Systèmes et Réseaux (ASR 2) - Notes de cours
Cours 4

Anne Benoit

February 11, 2015

1 Structure des systèmes d'exploitation

2 Gestion des processus

3 Les threads

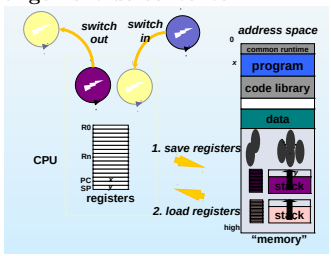
3.1 Threads vs processus

3.2 Threads utilisateurs/noyaux

3.3 Problèmes spécifiques aux threads

3.4 Implémentation des threads

Changement de contexte :



1. Le thread rend le contrôle à l'OS:
   - Evénement volontaire, par exemple un appel à Yield, ou bien appel système, ou en attente de synchronisation;
   - Evénement involontaire: interruption matérielle (gestion des interruptions du CPU), l'OS peut forcer un Yield lorsqu'il récupère le contrôle suite à une interruption.

   - Rappel: interruptions de timer pour le partage de temps.
2. Choix du prochain thread (par l'OS): on attend si pas de thread prêt, cf chapitre ordo pour les différentes politiques de choix.
3. L'OS sauvegarde l'état du thread courant: dans son TCB! Pas si facile, par exemple pour sauvegarder le PC:

   ```
   100 store PC in TCB
   101 switch to next thread
   ```

   Lorsqu'on restaure le thread, re-execute l'instruction 100! Il faut sauver l'adresse 102...
4. L'OS charge le prochain thread: récupérer son TCB en mémoire, charger les registres et le SP (la pile est déjà en mémoire).
5. L'OS exécute le thread suivant: aller au PC.

Exemple:

```
ChgtContexte (tcb1, tcb2)
   sauver regs dans tcb1
   charger regs de tcb2
   // maintenant, SP pointe vers la pile de tcb2
   aller à tcb2.pc

yield x (x=1,2)
   print "start yield Tx"
   ChgtContexte (tcbx, tcby)
   print "end yield Tx"

Thread x (x=1,2)
   print "start Tx"
   yieldx()
   print "end Tx"
```

On obtient: start T1, start yield T1, start T2, start yield T2, end yield T1, end T1, end yield T2, end T2 (pas de préemption, T1 chargé en premier).

Création d'un nouveau thread, ou "fork" d'un thread:

1. Allouer et initialiser un nouveau TCB;
2. Allouer une nouvelle pile;
3. Faire comme si le thread allait appeler une fonction: PC pointe vers la première instruction, SP pointe vers la nouvelle pile, la pile contient les arguments passés à la fonction;

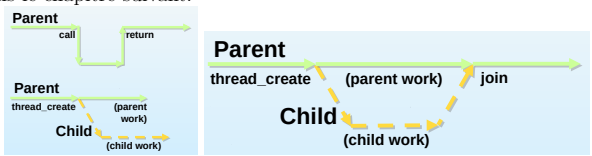
4. Ajouter le thread à la file "ready".

Comment implémenter un join?

```
Child: print "child works"

Parent: create child thread
            print "parent works"
            yield()
            print "parent continues"
```

En sortie, on veut avoir "parent works, child works, parent continues". Peut se produire si child est choisi pour l'exécution après le yield, mais pas de raison! yield = ralentir le CPU, le programme doit fonctionner +- n'importe quel yield! Solutions pour le join dans le chapitre suivant.



3.5 Conclusion

- Thread: flot de contrôle au sein d'un processus.
  - Processus multi-threadé: plusieurs threads dans un même espace d'adressage;
  - Avantages: capacité de réaction, partage des ressources, économie, utilisation d'une architecture multi-processeur.
- Threads utilisateurs (visibles au programmeur, inconnus du noyau) vs threads noyaux (gérés par l'OS): U plus rapides à créer et gérer, pas d'intervention du noyau.
- Les OS gèrent en général les threads, bibliothèques de threads qui fournissent des APIs au programmeur.
- Défis soulevés par la programmation multi-thread.

