Université Bordeaux I | U.F.R. Mathématiques et Informatique
Session de septembre 2001 | Alain Griffault

Maîtrise d'informatique
Systèmes d'exploitation
(Durée : 2 heures)

Documents interdits

1 Cours : 2 points

Question 1.1 *Dans un système d'exploitation, citez deux fonctionnalités, pour lesquelles la solution optimale (qui existe) n'est pas utilisée. Dans les deux cas, expliquez rapidement pourquoi les algorithmes optimaux ne sont pas utilisés.*

2 Synchronisation avec des sémaphores : 6 points

On dispose de processus `a, b, c` et `init` qui sont lancés au même instant. Le but est de contrôler l'ordonnancement des actions des processus `a, b` et `c`. Pour cela on dispose d'un langage de spécification comportant les opérations de concaténation séquentielle `'*'` et `'.'`, de choix exclusif `'+'` et de mise en parallèle `'||'`. Les processus `a, b` et `c` exécutent le code suivant :

```
Processus x() /* les  codes sont différents pour x = a, b ou c */
debut
   boucle { avantx ; actionsx ; apresx ; }
fin            /* actionsx n'est pas forcément atomique */
```

Remarquons que si `avantx` et `apresx` sont vides pour un processus donné, celui-ci exécute indéfiniment `actionsx`, et ce de façon séquentielle mais parallèlement aux autres processus.

Question 2.1 *Donnez, pour les spécifications suivantes (où x représente* `actionsx` *) le code (qui est toujours une séquence de* `wait()` *et/ou de* `signal()`*) du processus* `init`*, et de* `avantx` *et* `apresx` *pour chaque processus* `a, b` *et* `c`*.*

1. `(a.b.c)* soit : a . b . c . a . b . c .` *etc*
2. `(a+b+c)* exemple : a . b . c . a . a . c . c . b` *etc*
3. `(a||b||c)* soit : a||b||c . a||b||c . a||b||c .` *etc*
4. `((a.b)||c)* soit : (a.b)||c . (a.b)||c . (a.b)||c .` *etc*

Pour répondre, on utilisera le format suivant :

Langage de spécification : ...
Noms des sémaphores : ...
Code du processus init : ...

	a	b	c
avantx			
apresx			

Rappels :

`type semaphore = (n : entier; l : liste de processus);`

manipulée uniquement par les actions atomiques `wait` ($\equiv$ P) et `signal` ($\equiv$ V).

```
procedure wait(var S : semaphore)
debut
S.n := S.n - 1;
si (S.n < 0) alors debut
        ajouter(moi,S.l);
        block;
        fin
fin
```

```
procedure signal(var S : semaphore)
debut
S.n := S.n + 1;
si (S.n <= 0) alors debut
        enlever(P,S.l);
        wakeup(P);
        fin
fin
```

3 Allocation mémoire : 12 points

On considère un algorithme d'allocation de blocs contigüs en mémoire, où la liste des tailles possibles est $t_1, t_2, \ldots, t_n$ (pour une mémoire de taille totale t_n), avec les t_i donnés par :

$$t_1 = 1, t_2 = 2, t_i = t_{i-1} + t_{i-2} \text{ pour } i > 2$$

soit $t_3 = 3, t_4 = 5, t_5 = 8, \ldots$.
On maintient pour chaque valeur de k une liste L_k des blocs libres de taille t_k ; chaque bloc libre est repéré par son adresse de base. Au départ, toutes les listes sont vides sauf la liste L_n qui contient 0 ; 0 étant l'adresse de base du seul bloc libre correspondant à la mémoire totale. Les listes L_k sont gérées par l'algorithme FIFO.

L'algorithme d'allocation d'un bloc de taille t

1. calculer k le plus petit entier tel que $t_k \geq t$,
2. calculer p le plus petit entier tel que $p \geq k$ et L_p non vide. (il faut trouver un bloc assez grand)
   (a) si p n'existe pas, la requête est mise en attente.
   (b) si $p = k$, on alloue le premier bloc de la liste L_p et on le retire de la liste.
   (c) si $p > k$, on scinde ce bloc libre, de taille t_p, en deux blocs libres dits *compagnons*, de tailles t_{p-1} (pour la partie gauche des petites adresses) et t_{p-2} (pour la partie droite). On ajuste les trois listes L_p, L_{p-1} et L_{p-2} en conséquence, et on itère l'algorithme à partir du calcul de p.

L'algorithme de libération d'un bloc B de taille t_k

1. insérer dans la liste L_k le bloc libéré.
2. chercher si son bloc compagnon C est un bloc libre.
   (a) si C est libre, fusionner B et C en un seul bloc, supprimer B et C des listes et *libérer* le résultat de la fusion (algorithme récursif).
   (b) si C n'est pas libre, fin de l'algorithme.

Question 3.1 *Étudiez le partitionnement d'une mémoire de taille* 34, *obtenu suite aux requêtes d'allocation suivantes :* $+4, +10, +2$. *Pour chaque requête, indiquez très précisément le bloc alloué (adresse de base et taille), et le contenu des listes libres non vides après l'allocation.*

Question 3.2 *Quelle est alors la taille maximale d'une demande d'allocation acceptable par le système ?*

Question 3.3 *Quelle est dans cette situation la fragmentation interne de la mémoire ?*

Question 3.4 *Traitez les requêtes suivantes (le signe* $-$ *indique une libération) :* $-4, +4, -2, -10, -4$.

On ajoute à chaque bloc B un compteur $g(B)$ géré comme suit :
- $g(B) = 0$ pour le bloc B initial de taille maximale.
- Lorsqu'un bloc libre B est scindé en un compagnon gauche B_g et un compagnon droit B_d, on pose :

$$g(B_g) = g(B) + 1, g(B_d) = 0$$

Question 3.5 *Énoncez la règle de calcul de* g *lorsqu'on fusionne deux compagnons. Expliquez pourquoi il n'y a qu'une seule règle raisonnable.*

Question 3.6 *Traitez les requêtes* $+4, +10, +2, -4, +4, -2, -10, -4$ *(celles des questions 1 et 3) en précisant les valeurs de g(B) pour chacun des blocs.*

Question 3.7 *Donnez l'algorithme qui calcule pour un bloc* B *de base* b *et de taille* t_k*, la base et la taille de son compagnon.*

Question 3.8 *Ecrivez très précisément l'algorithme de libération d'un bloc* B *de base* b *et de taille* t_k.
Vous pourrez supposer disposer du type bloc (une structure qui contient une adresse de base (B.base), un entier pour indicer la taille (B.taille) et un entier pour coder g (B.g)) ; d'un tableau L[] de liste ; d'un tableau t[] des tailles des blocs et des fonctions suivantes :

```
void insererFIFO(liste l, bloc B);
bloc chercherBloc(liste l, adresse base);
void supprimer(liste l, bloc B);
bloc fabriquerBloc(adresse base, entier taille, entier g);
```