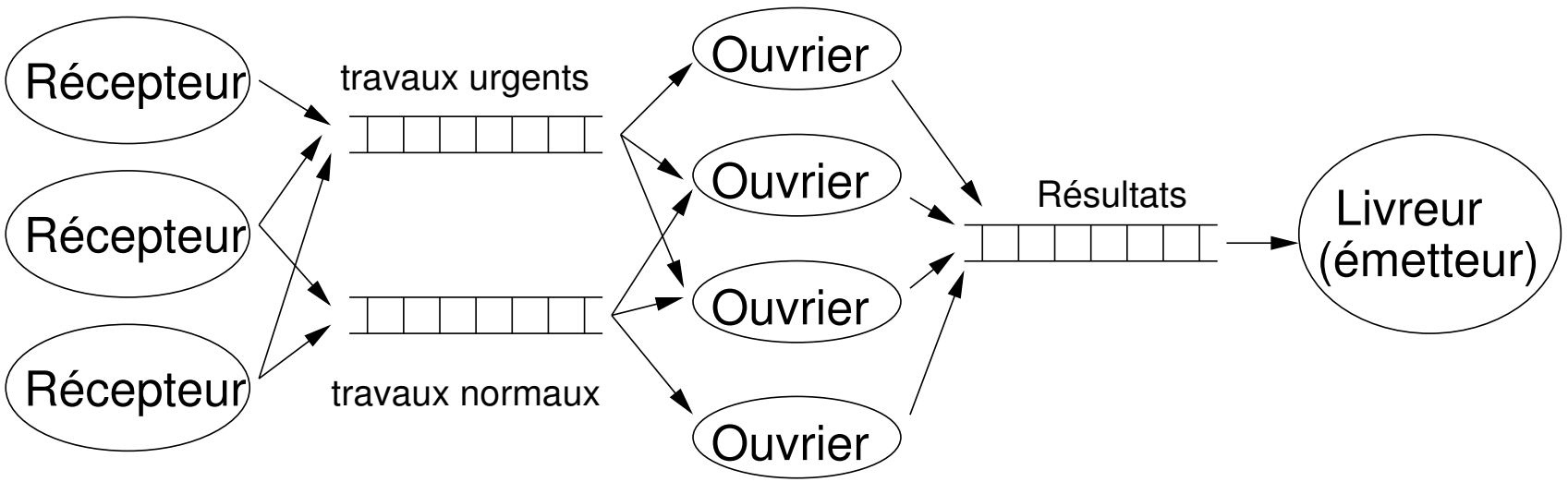
Contenu de cette partie

- Nature et particularités des programmes concurrents
- 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

Exemple de 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

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

Différence avec la programmation séquentielle

- Activités ± simultanées ⇒ explosion de l'espace d'états

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

  - P1 seul → 10 états ☺
  - P1 ‖ P2 → 10 x 10 = 100 états ☹
  - P1 ; P2 → 1 exécution ☺
  - P1 ‖ P2 → 184756 exécutions ☹
- Interdépendance des activités
  - logique : production/utilisation de résultats intermédiaires
  - chronologique : disponibilité des résultats

