TD1 : Programmation Concurrente, généralités

P. Trébuchet B.-M. Bui-Xuan

25 septembre 2013

Exercice 1 – Processus

Question 1

Expliquer les terminologies suivantes : système d'exploitation (ou *OS*), ordonnanceur (ou *scheduler*), concurrence des processus, préemption.

Solution:

Système d'exploitation : En franglais : *Boss* dirigeant l'utilisation du *hardware* par des *softwares*. En moins franglais : Chef d'orchestre dirigeant l'utilisation des ressources machine (mémoires, processeur, I/O, etc) par des processus. Exemple : UNIX, GNU/Linux, GNU/Linux, Linux, Linux, Linux...

Ordonnanceur : Partie du système d'exploitation qui décide du passage d'un processus d'un état actif (en exécution sur un processeur) à un état passif (en attente sur un processeur) et inversement.

Concurrence des processus : Exécution en parallèle des programmes sur une machine ou un ensemble de machines.

Préemption : Action de suspendre un processus en cours d'exécution, avant terminaison, pour passer la main à un autre processus en attente. Ceci force une commutation de contexte (ou *un context switch*) de coût variable en temps et en ressources machine.

Question 2

Y'a le bon et le mauvais ordonnanceur : "En fait le bon ordonnanceur, il monte sur scène, il a la RAM, et il cr....."
Que fait le bon ordonnanceur ?

Solution:

Il est :

- Sûr : l'ordonnanceur ne bloque pas, ne se retrouve pas dans un état incohérent
- Vivace : les processus doivent progresser dans leur exécution
- Equitable : chaque processus dispose de la même chance d'être réveillé (peut-être fonction d'une priorité)
- Efficace : le basculement d'un processus à un autre doit se faire le plus vite possible **et** le choix du prochain processus à exécuter doit se faire le plus vite possible

Question 3

Soient deux ensembles S et S' de processus définis ainsi :

$$S = [(x := x + 1; x := x + 1)||x := 2 * x],$$

$$S' = [(x := x + 1; x := x + 1)||(wait(x = 1); x := 2 * x)],$$

où un processus ne peut exécuter l'instruction $wait\ b$ que si l'expression booléenne b est vraie. On suppose que S et S' sont à mémoire commune (à l'instar des *threads*, voir Exercice 2). On suppose de plus que x est initialisée à 0 avant chaque exécution et que toutes les instructions de S et S' sont atomiques (c'est à dire non préemptible, mais on peut préempter entre deux actions atomiques).

Que peut valoir x après l'exécution de S en mode préemptif ? et après l'exécution de S' ? Que peut il se passer si les instructions ne sont pas atomiques ? Peut on avoir un scénario où x garde sa valeur 0 à la fin de l'exécution ?
Solution:
Si les instructions sont atomiques : Il y a 3 possibilités d'exécution pour S et 6 pour S' (voir l'exercice suivant). Les 3 possibilités d'exécution de S terminent sur des valeurs distinctes de x, qui sont 2, 3, et 4. Sur les 6 possibilités d'exécution de S', il y a 3 impossibilités (celles qui commencent par $wait(x = 1)$), et sur les 3 restantes, il y a deux qui terminent sur des valeurs 3 et 4 pour x, et une qui ne termine pas (à savoir $[x := x + 1; x := x + 1; wait(x = 1); x := 2 * x]$).

Si l'affectation est préemptible, dans S on peut commencer avec $(x := 2 * x)$ mais on préempte juste après l'évaluation $2 * x$ et donc, avant l'écriture de $2 * x$ dans la variable x, puis on exécute $(x := x + 1; x := x + 1)$ en totalité avant de redonner la main à l'autre processus pour qu'il termine son affectation $x := 2 * x$. De cette manière la variable x a pour valeur 0 à la fin de l'éxécution. Damned.

Question 4 – Exemple d'ordonnancement

Soit un Processus $P_1 = a_1; a_2; ...a_N$ et un processus $P_2 = b_1; b_2; ...b_M$. Chaque a_I ou b_J est une action atomique. Décrire une exécution possible de $[P_1||P_2]$: en mode non-préemtif ; puis en mode préemptif. **Solution**:

- En mode non-préemptif $[a_1; a_2; ... : a_N; b_1; b_2; ...; b_M]$
- En mode préemptif $[a_1; b_1; b_2; a_2; ...; b_M; a_N]$

Donner toutes les exécutions possibles pour `N=2` et `M=2`, **Solution**:

- En non-préemptif
  - $[a1; a2; b1; b2]$
  - $[b1; b2; a1; a2]$
- En préemptif
  - $[a1; a2; b1; b2]$
  - $[a1; b1; a2; b2]$
  - $[a1; b1; b2; a2]$
  - $[b1; a1; b2; a2]$
  - $[b1; b2; a1; a2]$
  - $[b1; a1; a2; b2]$

et le nombre d'exécutions possibles dans le cas général.
Solution:

- En non-préemptif : 2.
- En préemptif : Il s'agit de $\binom{M+N}{N}$. Une façon de voir est la suivante : le nombre d'exécutions possibles est égale au nombre de choix possibles pour les positions des a_I parmis les $M+N$ positions globales (ce qui fait $\binom{M+N}{N}$). La raison est parce qu'une fois les positions des a_I sont choisies, celles des b_J sont fixées, par ailleurs, l'ordre des a_I et des b_J est prédéterminé par la donnée de P_1 et P_2.
  Une autre façon de voir est l'ancienne soluce de Philippe : on prend toutes les permutations des ai et Bj = (N+M) ! Comme les ai et bj doivent être bien ordonnés, il faut retirer :
  - toutes les permutations des ai seuls = N !
  - toutes les permutations des bj seuls = M !

  Le résultat est donc : (N+M) ! / N !M ! = C(N,N+M) = C(M,N+M)
  Exemples : N=3, M=3 , C(N,N+M) = C(3,6) = 20
  N=4, M=4 , C(N,N+M) = C(4,8) = 70
  N=5, M=5 , C(N,N+M) = C(5,10) = 252
  N-6, M=6 , C(N,N+M) = C(6,10) = 924

Question 5 – Etats de processus

Quels sont les états possibles d'un processus ? **Solution**:

EN EXECUTION, EN SOMMEIL, EN ATTENTE (ENTREE/SORTIE,SYNCHRO), TERMINE ou ZOMBIE

Donner les transitions possibles entre états puis le graphe de transition.
Solution:
EN SOMMEIL – Election → EN EXECUTION
EN EXECUTION – SynchroLock → EN ATTENTE
EN EXECUTION – Quantum → EN SOMMEIL
EN EXECUTION → TERMINE
EN ATTENTE – SynchroUnlock → EN EXECUTION

Exercice 2 – Threads

Question 1
Citez les différences principales entre *threads* et *processus*.
Solution:
- Mémoire partagée
- Changement de contexte rapide
- environemment partagé
- pid unique

Attention : dans les transparentes de cours, le terme processus est plutôt une abstraction des programmes en informatique en toute généralité. Par exemple, le terme "processus à mémoire commune" fait beaucoup référence à des *threads*.

Question 2
Les *threads* sont aussi appelés *processus légers*, pourquoi une telle denomination ? Pourquoi a-t-on créé ces objets ?
Solution:
Processus légers car pas issu du clonage d'un processus existant donc économiques en mémoire et en ressources disque (swap). On les a créé à la base pour pouvoir modéliser facilement les systèmes concurrents ex : interfaces graphiques etc...

Question 3 – Gestion des threads
Qui s'occupe de la gestion des threads ?
Solution:
Threads Utilisateurs :
- processus utilisateur/bibliothèque (ex. : fair threads linkes) + : simples, efficaces sur un proc. en général, ne peut exploiter le SMP
- système d'exploitation (ex. linux kernel threads et primitive clone()) moins efficaces sur un proc. + : peut exploiter le SMP

Question 4 – Contraintes d'ordonnancement
Les contraintes d'ordonnancement des threads sont elles les mêmes que pour les processus ?
Solution:
Threads utilisateurs : non, l'ordonnancement y est en géneral plus spécialisé
Threads kernel : l'ordonnanceur est unique pour threads+processus

Question 5 – Difficultés des Threads
Quelles sont les difficultés principales liées à l'utilisation des threads ?
Solution:

- Non-déterminisme (préemptif/répartition)
- Difficile de comprendre/analyser/déboguer les programmes
- Gestion de la mémoire partagée
- Problèmes de sûreté :
  - blocages (deadlock)
  - lecture/écriture incohérente
- Problème de vivacité :
  - verrou actif (livelock) ⇒ Le programme semble progresser mais il ne fait rien d'intéressant
  - famines/inéquitabiliée ⇒ des threads accaparent des ressources, les autres bloquent
- Attentes actives : Consommation inutile de CPU

Exercice 3 – Dîner des philosophes

L'histoire se passe dans un monastère reculé...

En bref, nous avons n moines assis autour d'une table ronde ayant chacun une assiette privée mais partageant des fourchettes. Il y a une et exactement une fourchette entre chaque pair de moines. Pour pouvoir manger proprement un moine doit posséder à la fois la fourchette à sa gauche et la fourchette à sa droite. La vie d'un moine est une boucle infinie : penser, manger, penser, manger, penser, et ainsi de suite. Si un moine ne mange pas pendant un certain temps il meurt de faim, entraînant bien de tristes conséquences pour tout le monastère. Peut on définir des points supplémentaires dans le règlement intérieur pour un fonctionnement paisible et studieux, sans accident regrettable ?