4 Synchronisation des processus

Exemple: retour au paradigme producteur-consommateur. Dans la solution déjà étudiée, on avait au plus $N-1$ éléments dans le buffer (où N était la taille du buffer). Idée: rajouter un compteur "count" qui indique le nombre d'éléments dans le buffer (pratique pour tests vide/plein, et incrémenter/décrémenter lors de production/consommation).

Problème: s'assurer que deux processus ne vont pas toucher le compteur en même temps! Sinon on peut se retrouver dans un état incohérent.

Implémentation de $count++$:

```
register1 := count;
register1 := register1 + 1;
count := register1;
```

Et pour $count--$:

```
register2 := count;
register2 := register2 - 1;
count := register2;
```

On peut avoir l'exécution suivante, avec initialement $count = 5$:

```
register1 := count;
register1 := register1 + 1;
register2 := count;
register2 := register2 - 1;
count := register1;
count := register2;
```

Et au final on a le compteur à 4 alors que 5 buffers sont remplis!

On ne peut pas (et doit pas) faire d'hypothèses sur l'ordre relatif des exécutions. Seuls comptent: (i) l'ordre d'exécution interne d'un processus; (ii) les relations logiques entre processus (synchronisation).

Le résultat dépend de l'ordre d'exécution entre plusieurs processus, il s'agit d'une **situation de compétition** (race condition).

4.1 Sections critiques et actions atomiques

Objectif: protéger les accès aux variables partagées. Solution: Assurer qu'un ensemble d'opérations est exécuté de manière indivisible (atomique), par exemple les 3 opérations de $count++$.

Section critique: Ensemble d'opérations qui ne doivent pas être exécutées de façon concurrente.

Exclusion mutuelle: Permettre un accès exclusif à un ensemble d'opérations.

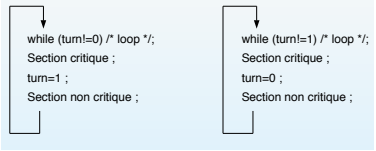
Solution: on veut des opérations d'entrée et de sortie de section critique qui garantissent l'exclusion mutuelle.

```
Processus Pi {
  ...
  Entrée en section critique
       Code section critique
  Sortie de section critique
  ...
}
```

Réalisation d'une section critique:

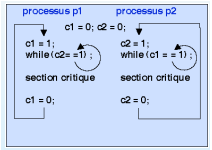
- Attente active: boucle sur un test d'entrée pour entrer en section critique.
  - Très inefficace;
  - Fonctionne s'il n'y a qu'un seul processeur;
  - Utilisé parfois en mode noyau pour de très courtes sections.
- Primitives spéciales atomiques.

Des solutions? 2 processus qui bouclent en attente active:



L'exclusion mutuelle est respectée, mais alternance stricte entre les processus: blocage alors qu'un processus n'est pas en section critique, non désiré!

Deuxième essai avec des drapeaux pour indiquer quand on va rentrer en section critique:



On a maintenant l'exclusion mutuelle et la progression, mais un risque d'interblocage!

Propriétés désirées:

1. Exclusion mutuelle: si P_i est dans sa section critique, aucun autre processus exécute sa section critique;
2. Progression: s'il n'y a aucun processus en section critique et des processus veulent rentrer en section critique, alors la sélection du prochain processus à rentrer en section critique ne peut être repoussée indéfiniment;
3. Attente bornée: lorsqu'un processus P veut rentrer en section critique, il doit y avoir une borne sur le nombre de fois que d'autres processus sont autorisés à rentrer en section critique avant que P puisse rentrer (chaque processus s'exécute à une vitesse non nulle, et pas d'hypothèses sur les vitesses d'exécution).