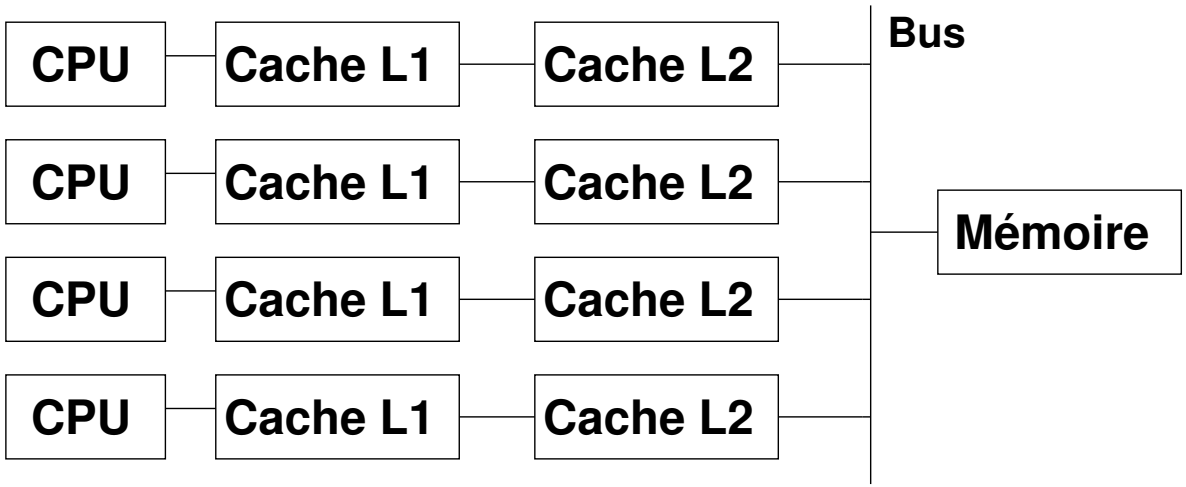
  ⇒non déterminisme

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

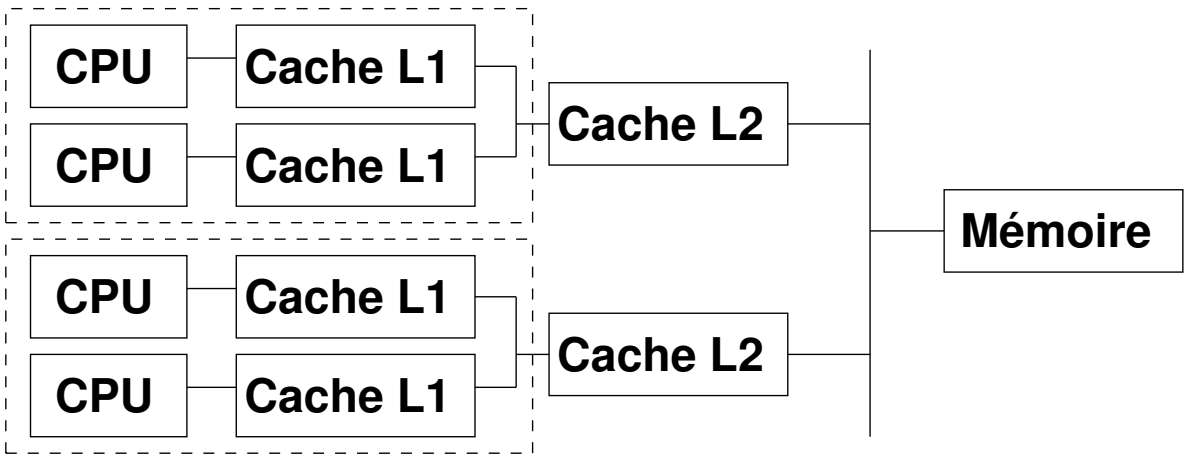
Plan

Architecture multiprocesseur

Multiprocesseur « à l'ancienne » :

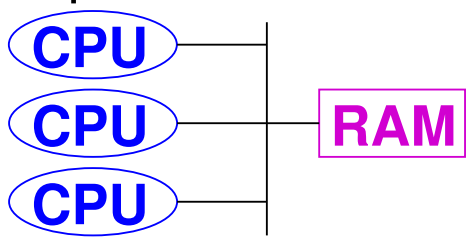


Multiprocesseur multicœur :

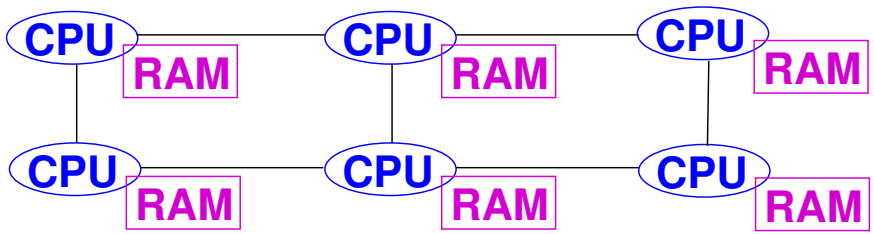


Architecture multiprocesseur

SMP Symmetric multiprocessor : une mémoire + un ensemble de processeurs



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



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..., 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 ∧ y = 0

Processeur P1	Processeur P2
(1) x ← 1	(a) y ← 1
(2) r1 ← y	(b) r2 ← x

Un résultat r1 = 0 ∧ r2 = 0 est possible en cohérence PRAM et *slow*, impossible en cohérence séquentielle.

