Systèmes et Réseaux (ASR 2) - Notes de cours
Cours 6

Anne Benoit

March 4, 2015

Objectifs: décrire les interblocages, qui empêchent un ensemble de processus concurrent de compléter leur tâche. Présenter des méthodes pour prévenir ou éviter les interblocages.

5.1 Le problème des interblocages

Ensemble de processus concurrents: chacun tient une ressource et attend une ressource tenue par un autre processus de l'ensemble.

Exemple 1: système avec 2 disques, P_1 et P_2 tiennent chacun un disque et attendent le deuxième pour poursuivre leur exécution.

Exemple 2: deux sémaphores A et B initialisés à 1, P_1 fait wait(A); wait(B); et P_2 fait wait(B); wait(A);

Exemple 3: un pont à sens-unique avec deux voitures qui se sont engagées simultanément. Pour résoudre l'interblocage, il faut qu'une voiture recule, peut obliger d'autres voitures à reculer... Possibilité de famine.

Modèle.

- Un ensemble de types de ressources $\{R_1, R_2, \ldots, R_m\}$ (peuvent être des cycles CPU, de l'espace mémoire, des fichiers, des périphériques IO, ...);
- Chaque ressource de type R_i a W_i instances;
- Un ensemble de processus $\{P_1, P_2, \ldots, P_n\}$;

- Chaque processus utilise les ressources en effectuant une requête de ressource (attend si la requête ne peut pas être satisfaite immédiatement), une utilisation, puis le relâchement de la ressource.

Caractérisation des interblocages. On a un interblocage si les 4 conditions suivantes sont vérifiées simultanément:

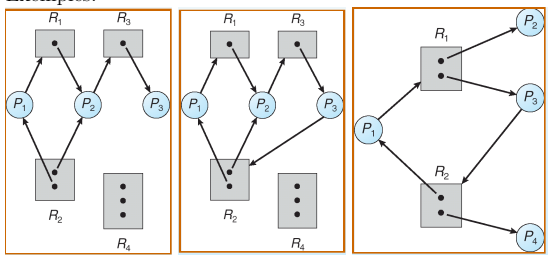
1. Exclusion mutuelle: seulement un processus à la fois peut utiliser une ressource;
2. Tient-et-attend: un processus qui tient au moins une ressource est en attente d'autres ressources;
3. Pas de préemption: une ressource ne peut être relâchée qu'intentionnellement par le processus qui la tient, une fois qu'il a terminé;
4. Attente circulaire: il y a un ensemble $\{P_1, \ldots, P_k\}$ de processus en attente, tels que P_1 attend une ressource tenue par P_2, ..., P_i attend une ressource tenue par P_{i+1}, ... et P_k attend une ressource tenue par P_1.

L'attente circulaire implique le tient-et-attend, les 4 conditions ne sont pas entièrement indépendantes.

Graphe de l'allocation des ressources. Les sommets sont constitués de deux types de noeuds: les processus du système $(P_1, \ldots, P_n)$, et les types de ressources $(R_1, \ldots, R_m)$. Les sommets R_j peuvent être sous-divisés en plusieurs instances de ressources.

Il y a deux types d'arêtes: une arête de **requête** $P_i \to R_j$ si P_i attend la ressource R_j, et une arête d'**assignement** $R_j \to P_i$ si P_i tient la ressource R_j (ou une instance de la ressource).

Exemples:

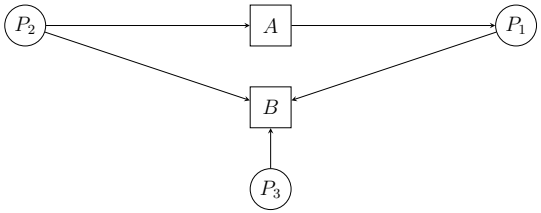


(a) sans interblocage; (b) avec interblocage: il y a un circuit; (c) avec un circuit mais sans interblocage.

- Si le graphe n'a pas de circuits, il n'y a pas d'interblocages.
- Si le graphe a un circuit, avec une seule instance par type de ressource il y a interblocage, et il y a possibilité d'interblocage s'il y a plusieurs instances par type de ressource.

Petit exercice: On considère un système avec 3 processus et 2 ressources: P_2 demande la ressource A, qui est détenue par P_1. De plus, les trois processus demandent la ressource B.

1. Dessiner le graphe d'allocation des ressources correspondant.



2. Commenter l'état du système si la ressource B est donnée au processus P_1, P_2 ou P_3.

   Si B est donnée à P_1, le graphe n'a pas de circuits et l'exécution se termine sans problèmes. En revanche, donner la ressource à P_2 résulte en un circuit $P_2 \to A \to P_1 \to B \to P_2$ et donc en un interblocage. Finalement, si B est donnée à P_3, le graphe n'a pas de circuits. L'exécution pourra se poursuivre si on donne ensuite B à P_1 puis à P_2 (pour ne pas se retrouver dans le cas d'interblocage).

