Systèmes et Réseaux (ASR 2) - Notes de cours
Cours 7

Anne Benoit

March 11, 2015

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.

Par la suite, 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?

   Déjà, on a bien $(1\ 0\ 2) \leq Available = (3\ 3\ 2)$. Si on satisfait la requête, $Available = (2\ 3\ 0)$ et le besoin de P_2 devient (0 2 0). On peut alors terminer P_2 et on obtient $Available = (5\ 3\ 2)$, puis P_4 et on obtient $Available = (7\ 4\ 3)$. On est alors dans le même état que dans la question précédente, on peut terminer tous les autres processus. L'état étant sûr, on peut satisfaire la requête.

4. P_5 effectue la requête (3 3 0). Peut-elle être satisfaite immédiatement?

   Non, car on n'a plus assez de ressources disponibles ($Available = (2\ 3\ 0)$ étant donné qu'on a satisfait la requête précédente).

5. P_1 effectue la requête (0 2 0). Peut-elle être satisfaite immédiatement?

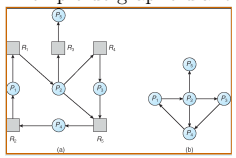
   Non, car on se trouve alors dans un état non sûr: si on satisfait la requête, $Available = (2\ 1\ 0)$ et le besoin de P_1 devient (7 3 3). On ne peut alors terminer aucun processus car $Need_i \leq Available$ pour $1 \leq i \leq 5$.

5.2.3 Détecter les interblocages

Idée: permettre de rentrer dans une situation d'interblocage: algorithme pour détecter un interblocage et méthode pour récupérer.

Avec une seule instance par type de ressources. Variante du graphe d'allocation des ressources: graphe d'attente. Les noeuds sont les processus, et on a un arc $P_i \rightarrow P_j$ si P_i attend que P_j relâche une ressource ($P_i \rightarrow R_q$ et $R_q \rightarrow P_j$).

Exemple de graphe d'allocation des ressources (a) et de graphe d'attente (b):



On invoque périodiquement un algorithme qui cherche un circuit dans le graphe. S'il y a un circuit, il y a un interblocage.

Avec plusieurs instances. Algorithme similaire à celui du banquier, avec le vecteur *Available* et la matrice *Allocation*. En plus, matrice $n \times m$ *Request*, qui indique la requête courante de chaque processus.

Algorithme de détection: utilise les vecteurs $Work$ et $Finish$, initialisés à $Work = Available$, et pour $1 \leq i \leq n$, si $Allocation_i \neq 0$, alors $Finish[i] = false$, sinon $Finish[i] = true$.
Tant qu'il existe un processus i tel que $Finish[i] = false$ et $Request_i \leq Work$,
$\{Work := Work + Allocation_i; Finish[i] := true; \}$
S'il existe P_i $(1 \leq i \leq n)$ tel que $Finish[i] = false$, alors il y a un interblocage. Chaque processus tel que $Finish[i] = false$ est en interblocage.

Exemple avec $n = 5$ processus, $m = 3$ types de ressources, A (7 instances), B (2 instances), et C (6 instances).

On a l'état suivant:

	Allocation	*Request*
P_1	0 1 0	0 0 0
P_2	2 0 0	2 0 1
P_3	3 0 3	0 0 0
P_4	2 1 1	1 0 0
P_5	0 0 2	0 0 2

Initialement, *Available* = (0 0 0).

1. Est-ce qu'il y a un interblocage?
2. Et si P_3 demande une instance supplémentaire de la ressource C?

Usage de l'algorithme de détection. Quand, et à quelle fréquence invoquer l'algorithme de détection? Dépend de la fréquence à laquelle on attend des interblocages, et du nombre de processus qui vont devoir reprendre leur exécution à un point antérieur (un processus par circuit disjoint dans le graphe).

5.2.4 Récupérer d'un interblocage

Solutions: terminer tous les processus en interblocage, ou un processus à la fois jusqu'à ce que l'interblocage soit éliminé. Dans quel ordre terminer les processus?

- priorité du processus;
- temps passé à calculer et temps restant jusqu'à complétion;
- ressources utilisées par le processus et besoin en ressources jusqu'à complétion;
- nombre de processus qu'on va devoir terminer;
- processus interactif ou batch.