Init : x = 0 ∧ y = 0

Processeur P1	Processeur P2
(1) x ← 1	(a) r1 ← y
(2) y ← 1	(b) r2 ← x

Un résultat r1 = 1 ∧ r2 = 0 est possible en cohérence *slow* ou PSO (*partial store order* – réordonnancement des écritures)

Que retenir : éviter de se trouver dans de telles situations !

Plan

Activité

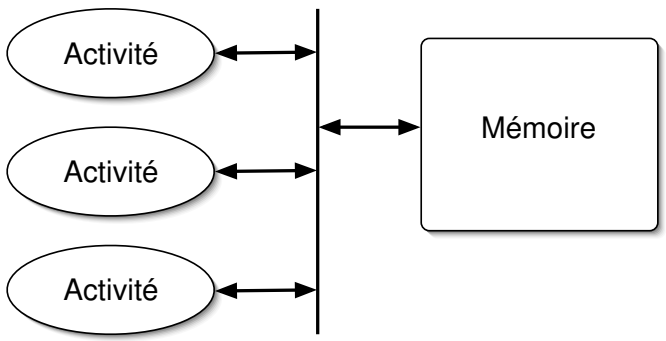
Activité/processus/tâches/threads/processus légers/...

- 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)
- synchronisation nécessaire pour déterminer l'instant où une interaction est possible



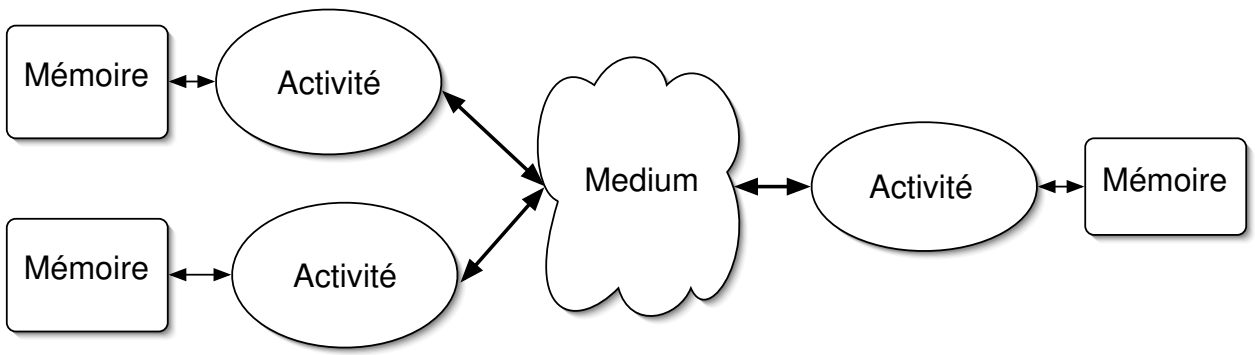
Exemples

- 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
- synchronisation implicite, découlant de la communication



Exemples

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

Modèle : l'entrelacement

Raisonner sur tous les cas parallèles est trop complexe
$\Rightarrow$ 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 $P \| Q$ 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

- C'est un modèle simplificateur (vraie concurrence, granularité de l'entrelacement, cohérence mémoire...)
- Il peut ne pas exister de code séquentiel équivalent au code parallèle.

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...