4.2 Solutions matérielles et logicielles

4.2.1 Solution de Peterson

Solution logicielle, suppose l'atomicité des opérations LOAD et STORE (elles ne peuvent pas être interrompues). Deux processus P_0 et P_1 qui se partagent deux variables *turn* (à qui c'est le tour de rentrer en section critique SC) et *Boolean* $flag[2]$ (vrai si P_i est prêt à rentrer en section critique).

Algorithme pour le processus P_i ($i = 0$ ou 1, et $j = 1 - i$ est l'autre processus):

```
while (true) {
  flag[i] := true;
  turn := j;
  while (flag[j] && turn == j);  // Si l'autre proc veut entrer en SC, il peut
  // SC
  flag[i] := false;
  // code hors section critique
}
```

Preuve:

1. L'exclusion mutuelle est préservée: Si les 2 proc sont en SC, alors $flag[i] = flag[j] = true$. Or, P_i entre en SC seulement si $flag[j] = false$ ou $turn = i$, et *turn* a une seule valeur → un seul proc peut entrer en SC (par exemple, P_j), l'autre (P_i) se bloque sur le while. On a donc $flag[j] = true$ et $turn = j$, et cela reste vrai tant que P_j est dans sa SC: seul P_j peut mettre *turn* à i, et mettra $flag[j]$ à $false$ en sortant de SC.

2. La progression et l'attente bornée sont satisfaits: P_i est bloqué seulement si $flag[j] = true$ et $turn = j$. Si P_j n'est pas prêt à rentrer, $flag[j] = false$ et P_i peut entrer. Si $flag[j] = true$, alors $turn = i$ ou j, et l'un des processus va pouvoir rentrer. Si c'était P_j, lorsqu'il sort, il met $flag[j]$ à $false$, ou *turn* à i si $flag[j]$ devient à nouveau $true$. P_i ne change pas la valeur de *turn* pendant la boucle while, et donc P_i va entrer en SC (progression) après au plus une entrée de P_j (attente bornée).

4.2.2 Synchronisation matérielle

Pas de *préemption*: pas d'interruption d'un processus s'exécutant en mode noyau, et donc pas de conditions de compétition! Par exemple, désactiver les interruptions sur un uni-processeur.

Par contre on utilise plutôt un noyau préemptif: un processus peut interrompre un processus noyau, c'est plus réactif, mais il faut mettre en oeuvre des solutions au problème de la section critique: besoin d'un verrou.

Machines modernes: instructions matérielles spéciales qui sont atomiques, i.e., on ne peut pas les interrompre. Deux opérations:

1. TestAndSet: teste un mot mémoire et change sa valeur à true

```
boolean TestAndSet (boolean *target) {
  boolean rv := *target;
  *target := true;
  return rv;
}
```

Utilisation: verrou *lock* partagé, initialement à $false$

```
while (true) {
  while (TestAndSet(&lock)); // boucle vide
  // SC
  lock := false
  // code hors section critique
}
```

2. Swap: échange le contenu de deux mots mémoire

```
void Swap (boolean *a, boolean *b) {
  boolean temp := *a;
  *a := *b;
  *b := temp;
}
```

Utilisation: verrou *lock* partagé, initialement à $false$, et une variable locale *key* par processus.

```
while (true) {
  key := true;
  while (key := true) { Swap(&lock, &key); }
  // SC
  lock := false;
  // code hors section critique
}
```

Les deux algorithmes satisfont l'exclusion mutuelle, mais pas l'attente bornée...

Algorithme avec attente bornée pour n processus, avec TestAnd Set: variable partagée *lock* et tableau de booléens *waiting*, initialisés à false, et variable locale *key*.

```
while (true) {
  waiting[i] := true;
  key := true;
  while (waiting[i] && key) { key := TestAndSet(&lock); }
  waiting[i] := false;
  // SC
  j := (i+1) % n;
  while ((j != i) && !waiting[j]) j := (j+1) % n;
  if (j==i) lock := false;
  else waiting[j] := false;
  // code hors section critique
}
```

Todo: Pour la prochaine fois, prouver que cet algorithme satisfait l'exclusion mutuelle, la progression et l'attente bornée.