On veut choisir une victime afin de minimiser le coût. Rollback: on retourne à un état sûr, et on redémarre le processus à partir de cet état. Problème de famine: le même processus peut être toujours choisi comme victime, rajouter le nombre de terminaisons dans le facteur coût.

5.3 Conclusion

- Interblocage: deux processus (ou plus) attendent indéfiniment un événement qui peut être produit uniquement par un bloqué. Approches:
  - Prévenir ou éviter les interblocages: il n'y en a jamais;
  - Permettre de détecter les interblocages et de s'en sortir;
  - Ignorer le problème.
- 4 conditions nécessaires: exclusion mutuelle, tient-et-attend, pas de préemption, attente circulaire.
  - Prévenir: empêcher une de ces conditions;
  - Eviter: information à priori sur l'utilisation des ressources;
  - Détecter: déterminer s'il y a interblocage;
  - Récupérer: terminer des processus ou préempter des ressources.
- Aucune approche n'est suffisante par elle-même.

Petit problème d'application de l'algorithme du banquier. On considère le système suivant, avec 5 processus et 4 ressources, avec plusieurs instances de chaque ressource (3 14 12 12):

	Allocation	*Max*
P_1	0 0 1 2	0 0 1 2
P_2	1 0 0 0	2 7 5 0
P_3	1 3 5 4	2 3 5 6
P_4	0 6 3 2	0 6 5 2
P_5	0 0 1 4	0 6 5 6

1. Donner la matrice $Need$ et le vecteur *Available*.
2. L'état est-il sûr?

Par la suite, si une requête peut être satisfaite, on considèrera qu'elle l'a été dans les questions qui suivent, sinon la requête est suspendue.

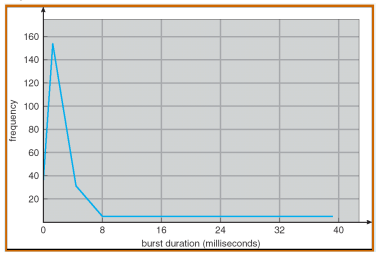
3. P_2 effectue la requête (0 4 2 0). Peut-elle être satisfaite immédiatement?
4. P_3 effectue la requête (1 0 0 1). Peut-elle être satisfaite immédiatement?
5. P_2 effectue la requête (1 0 0 0). Peut-elle être satisfaite immédiatement?

6 Ordonnancement des processus

Concepts de base, critères et algorithmes d'ordonnancement, ordonnancement multiprocesseur ou temps réel, exemples d'OS, évaluation des algorithmes.

6.1 Concepts de base

Multiprogrammation pour bien exploiter le CPU: l'execution d'un processus consiste en des cycles d'utilisation CPU et d'attente sur I/O. On parle de *CPU burst*.



Loi exponentielle / hyper-exponentielle: beaucoup de bursts courts, et très peu de bursts longs.

Ordonnanceur: choisit un processus parmi ceux en mémoire qui sont prêts à être exécutés, et lui donne le CPU. Décisions à prendre lorsqu'un processus:

1. passe de l'état running à l'état waiting;
2. passe de l'état running à l'état ready;
3. passe de l'état waiting à l'état ready;
4. se termine.

Ordonnancement coopératif, non-préemptif s'il n'y a que 1 et 4. Sinon, ordonnancement préemptif.

Dispatcher: module qui donne le CPU au processus choisi par l'ordonnanceur court-terme, ce qui implique de changer de contexte, passer en mode utilisateur, et aller au bon endroit dans le programme utilisateur pour redémarrer ce programme.

Doit être très rapide: invoqué à chaque changement de processus. Latence: temps mis pour arrêter un processus et en charger un nouveau.

6.2 Critères et algorithmes d'ordonnancement

6.2.1 Critères

Utilisation CPU – Maintenir le CPU le plus utilisé possible.

Débit – (*throughput*) C'est le nombre de processus qui terminent leur exécution par unité de temps (1 par heure, ou 10 par seconde?).

Temps de retournement – (*turnaround time*) Temps nécessaire pour exécuter un processus (depuis la soumission jusqu'à la terminaison).

Temps d'attente – (*waiting time*) Temps qu'un processus passe dans la file *ready*.