Deuxième partie

L'exclusion mutuelle

Contenu de cette partie

- Difficultés résultant d'accès concurrents à un objet partagé
- Besoin d'accès exclusif
- Réalisation de l'exclusion mutuelle

Plan

Avoir du pain à la maison

Vous

1. Arrivez à la maison
2. Constatez qu'il n'y a plus de pain
3. Allez à une boulangerie
4. Achetez du pain
5. Revenez à la maison
6. Rangez le pain

Votre colocataire

1. Arrive à la maison
2. Constate qu'il n'y a plus de pain
3. Va à une boulangerie
4. Achète du pain
5. Revient à la maison
6. Range le pain

Spécification

Propriétés de correction

- Sûreté : un seul pain est acheté
- Vivacité : s'il n'y a pas de pain, quelqu'un en achète

Que se passe-t-il si

- votre colocataire est arrivé après que vous êtes revenu de la boulangerie ?
- votre colocataire est arrivé pendant que vous étiez à la boulangerie ?
- votre colocataire est arrivé avant que vous n'alliez à la boulangerie ?
- votre colocataire attend que vous soyez là pour vérifier s'il y a du pain ?

$\Rightarrow$ *race condition* quand la correction dépend de l'ordonnancement des actions

Solution 1 ?

Vous (processus A)

```
A1. si (pas de pain
        && pas de note) alors
A2.   laisser une note
A3.   aller acheter du pain
A4.   enlever la note
   finsi
```

Colocataire (processus B)

```
B1. si (pas de pain)
        && pas de note) alors
B2.   laisser une note
B3.   aller acheter du pain
B4.   enlever la note
    finsi
```

$\Rightarrow$ deux pains possibles si entrelacement A1.B1.A2.B2...

Solution 2 ?

Vous (processus A)

```
A1. laisser une note A
A2. si (pas de note B) alors
A3.    si pas de pain alors
A4.       aller acheter du pain
       finsi
    finsi
A5. enlever la note A
```

Colocataire (processus B)

```
B1. laisser une note B
B2. si (pas de note A) alors
B3.    si pas de pain alors
B4.       aller acheter du pain
       finsi
    finsi
B5. enlever la note B
```

$\Rightarrow$ zéro pain possible (entrelacement A1, B1, A2, B2...)

Solution 3 ?

Vous (processus A)

```
laisser une note A
tant que note B faire
   rien
fintq
si pas de pain alors
     aller acheter du pain
finsi
enlever la note A
```

Colocataire (processus B)

```
laisser une note B
si (pas de note A) alors
   si pas de pain alors
      aller acheter du pain
   finsi
finsi
enlever la note B
```

Correct mais pas satisfaisant

Solution peu évidente, asymétrique / Hypothèse de progression / Attente active

Accès concurrents

Exécution concurrente

`init x = 0;` *// partagé*

$\langle$ `a := x; x := a + 1` $\rangle \parallel \langle$ `b := x; x := b - 1` $\rangle$

$\Rightarrow$ `x = -1, 0` ou `1`

Modification concurrente

$\langle$ `x := 0x`$\boxed{\texttt{00}}\boxed{\texttt{01}}$ $\rangle \parallel \langle$ `x := 0x`$\boxed{\texttt{02}}\boxed{\texttt{00}}$ $\rangle$

$\Rightarrow$ `x = 0x0001` ou `0x0200` ou `0x0201` ou `0x0000` ou `1234` !

Cohérence mémoire

`init x = 0` $\wedge$ `y = 0`

$\langle$ `x := 1; y := 2` $\rangle \parallel \langle$ `printf("%d %d",y,x);` $\rangle$

$\Rightarrow$ affiche `0 0` ou `2 1` ou `0 1` ou `2 0` !

On voudrait que les codes $\langle \cdots \rangle$ soient atomiques.

L'exclusion mutuelle

Une section critique est une section de code qui doit être exécutée atomiquement et sans interférence. L'exclusion mutuelle garantit qu'à tout instant il y a au plus une section critique en cours d'exécution.

