Université Bordeaux I
Session de septembre 2001

U.F.R. Mathématiques et Informatique
Alain Griffault

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

Documents interdits

SUJET + CORRIGE

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.*

Réponse à la question 1.1 (2 points) : Il existe plusieurs raisons, pour lesquelles une solution optimale n'est pas utilisée en pratique. Pour chacune nous donnons une liste non exhaustive d'exemples.

- Une hypothèse au moins n'est pas réaliste.
  1. L'algorithme optimal pour la gestion des défauts de page (quelle page doit-on évincer lorsqu'une nouvelle page doit être mise dans un cadre) utilise comme donnée en entrée, la séquence des pages qui seront utilisées dans le futur. Cette connaissance étant évidemment impossible, l'algorithme optimal ne peut être implanté.
  2. L'algorithme optimal pour l'ordonnancement de l'unité centrale est l'algorithme du *travail plus court d'abord*. Il a besoin de connaître, pour chaque processus, la durée du prochain cycle UC avant la prochaine entrée-sortie. Cette connaissance étant évidemment impossible en général, l'algorithme optimal ne peut être utilisé.

  Remarque : Ces algorithmes peuvent être implantés en utilisant comme entrées, la connaissance du passé. Cela revient à faire l'hypothèse que le futur est le miroir du passé.
- La solution possède au moins une propriété non souhaitée.
  1. L'algorithme optimal pour diminuer le temps de recherche des blocs sur le disque, va intuitivement privilégier les blocs *proches* de la position courante de la tête de lecture. Avec une telle solution, une requête d'un bloc *éloigné* peut ne jamais être satisfaite. Cet algorithme n'est dons pas équitable, et ne peut donc pas raisonnablement être implanté.
- Le coût de la solution est trop important.
  1. L'algorithme de prévention de l'interblocage (dit du banquier) est très peu souvent utilisé du fait de son coût CPU.

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*

Réponse à la question 2.1 (1 + 1 + 2 + 2 points) :

1. Langage de spécification : `(a.b.c)*`
   Noms des sémaphores : A, B et C
   Code du processus init : V(A)

	a	b	c
avantx	P(A)	P(B)	P(C)
apresx	V(B)	V(C)	V(A)

2. Langage de spécification : `(a+b+c)*`
   Noms des sémaphores : D
   Code du processus init : V(D)

	a	b	c
avantx	P(D)	P(D)	P(D)
apresx	V(D)	V(D)	V(D)

3. Langage de spécification : `(a||b||c)*`
   Noms des sémaphores : A, B et C
   Code du processus init : $V(A)^2V(B)^2V(C)^2$

	a	b	c
avantx	P(A)P(A)	P(B)P(B)	P(C)P(C)
apresx	V(B)V(C)	V(C)V(A)	V(A)V(B)

4. Langage de spécification : `((a.b)||c)*`
   Noms des sémaphores : A, B et C
   Code du processus init : V(A)V(A)V(C)

	a	b	c
avantx	P(A)P(A)	P(B)	P(C)
apresx	V(B)	V(C)V(A)	V(A)

Rappels :

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

manipulée uniquement par les actions atomiques `wait` (≡ P) et `signal` (≡ 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.*

Réponse à la question 3.1 (2 points) : Avec le découpage t_{p-1} à gauche on obtient :

	k	p	L_1 $t_1=1$	L_2 $t_2=2$	L_3 $t_3=3$	L_4 $t_4=5$	L_5 $t_5=8$	L_6 $t_6=13$	L_7 $t_7=21$	L_8 $t_8=34$	Bloc alloué
initial										0	
+4	4	8						21	0		
		6				29	21		0		
		4					21		0		L_4 adresse 29
+10	6	7					21, 13	0			
		6					21, 13				L_6 adresse 0
+2	2	5			26	21	13				
		3	28	26		21	13				
		2	28			21	13				L_2 adresse 26

Avec le découpage t_{p-1} à droite (le sujet précisait à gauche) on obtient :

	k	p	L_1 $t_1=1$	L_2 $t_2=2$	L_3 $t_3=3$	L_4 $t_4=5$	L_5 $t_5=8$	L_6 $t_6=13$	L_7 $t_7=21$	L_8 $t_8=34$	Bloc alloué
initial										0	
+4	4	8						0	13		
		6				0	5		13		
		4					5		13		L_4 adresse 0
+10	6	7					5, 13	21			
		6					5, 13				L_6 adresse 21
+2	2	5			5	8	13				
		3	5	6		8	13				
		2	5			8	13				L_2 adresse 6

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

Réponse à la question 3.2 (1 point) : La taille maximale est 8, c'est celle du plus grand bloc libre (L_5).

