Interblocage

Caractérisation

Méthodes de gestion d'interblocage

Prévention

Évitement

Détection

Récupération

Modèle

Un système est constitué de *ressources*

Ressources classifiées par leur *type* R_i (e.g. memory, I/O device, ...)

Chaque type R_i est disponible en quantité W_i

L'utilisation est divisée en 3 étapes: *request*, *use*, *release*

Université de Montréal

Interblocage avec verrous

Thread1 fait: `request(L1); request(L2);`

Thread2 fait: `request(L2); request(L1);`

L'interblocage a lieu lorsque les deux threads bloquent à mi-chemin

Dépend des choix d'ordonnancement: non-déterministe

Il peut être très difficile d'identifier et de tester les interblocages

Caractérisation

Il faut 4 conditions pour un interblocage:

- Exclusion mutuelle: dépend d'une ressource que l'on ne peut partager
- Hold&wait: un thread tient une ressource et en attend une autre
- Pas de préemption: un ressource n'est libérée que volontairement
- Circularité: il y a un cycle de threads ou chaque thread attend une ressource tenue par le thread suivant

[Circularité implique Hold&wait]

Graphe d'allocation des ressources

Un graphe dirigé bipartite

Chaque thread est représenté par un sommet T_i

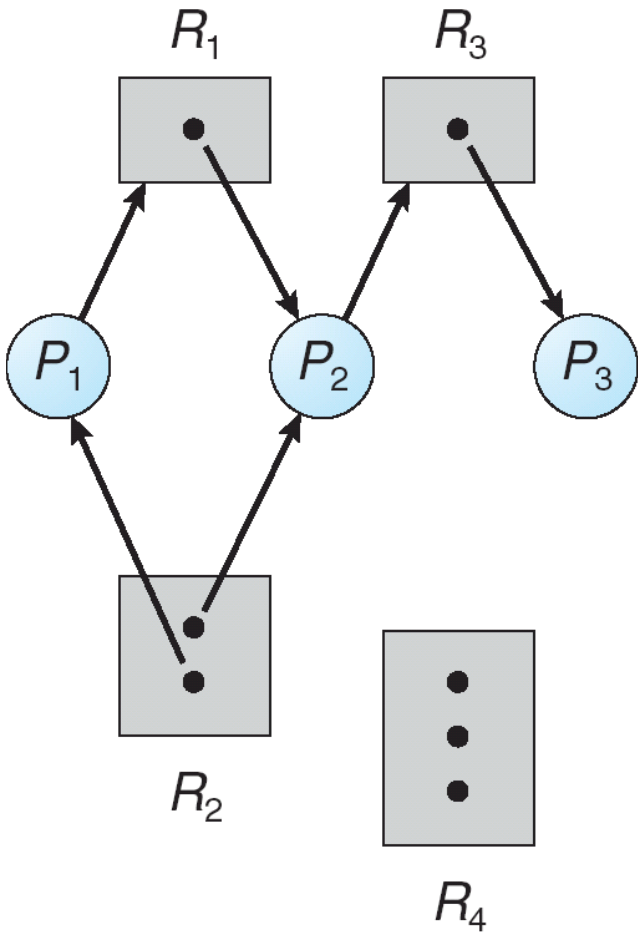
Chaque *type* de ressource est représentée par un sommet R_j

