TD 1 - LIF12 système d'exploitation

Passage d'informations sur les pipes

5 avril 2017

I Ordonancement temps réel

I.1 Explication

Lorsqu'on étudie le cas du temps réel, on considère généralement des tâches périodiques dont on connaît la durée. Le but de l'ordonnancement est alors d'assurer que toutes les tâches pourront se terminer avant le retour de la tâche.

L'une des manières d'assurer l'ordonnancement est de donner une priorité fixe aux tâches en fonction de la fréquence (l'inverse de la période). On exécute en priorité la tâche la plus fréquente. Ce n'est pas la manière la plus efficace mais :

— Cela est possible dans beaucoup de systèmes (il suffit qu'ils permettent un ordonnancement à priorité fixe).

— Il existe une condition suffisante simple qui permet d'assurer l'ordonnancement ; ce dernier est possible si :

$$U \quad = \quad \sum_{i=1}^{n} \frac{D_i}{T_i} \quad \leq \quad n(\sqrt[n]{2} - 1)$$

où n est le nombre de tâches, D_i la durée de la tâche i et T_i sa période. On appelle U le taux d'utilisation.

De plus,

— Si $n \to +\infty$, $n(\sqrt[n]{2} - 1) \to \ln 2 \simeq 0.69314\ldots$.

— $\ln(2) \leq n(\sqrt[n]{2} - 1) \leq 1$

I.2 Exercice

Soient les tâches suivantes :

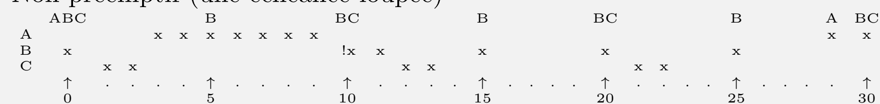
Tâche	Date d'arrivée	Période	Durée	Échéance
A	0	29	7	Fin de la période
B	0	5	1	Fin de la période
C	0	10	2	Fin de la période

1. Calculez le taux d'utilisation pour RM. Le jeu de tâches est-il ordonnançable ?

   $U = 0.64138 \leq 0.77976$. Le jeu de tâche est ordonnançable.

2. Dessinez les 30 premières unités de temps de l'ordonnancement généré par RM. D'abord en version préemptive, puis en version non préemptive. Que constatez-vous ?

   Non préemptif (une échéance loupée)



Préemptif ça marche

```
  ABC            B              BC             B              BC             B        A     BC
A          x  x     x  x  x  x           x                                            x
B x              x              x              x              x              x              x
C    x  x                          x  x                          x  x
  ↑  .  .  .  .  ↑  .  .  .  .  ↑  .  .  .  .  ↑  .  .  .  .  ↑  .  .  .  .  ↑  .  .  .  .  ↑
  0              5              10             15             20             25             30
```

3. Modifions le jeu de tâches :

Tâche	Date d'arrivée	Période	Durée	Échéance
A	0	30	6	Fin de la période
B	0	5	3	Fin de la période
C	0	10	2	Fin de la période

Dans cet exemple, le jeu de tâches est dit harmonique car chaque période est multiple des autres périodes. Calculez le taux d'utilisation, qu'en déduisez-vous ?

$U = 1 > 0.77976$. Le jeu de tâche n'est peut-être pas ordonnançable.

4. Dessinez de nouveau l'ordonnancement RM (mode préemptif). Que constatez-vous ?

```
  ABC            B              BC             B              BC             B              ABC
A                        x  x                          x  x                          x  x
B x  x  x        x  x  x        x  x  x        x  x  x        x  x  x        x  x  x        x
C          x  x                          x  x                          x  x
  ↑  .  .  .  .  ↑  .  .  .  .  ↑  .  .  .  .  ↑  .  .  .  .  ↑  .  .  .  .  ↑  .  .  .  .  ↑
  0              5              10             15             20             25             30
```

Pourtant, oui. Le processeur est utilisé au mieux. Comme le jeux de tâche est harmonique, on se retrouve au bout de l'instant 30 en même situation qu'au début. Il n'y aura donc jamais de problème.

5. Calculez le temps de réponse de chaque tâche.

— A 30
— B 3
— C 5

II Pourquoi faire un programme multithread

On souhaite comparer l'efficacité d'un serveur de fichiers en mode mono ou multithread, même sur un ordinateur disposant d'un seul processeur monocoeur.

L'intérêt du multithread se trouve uniquement lors des accès disque car le thread qui demande l'accès à un fichier doit attendre pendant que les données sont lues sur le disque. Le serveur de fichiers dispose d'un cache en mémoire pour les fichiers les plus couramment lus. On suppose :

— la durée pour traiter une requête sans accès disque est de 15ms (récupérer la requête, chercher dans le cache, rendre le résultat) ;
— pour gérer le multithreading un surcoût de 5ms est nécessaire (changement de contexte et passage en mode noyau pour passer d'un thread à l'autre) ;
— si le fichier ne se trouve pas en cache il faut 75ms supplémentaires pour la lecture sur le disque ;
— en moyenne un fichier est disponible dans le cache dans 2/3 des cas.