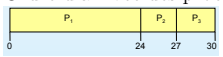
Temps de réponse – (*response time*) Temps entre la soumission d'une requête et la première réponse (et non la sortie finale).

On cherche à maximiser l'utilisation du CPU et le débit, et à minimiser les temps de retournement, attente et réponse. On peut chercher à optimiser la moyenne ou bien le minimum/maximum.

6.2.2 Ordonnancement First-Come, First-Served (FCFS)

Processus	Temps de *burst*
P_1	24
P_2	3
P_3	3

Ordre d'arrivée des processus: P_1, P_2, P_3. L'exécution est:

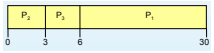


Temps d'attente:

P_1	0
P_2	24
P_3	27

ce qui fait un temps d'attente moyen de $\frac{0+24+27}{3} = 17$.

Si l'ordre d'arrivée est P_2, P_3, P_1, on obtient



Temps d'attente:

P_1	6
P_2	0
P_3	3

ce qui fait un temps d'attente moyen de $\frac{6+0+3}{3} = 3$, bien mieux que l'exécution précédente!

Exemple avec n processus gourmands en I/O, et un gourmand en CPU: si le gourmand en CPU s'exécute en premier, les autres doivent attendre longtemps avant de pouvoir s'exécuter. On parle de l'effet de convoi (*convoy effect*).

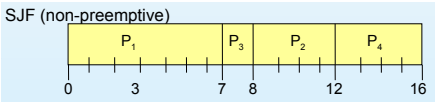
6.2.3 Ordonnancement Shortest-Job-First (SJF)

A chaque processus, on associe la longueur de son prochain burst CPU, et on choisit le processus avec le temps le plus court. Deux versions:

1. **Non préemptif**. Une fois le CPU donné à un processus, on ne peut pas le préempter avant qu'il ait terminé son CPU burst.

Exemple:

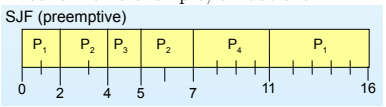
Processus	Temps d'arrivée	Temps de *burst*
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4



Temps d'attente moyen: $\frac{0+6+3+7}{4} = 4$.

2. **Préemptif**. Si un nouveau processus arrive avec un temps d'exécution plus court que le temps restant au processus en cours d'exécution, on lui donne le CPU. *Shortest-Remaining-Time-First (SRTF).*

Avec le même exemple, on obtient



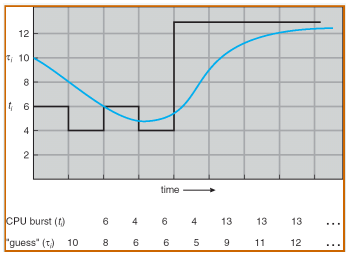
Temps d'attente moyen: $\frac{9+1+0+2}{4} = 3$.

SJF est optimal (dans sa version préemptive, ou avec tous les temps d'arrivée à 0) pour minimiser le temps d'attente moyen.
Si l'on fait passer un processus court devant un plus long, cela diminue le temps d'attente du court plus que l'augmentation du temps d'attente du long. Ainsi, le temps d'attente moyen diminue.
Exemple non préemptif: P_1 arrive au temps 0 et dure 10, P_2 arrive au temps 2 et dure 2. SJF exécute d'abord P_1 puis P_2, ce qui donne un temps d'attente de $8/2 = 4$, alors que l'exécution P_2 puis P_1 donne un temps d'attente de $4/2 = 2$.

Problèmes de SJF: il faut connaître la longueur du prochain burst CPU! S'il n'est pas donné par l'utilisateur (limite sur le temps d'exécution), il faut prédire cette longueur. Utilisation du moyennage exponentiel (*exponential averaging*):

- α est tel que $0 \leq \alpha \leq 1$;
- t_n est la longueur du burst n;
- τ_{n+1} est la valeur prédite du prochain burst $n+1$, et $\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$.

Exemple avec $\alpha = \frac{1}{2}$ et $\tau_0 = 10$.



Si $\alpha = 0$, l'histoire récente ne compte pas (i.e., $\tau_{n+1} = \tau_n$), alors que si $\alpha = 1$, seul le dernier CPU burst compte (i.e., $\tau_{n+1} = t_n$). On peut étendre la formule:

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\alpha t_{n-1} + \cdots + (1-\alpha)^j \alpha t_{n-j} + \cdots + (1-\alpha)^{n+1}\tau_0.$$

α et $1-\alpha$ sont plus petits que 1, et donc les termes ont de moins en moins de poids.