Chaque requête en attente est représentée par un arc $T_i \rightarrow R_j$

Chaque ressource assignée est représentée par un arc $R_j \rightarrow T_i$

Université de Montréal

Exemple de graphe



P_1 tiens une ressource R_2

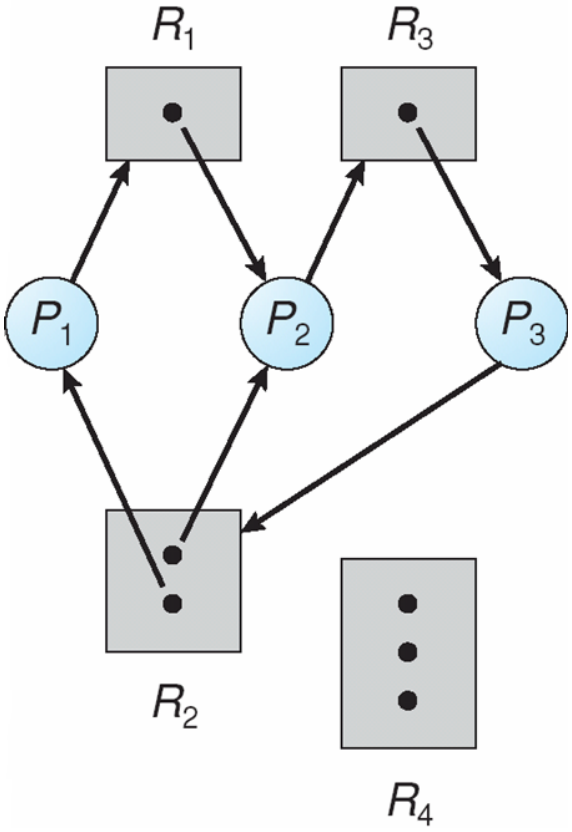
P_1 attend une ressource R_1

...

Pas de cycle

$\Rightarrow$ pas d'interblocage

Exemple de cycle

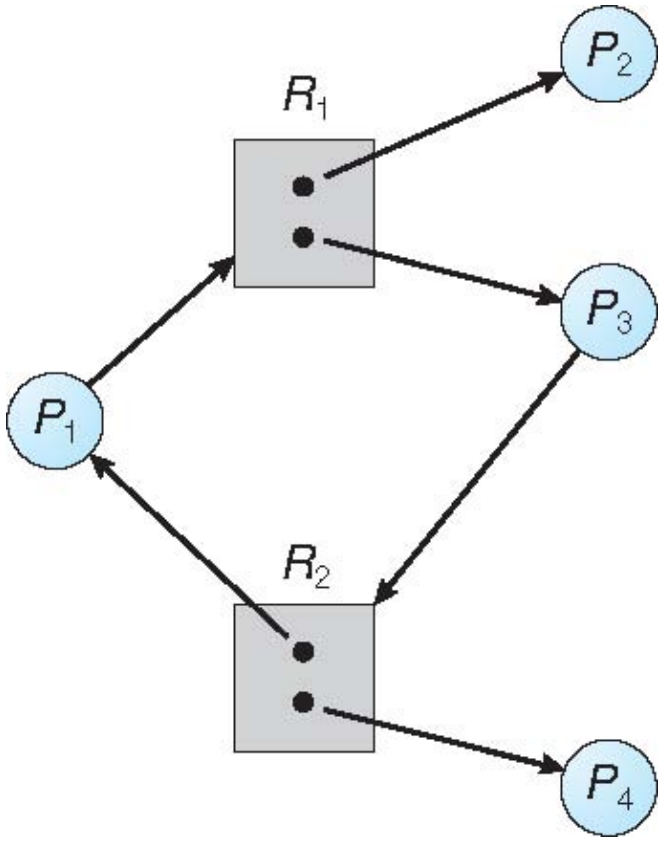


P_3 attend une ressource R_2

$\Rightarrow$ Crée 2 cycles!

Université de Montréal

Exemple de cycle sans interblocage



Un cycle:

$$P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$$

P_2 peut rendre R_1

et P_4 peut rendre R_2

Condition d'interblocage

Pas de cycle $\Rightarrow$ pas d'interblocage

Si une seule ressource par type, alors interblocage *ssi* cycle

Si plusieurs ressources par type, c'est plus compliqué!

Gestion des interblocage

S'assurer qu'un interblocage est *impossible*

- Prévention: impossible de par la conception du système
- Évitement: garder trace de l'état du graphe pour éviter de tomber dans un cycle

Détecter les interblocage et les résoudre par un système de *recovery*

La méthode de l'autruche (a.k.a "Plug'n'Pray")

- Permettre les interblocages
- Ne même pas essayer de les détecter
- Méthode la plus populaire

Prévention d'interblocage

Conception du système élimine une des conditions nécessaire:

- Exclusion mutuelle: peut être éliminée par virtualisation pour certaines ressources, mais pas pour toutes.
- Hold&wait: Obliger à allouer toutes les ressources d'un seul coup
- Préemption: relâcher les ressources tenues quand on en attend d'autres
- Circularité: imposer un *ordre total* entre tous les types de ressources

Évitement d'interblocage

Besoin d'information sur le futur

E.g. chaque thread annonce son usage maximum de ressources

En cours d'exécution, le système vérifie l'*état d'allocation de ressources* pour garantir qu'on ne tombera pas dans un cycle

l'*état d'allocation de ressources* est la quantité de ressources allouées et disponibles, ainsi que les usages maximaux annoncés

Évitement: État sûr

Allocation: le système doit décider si une allocation laisse le système dans un état sûr (*safe*)

L'état est *safe* s'il existe une séquence d'exécution $T_1, T_2, \ldots$ de tous les threads telle que pour chaque T_i, les ressources dont il a besoin seront disponibles quand tous les $T_j, j < i$ auront terminé leur exécution

safe $\Rightarrow$ pas d'interblocage

C'est une condition conservatrice!