Q.II.1) - Donner le nombre de requêtes traitées par seconde pour un serveur monothread

Pour un serveur monothread, dans 2/3 des cas le fichier est dans le cache et demande 15ms de traitement ; dans 1/3 des cas le fichier n'est pas en cache, il faut donc 75ms+15ms pour traiter la requête. En moyenne, il faut donc pour traiter une requête : 2/3*15 + 1/3*90 = 40ms. Le nombre de requêtes traitées par secondes est donc 1000/40 = 25.

Q.II.2) - Donner le nombre de requêtes traitées par seconde pour un serveur multithread utilisant des threads noyaux.

Dans le cas du multithreading, si on suppose que les requêtes sont bien réparties, on permet au processeur et au disque de travailler en parallèle donc on doit calculer séparément ce dont sont capable le processeur et le disque :

— Le processeur :
Dans 2/3 des cas le fichier est en cache, et la requête demande 15+5 = 20ms. Dans 1/3 des cas, le fichier n'est pas dans le cache, mais il est chargé en parallèle des traitements d'autres requêtes : dans le 1/3 des cas restants, il faut donc également 20ms pour traiter ces requêtes. En moyenne, cela donne donc une requête traitée toutes les 20ms, soit 50 requêtes traitée par seconde.

— Le disque :
Dans 1/3 des cas le temps nécessaire au disque est 75ms dans les autres cas c'est 0. Donc en moyenne c'est 25ms. donc le disque est capable de traiter 40 requêtes par seconde.

Au final, le plus lent impose son rythme donc le serveur peut traiter 40 requêtes par seconde.

Q.II.3) - Est-il intéressant d'utiliser des threads utilisateurs (green threads) ?

Faire rappel sur les threads utilisateurs, perf noyau et modèle programmation (N-M).
Les threads utilisateurs sont plus rapidement gérés (pas de surcoût pour la gestion des threads, ou très peu, car tout se passe en mode utilisateur sans changement de contexte). On peut donc enlever les 5ms. Par contre, le noyau ne voit pas le fait que le processus est multithread donc l'accès disque est bloquant. On ne gagne pas les 75ms d'accès disque. Donc le temps est identique à celui du serveur monothread, mais le programme est plus compliqué car c'est le programmeur qui doit gérer la file d'attente des connexion (via les threads) au lieu de laisser faire le système (via l'API des sockets).

III Ordonancement posix (CCF avril 2010)

Nous allons utiliser l'implantation POSIX 1003b sur Linux. Il existe 100 niveaux de priorité :

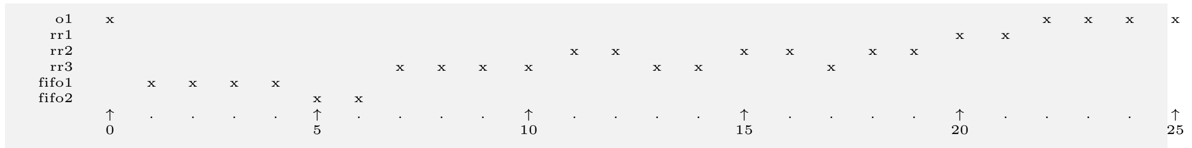
— Le niveau 0 est réservé à SCHED_OTHER et les niveaux de priorité 1 à 99 aux politiques SCHED_FIFO et SCHED_RR.
— Les tâches de priorité 99 sont les tâches de plus forte priorité.
— SCHED_OTHER est dédié à l'ordonnanceur temps partagé.
— Le quantum utilisé par la politique SCHED_RR est de 2 unités de temps.
— Pour SCHED_FIFO et SCHED_RR, lorsqu'une nouvelle tâche arrive elle est placée en tête.
— Pour simplifier, nous supposerons que la politique SCHED_OTHER alloue le processeur de façon inversement proportionnelle au temps processeur déjà consommé par les tâches.
— L'ordonnancement est préemptif.

Soit le jeu de tâches apériodiques suivant :

Tâche	Date d'arrivée	Priorité	Durée	Politique
o1	0	0	5	SCHED_OTHER
rr1	7	5	2	SCHED_RR
rr2	10	10	6	SCHED_RR
rr3	6	10	7	SCHED_RR
fifo1	1	10	4	SCHED_FIFO
fifo2	3	10	2	SCHED_FIFO

Q.III.1) - On suppose qu'une fois arrivées, les tâches sont toujours prêtes. Dessinez de l'instant 0 à l'instant 26, l'ordonnancement généré par l'ordonnanceur.