- Ensemble d'activités concurrentes A_i
- Variables partagées par toutes les activités
  Variables privées locales à chaque activité
- Structure des activités :

```
cycle
      entrée   section critique   sortie
      ⋮
fincycle
```

- Hypothèses :
  - vitesse d'exécution non nulle
  - section critique de durée finie

Propriétés du protocole d'accès

- (sûreté) à tout instant, au plus une activité est en cours d'exécution d'une section critique

$$\textbf{invariant } \forall i, j \in 0..N-1 : A_i.excl \wedge A_j.excl \Rightarrow i = j$$

- (progression ou vivacité globale) lorsqu'il y a (au moins) une demande, une activité qui demande à entrer sera admise

$$(\exists i \in 0..N-1 : A_i.dem) \rightsquigarrow (\exists j \in 0..N-1 : A_j.excl)$$
$$\forall i \in 0..N-1 : A_i.dem \rightsquigarrow (\exists j \in 0..N-1 : A_j.excl)$$

- (vivacité individuelle) si une activité demande à entrer, elle finira par obtenir l'accès (son attente est finie)

$$\forall i \in 0..N-1 : A_i.dem \rightsquigarrow A_i.excl$$

($p \rightsquigarrow q$: à tout instant, si p est vrai, alors q sera vrai ultérieurement)

Plan

Instruction matérielle TestAndSet

Soit TestAndSet(x), instruction indivisible qui positionne x à vrai et renvoie l'ancienne valeur :

Définition

```
function TestAndSet (x : in out boolean) : boolean
     declare oldx : boolean
begin
     oldx := x; x := true;
     return oldx;
end TestAndSet
```

Algorithme (Protocole d'exclusion mutuelle)

```
occupé : shared boolean := false;

tant que TestAndSet(occupé) faire nop;
     section critique
occupé ← false;
```

Instruction FetchAndAdd

Définition

```
function FetchAndAdd (x : in out int) : int
     declare oldx : int
begin
     oldx := x; x := oldx + 1;
     return oldx;
end FetchAndAdd
```

```
ticket : shared int := 0;
tour : shared int := 0;
montour : int;   // local à l'activité
montour ← FetchAndAdd(ticket);
tant que tour ≠ montour faire nop;
     section critique
FetchAndAdd(tour);
```

Spinlock x86

Spinlock Linux 2.6

```
; initialement Lock = 1
acquire: lock dec word [Lock]
         jns cs                 ; jump if not signed
    spin: cmp dword [Lock], 0
         jle spin               ; loop if ≤ 0
         jmp acquire            ; retry entry
      cs:                       ; section critique
release: mov dword [Lock], 1
```

`lock dec` = décrémentation atomique multiprocesseur avec positionnement du bit "sign"

Tous les processeurs actuels possèdent une instruction analogue au TestAndSet et adaptée aux multiprocesseurs symétriques.

La réalité

Actuellement, tout environnement d'exécution fournit un mécanisme de verrou (*lock*), avec les opérations atomiques :

- obtenir (*acquire*) : si le verrou est libre, l'attribuer à l'activité demandeuse ; sinon bloquer l'activité demandeuse
- rendre/libérer (*release*) : si au moins une activité est en attente du verrou, transférer la possession à l'un des demandeurs et le débloquer ; sinon marquer le verrou comme libre.

Algorithme

```
accès : shared lock
accès.acquire
    section critique
accès.release
```

Troisième partie

Interblocage

Contenu de cette partie

- Définition et caractérisation des situations d'interblocage
- Protocoles de traitement de l'interblocage
  - préventifs
  - curatifs
- Apport déterminant d'une bonne modélisation/formalisation pour la recherche et la validation de solutions

Plan

Le problème

Contexte : allocation de ressources réutilisables

- non réquisitionnables,
- non partageables,
- en quantités entières et finies,
- dont l'usage est indépendant de l'ordre d'allocation.

En particulier les verrous.