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

Réponse à la question 3.3 (1 point) : La fragmentation interne est : $(4+10+2)/(5+13+2) = 16/20 = 0,8$

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

Réponse à la question 3.4 (2 points) : Avec le découpage t_{p-1} à gauche on obtient :

	k	p	L_1 $t_1=1$	L_2 $t_2=2$	L_3 $t_3=3$	L_4 $t_4=5$	L_5 $t_5=8$	L_6 $t_6=13$	L_7 $t_7=21$	L_8 $t_8=34$	Bloc alloué
-4			28			21, 29	13				
+4	4	4	28			29	13				L_4 adresse 21
-2			28	26		29	13				
					26	29	13				
-10					26	29	13	0			
					26	29			0		
-4					26	29, 21			0		
						29	21		0		
								21	0		
										0	

Avec le découpage t_{p-1} à droite (le sujet précisait à gauche) on obtient :

	k	p	L_1 $t_1=1$	L_2 $t_2=2$	L_3 $t_3=3$	L_4 $t_4=5$	L_5 $t_5=8$	L_6 $t_6=13$	L_7 $t_7=21$	L_8 $t_8=34$	Bloc alloué
-4			5			8, 0	13				
+4	4	4	5			0	13				L_4 adresse 8
-2			5	6		0	13				
					5	0	13				
-10					5	0	13	21			
					5	0			13		
-4					5	0, 8			13		
						0	5		13		
								0	13		
										0	

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.*

Réponse à la question 3.5 (1,5 point) : Soient B_g et B_d deux compagnons libres. La fusion produit un nouveau bloc libre B avec :

$$g(B)=g(B_g)-1.$$

Cette règle est la seule à vérifier la propriété : soit une configuration quelconque de la mémoire, soit une liste de requêtes dont l'effet global est nul, alors la configuration terminale est identique à la configuration de départ.

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.*

Réponse à la question 3.6 (1,5 point) : Avec le découpage t_{p-1} à gauche on obtient :

	k	p	L_1 $t_1=1$	L_2 $t_2=2$	L_3 $t_3=3$	L_4 $t_4=5$	L_5 $t_5=8$	L_6 $t_6=13$	L_7 $t_7=21$	L_8 $t_8=34$	Bloc alloué
initial										0_0	
+4	4	8						21_0	0_1		
		6				29_0	21_1		0_1		
		4					21_1		0_1		L_4 adresse 29
+10	6	7					21_1, 13_0	0_2			
		6					21_1, 13_0				L_6 adresse 0
+2	2	5			26_0	21_2	13_0				
		3	28_0	26_1		21_2	13_0				
		2	28_0			21_2	13_0				L_2 adresse 26
-4			28_0			21_2, 29_0	13_0				
+4	4	4	28_0			29_0	13_0				L_4 adresse 21
-2			28_0	26_1		29_0	13_0				
					26_0	29_0	13_0				
-10					26_0	29_0	13_0	0_2			
					26_0	29_0			0_1		
-4					26_0	29_0, 21_2			0_1		
						29_0	21_1		0_1		
								21_0	0_1		
										0_0	

Avec le découpage t_{p-1} à droite (le sujet précisait à gauche) on obtient :

	k	p	L_1 $t_1=1$	L_2 $t_2=2$	L_3 $t_3=3$	L_4 $t_4=5$	L_5 $t_5=8$	L_6 $t_6=13$	L_7 $t_7=21$	L_8 $t_8=34$	Bloc alloué
initial										0_0	
+4	4	8						0_1	13_0		
		6				0_2	5_0		13_0		
		4					5_0		13_0		L_4 adresse 0
+10	6	7					5_0, 13_1	21_0			
		6					5_0, 13_1				L_6 adresse 21
+2	2	5			5_1	8_0	13_1				
		3	5_2	6_0		8_0	13_1				
		2	5_2			8_0	13_1				L_2 adresse 6
-4			5_2			8_0, 0_2	13_1				
+4	4	4	5_2			0_2	13_1				L_4 adresse 8
-2			5_2	6_0		0_2	13_1				
					5_1	0_2	13_1				
-10					5_1	0_2	13_1	21_0			
					5_1	0_2			13_0		
-4					5_1	0_2, 8_0			13_0		
						0_2	5_0		13_0		
								0_1	13_0		
										0_0	

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.*

Réponse à la question 3.7 (1,5 point) : Soit C le compagnon.

```
si g(B) == 0 alors // B est un compagnon droit
   debut
   taille(C) = taille des blocs de la liste L(k-1);
   base(C) = b - taille(C)
   fin
```

```
sinon // g(B) est strictement positif et B est un compagnon gauche
   debut
   taille(C) = taille des blocs de la liste L(k+1);
   base(C) = b + taille(B)
   fin
```

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);
```

Réponse à la question 3.8 (1,5 point) :

```
fonction libererBloc(B : bloc)
var Courant, Compagnon : bloc;
debut
si B.g == 0 alors // B est un compagnon droit
   debut
   Compagnon = chercherBloc(L[B.taille-1], b - t[B.taille-1]);
   si Compagnon != null alors
      debut
      supprimer(L[B.taille - 1], Compagnon);
      Courant = fabriquerBloc(b - t[B.taille-1], t[B.taille+1], g.Compagnon - 1);
      libererBloc(Courant);
      fin
   sinon
      insererFIFO(L[B.taille], B);
   fin
sinon // g(B) est strictement positif et B est un compagnon gauche
   debut
   Compagnon = chercherBloc(L[B.taille + 1], b + t[B.taille]);
   si Compagnon != null alors
      debut
      supprimer(L[B.taille + 1], Compagnon);
      Courant = fabriquerBloc(b, t[B.taille+2], g.B - 1);
      libererBloc(Courant);
      fin
   sinon
      insererFIFO(L[B.taille], B);
   fin
fin
```