Q.III.2) - Donner le temps réponse de chaque tâche.



IV Le pont de Miralonde

Le pont de Miralonde est trop étroit pour que 2 voitures puissent se croiser. Vous devez mettre en place un système qui évitera tout incident. Le système se déclenchera automatiquement à l'arrivée d'une voiture à l'une des extrémités du pont. Il autorisera ou non le passage en fonction de la configuration sachant que :

- Si le pont est vide, la première voiture qui arrive peut passer.
- Si le pont contient une voiture qui va du nord au sud, seules les voitures circulant dans le même sens (donc arrivant au nord) peuvent passer.
- Inversement, si le pont contient une voiture qui va du sud au nord, seules les voitures arrivant au sud ont l'autorisation de passer.

Pour éviter tout problème, votre système dispose de verrous (mutex) dont le blocage est atomique. Vous pouvez les bloquer avec la fonction Lock et les débloquer avec UnLock. À chaque fois qu'une voiture arrive à l'extrémité nord du pont, le système appelle automatiquement la fonction EntreeNS() puis lorsque cette voiture sort du pont, la fonction SortieNS() est appelée. Inversement, les fonctions EntreeSN() et SortieSN() servent à gérer les voitures qui circulent dans l'autre sens.

Nous supposerons que si la fonction EntreeNS() (ou EntreeSN()) se termine, le véhicule est autorisé à passer, alors que si elle bloque, le véhicule est aussi bloqué.

Q.IV.1) - Donnez l'algorithme des fonctions EntreeNS(), EntreeSN(), SortieNS() et SortieSN().

Cela ressemble au lecteur rédacteur (2 classes de threads qui ne se bloquent pas entre eux) mais les rôles des 2 classes sont équivalents. On utilise donc une solution similaire.

Il faut 3 mutex :

- M_N qui sert à bloquer ceux qui arrivent au nord et assure que nb_N est bien calculé.
- M_S qui sert à bloquer ceux qui arrivent au sud et assure que nb_S est bien calculé.
- M_o sert à assurer l'équité et empêche la famine. De plus, il assure qu'il n'y a qu'un véhicule dans l'une des fonctions d'entrée.

entreNS

début

- Lock(M_o)
- Lock(M_N)
- nb_N++
- **si** nb_N == *1* **alors**
  - Lock(M_S)
- UnLock(M_N) LaissePassageNord() // fonction qui leve la barrière au nord UnLock(M_o)

fin

sortieNS

début

- Lock(M_N)
- nb_N–
- **si** nb_N == *0* **alors**
  - UnLock(M_S)
- UnLock(M_N)

fin

Les fonctions de circulation dans l'autre sens sont symétriques

entreSN

début

Lock(M_o)

Lock(M_S)

nb_S++

si nb_S == *1* **alors**

Lock(M_N)

UnLock(M_S) LaissePassageSud() // fonction qui lève la barrière au sud UnLock(M_o)

fin

sortieSN

début

Lock(M_S)

nb_S–

si nb_S == *0* **alors**

UnLock(M_N)

UnLock(M_S)

fin

Cela fait ce qu'on demande car :

- La première voiture arrivant au nord bloque celles arrivant au sud.
- La dernière voiture sortant au sud débloque celle qui arrivent au sud.
- Par symétrie, l'inverse est vrai

Il n'y a pas d'interblocage car :

- Supposons une voiture bloquée dans Lock(M_S) de EntreeNS. Elle ne peut pas l'être par une voiture dans entreSN ou entreeNS grâce au mutex M_o. Elle ne peut pas être bloquée par une voiture allant du nord au sud car alors $nb_N > 1$. Donc elle l'est par une voiture dans sortieSN qui ne peut pas être bloquée et liberera le passage.
- Par symétrie, c'est identique pour Lock(M_N) de EntreeSN.
- Supposons une voiture bloquée dans Lock(M_N) de EntreeNS. Elle ne peux pas l'être par une voiture dans EntreeNS grâce au mutex M_o. Elle peut l'être par une voiture allant du sud au nord. Mais dans ce cas, elle est seule à être dans EntreeNS donc rien n'empêchera la voiture allant du sud au nord de libérer le passage. Elle peut aussi être bloquée par une voiture dans sortieNS qui ne fait que des unlock.
- Par symétrie, c'est identique pour Lock(M_S) de EntreeSN.
- Supposons une voiture bloquée par Lock(M_N) de SortieNS. elle l'est que par une voiture dans EntreeNS ou SortieNS. Dans ce cas, cette dernière ne fait que des unlock avant de libérer le passage.
- Par symétrie, c'est identique pour Lock(M_S) de SortieSN.
- Un voiture bloquée sur Lock(M_o) l'est par une voiture dans EntreeSN ou EntreeNS. Dans ce cas, comme les voiture ne peuvent être bloquées indéfiniment, le passage sera libéré.