Problème

P_1 demande A puis B,
P_2 demande B puis A
$\rightarrow$ risque d'interblocage :

1. P_1 demande et obtient A
2. P_2 demande et obtient B
3. P_2 demande $A \rightarrow$ se bloque
4. P_1 demande $B \rightarrow$ se bloque

Définition de l'interblocage

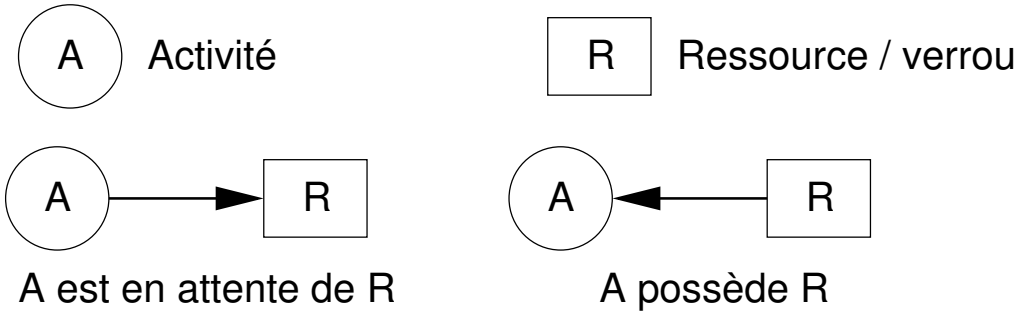
Interblocage

Un ensemble d'activités est en interblocage si et seulement si toute activité de l'ensemble est en attente d'une ressource qui ne peut être libérée que par une autre activité de cet ensemble.

Pour l'ensemble d'activités interbloquées :
Interblocage $\equiv$ négation de la progression

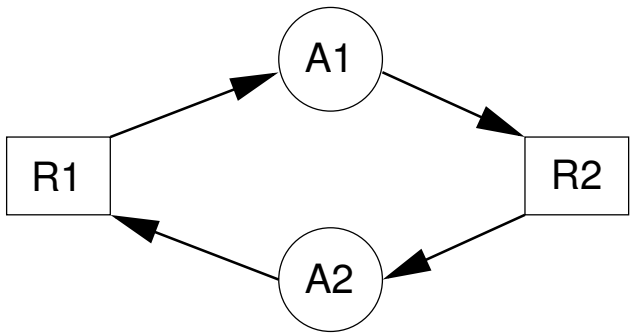
L'interblocage est un état stable.

Notation : graphe d'allocation



Condition nécessaire et suffisante à l'interblocage

Attente circulaire : cycle dans le graphe d'allocation



Solutions

Prévention : empêcher la formation de cycles dans le graphe

Détection + guérison : détecter l'interblocage et l'éliminer

Plan

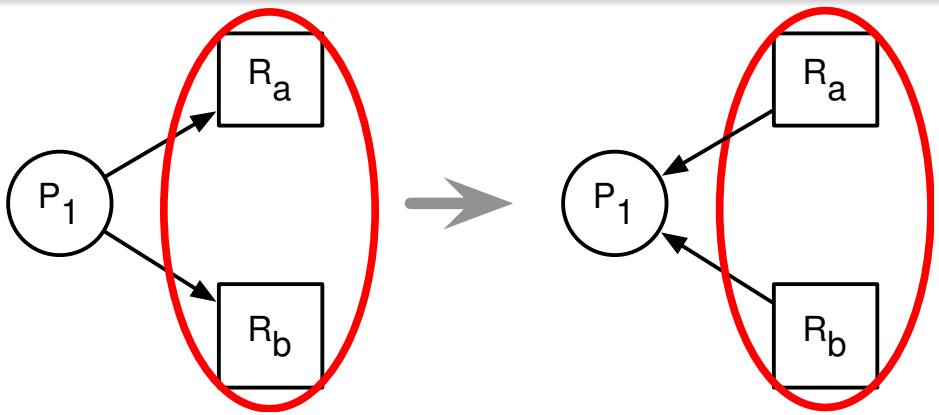
Allocation globale

Éviter les demandes fractionnées

Allocation globale : chaque activité demande et obtient en bloc, en une seule fois, toutes les ressources nécessaires

→ une seule demande pour chaque activité

- demande satisfaite → arcs entrants uniquement
- demande non satisfaite → arcs sortants (attente) uniquement