5.2 Méthodes pour gérer les interblocages

Différentes catégories de méthodes:

- S'assurer que le système n'entrera jamais en interblocage: prévenir ou éviter les interblocages;
- Autoriser l'entrée en situation d'interblocage, les détecter et récupérer;
- Ignorer le problème et prétendre qu'il n'y a jamais d'interblocages; utilisé par la plupart des OS, incluant UNIX et Windows!

Souvent, on peut combiner les méthodes (pour différents problèmes d'allocation de ressources du système).

5.2.1 Prévenir des interblocages

Contrainte sur la façon dont les requêtes peuvent être effectuées: s'assurer qu'au moins une des 4 conditions n'est pas vérifiée.

1. Exclusion mutuelle: on peut s'en passer pour les ressources qui peuvent être partagées, mais requis pour les ressources non partageables.

2. Tient-et-attend: il faut s'assurer que quand un processus demande une ressource, il n'en tient pas d'autres: (a) imposer que le processus demande et obtiennent toutes les ressources dont il a besoin avant de commencer son exécution, ou (b) ne permettre de demander une ressource uniquement lorsque le processus n'en tient pas.
   Avec (a), des appels systèmes avant l'exécution pour demander les requêtes, avant tout autre action du processus. Exemple: copier données d'un DVD vers un fichier sur disque, trier le fichier, puis imprimer: il faut réserver le DVD, le disque, et l'imprimante avant de commencer.
   Avec (b), on peut tenir des ressources puis les relâcher avant d'en redemander de nouvelles; pour l'exemple, le processus prend le DVD et le disque, puis le disque et l'imprimante.
   Problème de faible utilisation des ressources, et risque de famine.

3. Pas de préemption: garantir la préemption: si un processus tient des ressources et en demande une autre qui n'est pas disponible, toutes les ressources qu'il tenait lui sont retirées, et ces ressources sont rajoutées à la liste des ressources qu'il attend. Processus redémarré lorsqu'il peut avoir toutes les ressources dont il a besoin. Il faut pouvoir sauvegarder/restaurer facilement l'état d'une ressource.

4. Attente circulaire: Imposer un ordre total sur les types de ressources, et imposer une demande des ressources dans l'ordre croissant. $F(R_i)$: numéro associé à la ressource. Interblocage: P_i tient R_{i-1} et attend R_i, tenu par P_{i+1}. Du coup, on a $F(R_{i-1}) < F(R_i)$, et pour le cycle de k processus, on obtient $F(R_1) < F(R_2) < \cdots < F(R_k) < F(R_1)$, ce qui est absurde. Il faut s'assurer que l'ordre est respecté: témoin qui enregistre l'ordre et envoie des messages d'avertissement.

5.2.2 Eviter les interblocages

Besoin d'information additionnelle, sur la façon dont les ressources vont être demandées.

Modèle (simple et utile): chaque processus déclare le nombre maximum de ressources de chaque type dont il peut avoir besoin. Algorithme: examine l'état d'allocation des ressources (nombre de ressources disponibles et allouées, et demandes max), pour s'assurer qu'il n'y aura jamais de condition d'attente circulaire.

Etat sûr: il existe une séquence $< P_1, \ldots, P_n >$ de tous les processus du système, telle que pour chaque P_i, les ressources que P_i peut encore demander sont soit disponibles immédiatement, soit tenues par les P_j avec $j < i$.

Ainsi, si P_i ne peut pas s'exécuter immédiatement, il peut attendre que tous les P_j, $j < i$, aient terminé, et alors il peut récupérer les ressources dont il a besoin, s'exécuter, et relâcher ses ressources, permettant ainsi à P_{i+1} de s'exécuter.

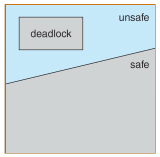
Exemple avec 12 instances d'une ressource.

	Besoin max	Ressources utilisées
P_1	10	5
P_2	4	2
P_3	9	2

L'état est sûr, avec la séquence $< P_2, P_1, P_3 >$.

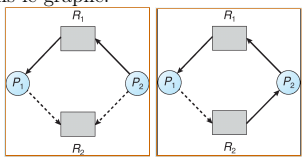
Par contre, si on donne une ressource de plus à P_3, alors on ne peut faire que P_2, puis on a seulement 4 ressources de disponible et on ne peut pas avancer. L'état n'est pas sûr.