6.2.4 Ordonnancement avec priorité

Associer une priorité à chaque processus, et donner le CPU au processus le plus prioritaire (petit entier = haute priorité). Version préemptive ou non.

SJF: algorithme avec priorité, où la priorité est l'estimation de la longueur du prochain burst CPU.

Problème de famine: un processus de faible priorité peut ne jamais s'exécuter.

Solution: *aging* (vieillissement): augmenter la priorité d'un processus avec le temps.

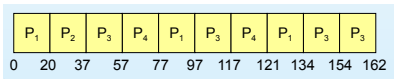
6.2.5 Ordonnancement Round-Robin (RR)

Chaque processus obtient le CPU pour une courte durée (*time quantum*), usuellement entre 10 et 100 millisecondes. Ensuite, le processus est préempté et remis dans la file ready.

Ainsi, s'il y a n processus ready et le time quantum est q, chaque processus obtient $1/n$-ème du temps CPU en tranches d'au plus q à la fois. Ainsi, chaque processus n'attend jamais plus de $(n-1)q$ unités de temps.

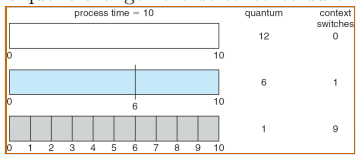
Exemple avec $q = 20$ et 4 processus:

Processus	Temps de *burst*
P_1	53
P_2	17
P_3	68
P_4	24



Le temps de turnaround est souvent plus élevé que celui de SJF, mais meilleur temps de réponse.

Si q est très grand, on se retrouve avec du FCFS. Si q est petit, on a un trop grand surcoût lié aux changements de contexte. Typiquement, q entre 10 et 100 milli-secondes, alors que le changement de contexte dure moins de 10 micro-secondes.

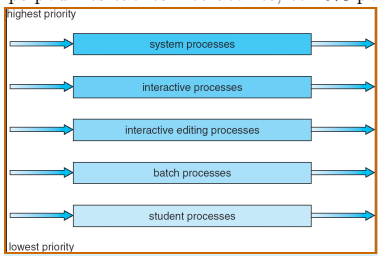


Le temps de turnaround moyen est plus petit si la plupart des processus terminent leur CPU burst durant un q; bon résultats si 80% des CPU burst sont plus petits que q.

6.2.6 Files à plusieurs niveaux

Partager la file ready en plusieurs files distinctes, et chaque file a son propre algorithme d'ordonnancement. Exemple: tâches interactives ordonnancées en RR, et tâches batch en FCFS.

Ordonnancement entre les files: à priorité fixée (on sert d'abord les tâches de la file la plus prioritaire; risque de famine) ou avec un partage de temps (par exemple, 80% du temps pour les tâches interactives, et 20% pour les batch).



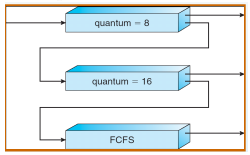
6.2.7 File avec retour d'information

Permet de déplacer un processus d'une file à une autre, ce qui permet d'implémenter le vieillissement.

Ordonnanceur défini par

- le nombre de files;
- l'algorithme d'ordonnancement de chaque file;
- méthode pour décider quand promouvoir un processus;
- méthode pour décider quand rétrograder un processus;
- méthode pour déterminer dans quelle file faire rentrer un nouveau processus.

Exemple avec 3 files, deux avec RR ($q = 8$ ou $q = 16$) et une avec FCFS:



Entrée dans la file avec $q = 8$, s'il ne termine pas en 8 millisecondes, il passe dans la file avec $q = 16$. S'il ne termine pas avec les 16 millisecondes de plus, il passe dans la file FCFS.

6.3 Ordonnancement multi-processeur ou temps réel

L'ordonnancement est plus compliqué en multi-processeur, il faut équilibrer la charge entre les processeurs. Besoin de synchronisation entre processeurs.

Temps réel: il faut terminer des tâches critiques avant des dates fixées d'avance (*hard*), ou alors donner une plus grande priorité aux tâches critiques (*soft*).