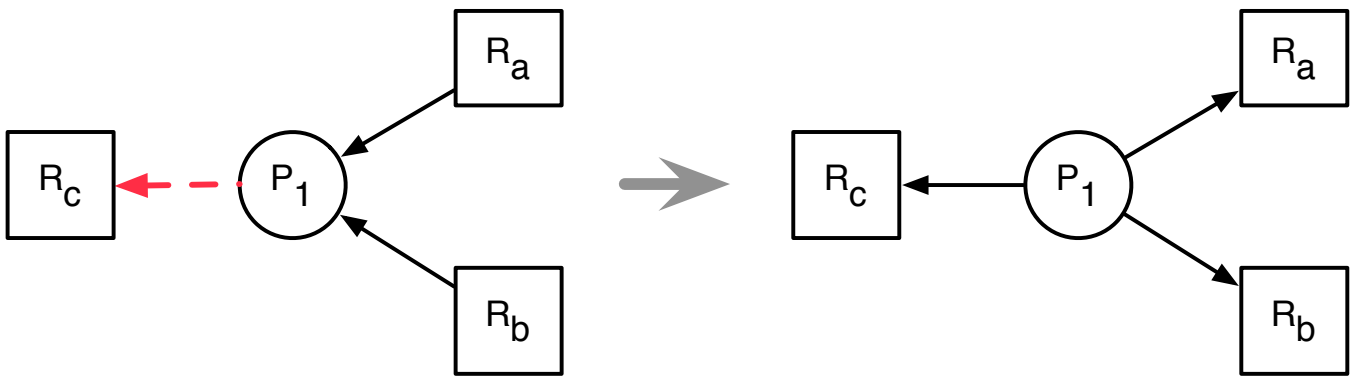
- Revient à remplacer plusieurs verrous par un unique verrou
- Suppose la connaissance a priori des ressources / verrous utilisés
- Sur-allocation, risque de famine, réduction du parallélisme

Non conservation des ressources allouées

Permettre la réquisition des ressources allouées

Inverser les arcs entrants d'une activité si création d'arcs sortants.
Une activité demandeuse doit :

- libérer les ressources qu'elle a obtenues
- réobtenir les ressources libérées, avant de pouvoir poursuivre
  - risque de famine
  - programmation complexe

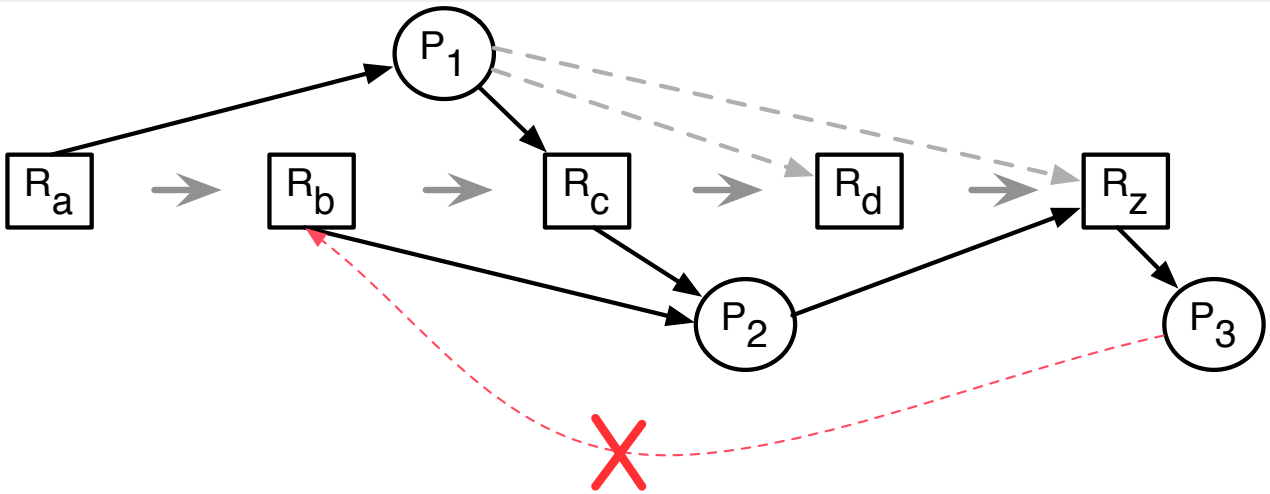


(optimisation : restitution paresseuse des ressources : libération que si la demande est bloquante)