Si l'état est sûr, il n'y a pas interblocage. Possibilité d'interblocage si l'état n'est pas sûr. Eviter les interblocages = s'assurer qu'on n'entrera jamais dans un état non sûr.



Avec une seule instance par type de ressource. Utilisation d'un graphe d'allocation des ressources. Arête de *demande* $P_i \to R_j$ qui indique qu'un processus P_i peut avoir besoin de R_j durant son exécution, représentée en pointillés. Devient une arête de *requête* lorsque le processus demande effectivement la ressource. Arête de requête transformée en arête d'*assignement* lorsque la ressource est allouée au processus (arête $R_j \to P_i$). Lorsque la ressource est relâchée, on retourne à l'état avec une arête de demande.

Chaque processus doit donc demander à priori les ressources dont il peut avoir besoin. Exemple où P_2 demande R_2, mais on ne peut pas lui allouer R_2 sinon il y a un circuit dans le graphe.



Algorithme: si P_i demande R_j, la ressource ne lui est allouée que si convertir l'arête de requête en arrête d'assignement ne crée pas de circuit dans le graphe d'allocation des ressources.

Algorithme de recherche de circuit? DFS qui colorie les sommets en blanc, gris, noir. Si on retombe sur un sommet gris (arc arrière), il y a un circuit (cf cours Algo2). Recherche en $O(|V|+|E|)$, où $|V|$ est le nombre de sommets et $|E|$ est le nombre d'arêtes.

Avec plusieurs instances par type de ressource: l'algorithme du banquier. Soit n le nombre de processus et m le nombre de types de ressources. On maintient les structures suivantes:

- *Available* est un vecteur de longueur m: si $Available[j] = k$, il y a k instances de R_j de disponible;
- Max est une matrice $n \times m$: si $Max[i, j] = k$, P_i peut demander au plus k instances de R_j à la fois durant son exécution;
- *Allocation* est une matrice $n \times m$: si $Allocation[i, j] = k$, P_i détient actuellement k instances de R_j;
- *Need* est une matrice $n \times m$: $Need = Max - Allocation$, i.e., si $Need[i, j] = k$, P_i peut demander au plus k instances supplémentaires de R_j pour compléter sa tâche.

Algorithme de sûreté: détermine si un état est sûr. Utilise les vecteurs $Work$ et $Finish$, de longueur m et n. Initialement, $Work = Available$, et $Finish[i] = false$ pour $1 \leq i \leq n$.
Algo: Tant qu'il existe un processus i tel que $Finish[i] = false$ et $Need_i \leq Work$,
$\{Work := Work + Allocation_i$; $Finish[i] := true$; $\}$
Si $Finish[i] = true$ pour $1 \leq i \leq n$, alors l'état est sûr.

Algorithme du banquier: prend en entrée un vecteur de requêtes du processus P_i, $Request_i$. Si $Request_i[j] = k$, alors P_i demande k instances de R_j.
Algo:
si $Request_i > Need_i$, erreur car le processus a demandé plus que sa demande maximum;
si $Request_i > Available_i$, P_i doit attendre car il n'y a pas assez de ressources disponibles;
sinon, effectuer l'allocation des ressources à P_i:
$Available := Available - Request_i$;
$Allocation_i := Allocation_i + Request_i$;
$Need_i := Need_i - Request_i$;
si l'état obtenu est sûr, les ressources sont allouées à P_i;
sinon, P_i doit attendre et on retourne à l'état précédent.

Exemple avec $n = 5$ processus, $m = 3$ types de ressources, A (10 instances), B (5 instances), et C (7 instances).

On a l'état suivant:

	Allocation	*Max*
P_1	0 1 0	7 5 3
P_2	2 0 0	3 2 2
P_3	3 0 2	9 0 2
P_4	2 1 1	2 2 2
P_5	0 0 2	4 3 3

Initialement, $Available = (3\ 3\ 2)$.

1. Donner la matrice *Need*.

	Need
P_1	7 4 3
P_2	1 2 2
P_3	6 0 0
P_4	0 1 1
P_5	4 3 1

2. L'état est-il sûr?

   Oui, on peut terminer immédiatement P_2 et P_4, ce qui donne $Available = (7\ 4\ 3)$, puis tous les autres processus.

Todo: Répondre aux petites questions qui suivent.
Notez que si une requête peut être satisfaite, on considèrera qu'elle l'a été dans les questions qui suivent, sinon la requête est suspendue.

3. P_2 effectue la requête (1 0 2). Peut-elle être satisfaite immédiatement?
4. P_5 effectue la requête (3 3 0). Peut-elle être satisfaite immédiatement?
5. P_1 effectue la requête (0 2 0). Peut-elle être satisfaite immédiatement?