Question 1

Donner une solution dans le cas où n est pair.

Solution:

Dans le cas où n est pair, monsieur X. propose les points suivants.

- Les moines doivent être numérotés successivement selon leur place à table.
- Un moine ayant un numéro pair doit d'abord posséder la fourchette gauche avant la fourchette droite.
- Un moine ayant un numéro impair doit d'abord posséder la fourchette droite avant la fourchette gauche.
- Un moine ayant une fourchette garde la fourchette jusqu'à ce qu'il ait la deuxième.
- Un moine ayant deux fourchettes doit manger pendant un temps fini.
- Un moine ayant fini de manger doit déposer ses fourchettes et marquer une pause.

C'est suffisant si on fait confiance à l'intinct d'un moin affamé pour chercher ses fourchettes activement. Parce que si un moine numéroté pair a une seule fourchette (celle de gauche) ça veut dire que son voisin de droite (numéroté impair) est entrain de manger. Les autres cas sont similaires.

Question 2

Dans le cas général, la proposition Chandy-Misra (1984) est basée sur le principe fondamental de la politesse et peut se résumer ainsi :

- A l'inauguration du monastère on salit toutes les fourchettes, installe les moines, et confit chaque fourchette au plus ancien des deux moines voisins.
- Un moine manquant une fourchette demande poliment à son voisin l'obtention de celle-ci. De même s'il lui manque les deux fourchettes (il demande alors à chacun de ses deux voisins).
- Un moine recevant une demande de fourchette examine la fourchette en question. S'elle est sale alors très poliment le moine nettoie et donne celle-ci au moine qui la demande. Sinon, il la garde toute de même.
- Un moine ayant deux fourchettes propres doit manger pendant un temps fini.
- Les fourchettes deviennent sales après chaque usage.

Est-ce suffisant ?

Solution:

Oui.

Exercice 4 – Implantation de processus et threads

Question 1

En utilisant l'appel système `pid_t fork(void);`, créer deux processus affichant la valeur de retour de l'appel à `fork`.

Solution:

```
#include<unistd.h>
#include<sys/types.h>
#include<stdio.h>

int main()
{
        int i=0;
        int pid=fork();
        printf("Ma varible pid est %d\n",pid);
}
```

Question 2

Ecrire maintenant un programme possédant une variable entière `i`, créant un processus fils et tel que le père affiche un message indiquant qu'il est le père et demande à l'utlisateur de saisir une valeur pour `i` au clavier et tel que le fils commence par dormir 4 secondes, puis affiche le contenu de `i`.

Solution:

```
#include<unistd.h>
#include<sys/types.h>
#include<stdio.h>

int main()
{
        int i=0;
        int pid=fork();
        if(pid){
                printf("Je suis le pere et ma varible pid est %d\n",pid);
                printf("donne moi un nombre maintenant\n");
                scanf("%d",&i);
                printf("ma variable i vaut %d\n",i);
                }
        else
        {
                sleep(4);
                printf("\nje suis le fils et ma variable pid vaut %d\n",pid);
                printf("je suis le fils et ma variable i vaut %d\n",i);
        }


}
```

Question 3

En utilisant la fonction `pid_t getpid(void);` (`pid_t` est compatible avec `int`), écrire le même programme

avec des `threads` POSIX sauf que le père attend cette fois ci la fin du fils avant de terminer. Que constatez vous quant au contenu de la variable `i` dans le fils ? Qu'en deduisez vous sur le fonctionnment des threads Posix ? Quelle précautions sont à prendre pour éviter au programme précédant de produire un résultat indéfini ?
Solution:

```
#include<unistd.h>
#include<sys/types.h>
#include<stdio.h>
#include <pthread.h>
#include<stdlib.h>

int i=0;
pthread_t th1;

void *fun_fils(void *toto)
{
  sleep(10);
  printf("c'est moi le fils et le contenu de pid vaut %d\n",getpid());
  printf("c'est moi le fils et le contenu de i vaut %d\n",i);
  return NULL;
}

int main()
{
  if (!pthread_create(&th1,NULL,fun_fils,NULL)) {
    printf("je suis le pere et ma variable pid vaut %d\n",getpid());
    printf("donne moi vite (moins de 10 secondes) un nombre \n");
    scanf("%d",&i);
    printf("je suis le pere et ma variable i vaut %d\n",i);
    pthread_join(th1,NULL);
  } else {
    printf("Creation de thread impossible.\n");
  }

}
```

On voit ici que la mémoire est partagéee il faut répondre avant les 4 secondes de sommeil pour le fils sinon on ne sait pas ce qui se passe.