Classes ordonnées

Fixer un ordre global sur les demandes : classes ordonnées

- Un ordre est défini sur les ressources
- Toute activité doit demander les ressources en respectant cet ordre



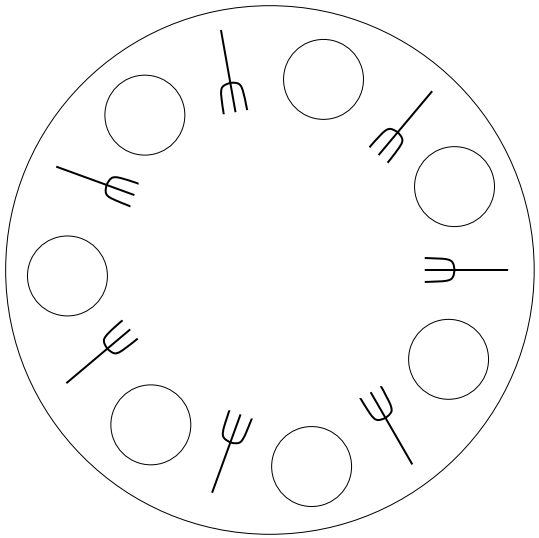
→ pour chaque activité, les chemins du graphe d'allocation vont des ressources inférieures (déjà obtenues) aux supérieures (demandées)

⇒ tout chemin du graphe d'allocation suit l'ordre des ressources

⇒ le graphe d'allocation est sans cycle

Exemple : philosophes et interblocage (1/2)

N philosophes sont autour d'une table. Il y a une assiette par philosophe et une fourchette entre chaque assiette. Pour manger, un philosophe doit utiliser les deux fourchettes adjacentes à son assiette (et celles-là seulement).



Un philosophe peut être :

- penseur : il n'utilise pas de fourchettes ;
- mangeur : il utilise les deux fourchettes adjacentes ; aucun de ses voisins ne peut manger ;
- demandeur : il souhaite manger mais ne dispose pas des deux fourchettes.

Exemple : philosophes et interblocage (2/2)

Risque d'interblocage

Chaque philosophe demande sa fourchette gauche et l'obtient. Puis quand tous ont leur fourchette gauche, chaque philosophe demande sa fourchette droite et se bloque $\Rightarrow$ interblocage

Solutions

Allocation globale : chaque philosophe demande simultanément les deux fourchettes.

Non conservation : quand un philosophe essaye de prendre sa seconde fourchette et qu'elle est déjà prise, il relâche la première et se met en attente sur la seconde.

Classes ordonnées : imposer un ordre sur les fourchettes $\equiv$ tous les philosophes prennent d'abord la gauche puis la droite, sauf un qui prend d'abord droite puis gauche.

Plan

Détection - guérison

Détection

- Construire le graphe d'allocation
- Détecter l'existence d'un cycle

Coûteux → exécution périodique (et non à chaque allocation)

Guérison : Réquisition des ressources allouées

- Fixer des critères de choix de l'activité victime (priorités...)
- Annulation du travail effectué par la(les) activité(s) victime(s)

– coûteux (détection + choix + travail perdu + restauration)
– pas toujours acceptable (systèmes interactifs ou embarqués)
– nécessité de points de reprise pour retour arrière
+ allocation simplifiée et plus de parallélisme qu'avec la prévention

Conclusion

Exclusion mutuelle

- Nécessité de bloc atomique "lire-écrire" (*test-and-set*) ou "écrire plusieurs variables"
- Solution classique : verrous d'exclusion mutuelle

interblocage

Usuellement : inconvénient occasionnel

- $\rightarrow$ laissé à la charge de l'utilisateur / du programmeur
- utilisation de méthodes de prévention simples (p.e. classes ordonnées)
- ou détection empirique (délai de garde) et guérison par choix manuel des victimes