Exemple d'analyse d'état sûr

Ressources: 12 chaises

	Maximum	Actuel
P_0	10	5
P_1	4	2
P_2	9	2

Exécution possible: P_1 (max 11, tombe à 7)

$\Rightarrow P_0$ (max 12, tombe à 2)

$\Rightarrow P_2$ (max 9, tombe à 0)

Que se passe-t-il si P_2 demande une chaise de plus?

Algorithmes d'évitement

Pour des ressources uniques: utiliser un algorithme basé sur le graphe d'allocation de ressources.

Pour des ressources multiples: utiliser l'algorithme du banquier

Évitement basé sur le graphe d'allocation

Deux types d'arcs $T_i \rightarrow R_j$:

- *Normaux* pour les requêtes en attente
- *Futurs* pour les requêtes qui pourraient encore arriver

Il faut connaître tous les arcs *futurs* dès le départ

Une ressource est allouée seulement si cela ne crée pas de cycle

Algorithme de complexité O(N^2)

Algorithme du banquier: données

Available$[r]$: quantité de ressource r disponible

Max$[t, r]$: quantité maximum de ressource r utilisée par t

Allocated$[t, r]$: quantité de ressource r allouée à t

$$\textit{Needed}[t, r] = \textit{Max}[t, r] - \textit{Allocated}[t, r]$$

Needed$[t, r]$: quantité de ressource r dont t pourrait encore avoir besoin pour terminer son exécution

Algorithme du banquier

Une nouvelle requête (sous forme d'un vecteur *Requested*) de T

Si *Requested* $>$ *Needed*$[T]$: signaler une erreur!

Si *Requested* $>$ *Available*: T doit attendre!

Sinon, calculer un nouvel état hypothétique:

$$\textit{Available} \mathrel{-}= \textit{Requested}$$

$$\textit{Allocated}[T] \mathrel{+}= \textit{Requested}$$

S'il est *safe*, alors la requête est acceptée immédiatement!

Sinon: T doit attendre!

Université de Montréal

Algorithme du banquier: safe state

```
Work = Available;
Ts = AllThreads;
safe = true;
while (safe && !empty (Ts)) {
  safe = false;
  for (T in Ts) {
    if (Needed[T] <= Work) {
      Ts -= T;
      Work = Work + Allocated[T];
      safe = true;
} } }
return safe;
```

Détection d'interblocage

Maintenir un graphe d'allocation de ressources (ou une version sans les ressources)

Invoquer un algorithme de recherche de cycle dans le graphe

L'algorithme est aussi O(N^2)

Pour le cas multiple, la complexité est multipliée par M

Université de Montréal

Détection d'interblocage: cas multiple

```
Work = Available;
Ts = AllThreads;
safe = true;
while (safe && !empty (Ts)) {
  safe = false;
  for (T in Ts) {
    if (Requesting[T] <= Work) {
      Ts -= T;
      Work = Work + Allocated[T];
      safe = true;
} } }
return safe;
```

Université de Montréal

Détection d'interblocage: usage

Quand lancer l'algorithme de détection?

Dépend de la fréquence attendue et de la réaction espérée

Plus souvent donne un meilleur diagnostic et une réaction plus prompte

Moins souvent coûte moins cher

Seulement en cas de suspicion:

- Usage CPU bas: détecte quand on a rien d'autre à faire
- Processus en attente depuis "longtemps"

Récupération: préemption

Forcer un thread à relâcher une ressource et faire un *rollback*

Seulement si la préemption est possible

Choisir une/des victimes:

- minimiser le coût des *rollback*
- maximiser les chances de ne pas retomber dans un interblocage
- Éviter la *famine*, donc ne pas toujours choisir le même thread

Récupération: terminer les threads

Lorsque possible: demander de l'aide à un opérateur humain

Même problème de choix de victimes

Le coût est différent:

- Priorité du thread?
- Temps de calculs accumulé?
- Temps de calcul encore a faire?
- Quels autres thread dépendent de celui-ci?
- ...