6.4 Exemples d'OS

Solaris 2. Trois catégories:

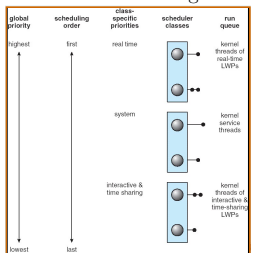


Table de dispatch pour les threads interactifs et à partage de temps. Les processus de forte priorité ont un plus petit q, et une nouvelle priorité plus basse à expiration de q (3ème colonne). Après une I/O, plus grande priorité (4ème colonne).

	priority	time quantum	time quantum expired	return from sleep
(low)	0	200	0	50
	5	200	0	50
	10	160	0	51
	15	160	5	51
	20	120	10	52
	25	120	15	52
	30	80	20	53
	35	80	25	54
	40	40	30	55
	45	40	35	56
	50	40	40	58
	55	40	45	58
(high)	59	20	49	59

Windows XP. Basé sur la priorité, algo préemptif et RR. Listes parcourues par ordre de priorité, l'ordonnanceur cherche le thread de plus haute priorité qui est ready.

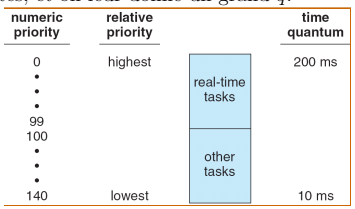
Priorités: classe temps-réel, entre 16 et 31, et les autres ont des priorités entre 1 et 15. Comme avec Solaris, priorité plus basse à expiration d'un time quantum, et priorité augmentée au retour d'une I/O. (Priorité 0 réservée pour la gestion mémoire).

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

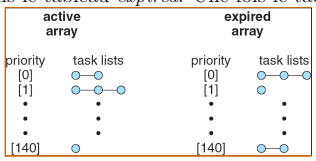
Linux. Deux classes de priorité, pour le partage de temps (mise à jour des priorités comme précédemment) et le temps réel (priorités statiques).

Ordonnancement basé sur des crédits: le processus avec le plus de crédits est ordonnancé; on enlève des crédits lors d'une interruption du timer, et si le crédit atteint 0, on choisit un autre processus. Lorsque tous les processus ont un crédit de 0, on redonne des crédits (basé sur la priorité et l'historique).

Contrairement à précédemment, les petits numéros correspondent aux grandes priorités, et on leur donne un grand q:



On a une liste de tâches par priorité, et à la fin de la tranche de temps, la tâche va dans le tableau *expired*. Une fois le tableau *active* vide, on les échange.



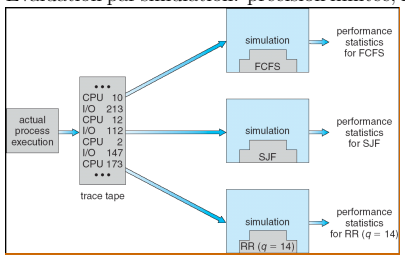
6.5 Evaluation des algorithmes

Différents critères, évoqués précédemment: utilisation CPU, temps de réponse, débit, ...

Modélisation déterministe: pour un workload donné, comparer la performance des algorithmes (comme vu sur exemples).

Modèles de files d'attente: distribution des burst CPU et I/O, et distribution des temps d'arrivée. Utilisation de la formule de Little: si la file a une longueur n, le temps d'attente moyen est W, et le taux moyen d'arrivée est λ, alors $n = \lambda \times W$. Exemple avec λ= 3 processus/sec: pendant une attente de $W = 1$ sec, 3 processus arrivent. A l'état stationnaire, il y a autant d'entrées que de sorties, d'où $n = \lambda \times W$.

Evaluation par simulation: précision limitée, mais plus facile qu'une vraie implémentation.



6.6 Conclusion

- Ordonnancement: choisir un processus prêt et lui donner du temps CPU.
- FCFS: algo simple mais des processus courts peuvent attendre longtemps. SJF: optimal mais difficile de prédire la longueur d'un burst. Cas spécial d'algos avec priorités. Attention à éviter la famine (faire vieillir les processus).
- RR: algo préemptif pour systèmes interactifs.
- Algos à plusieurs niveaux de files d'attentes, combinant par exemple RR et FCFS.
- Méthodes pour évaluer ces algorithmes.