Université Bordeaux I
Session de janvier 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 *Décrivez (avec un schéma éventuellement) l'exécution d'un appel de procédure distant (RPC).*

Réponse à la question 1.1 (2 points) :

1. Le client appelle le noyau : `envoyer(RPC, procédure X, message)`,
2. le noyau utilise le port RPC fixe *matchmaker* du serveur pour lui demander où se trouve la procédure X : `(From : client, To : serveur, Port : matchmaker, Request : address for procédure X)`,
3. le serveur reçoit le message, puis retourne la réponse trouvée dans une table : `(From : serveur, To : client, Port : noyau, Response : RPC X at Port P)`,
4. le noyau reçoit le message et effectue pour le client l'appel distant : `(From : client, To : serveur, Port : P, message)`,
5. le démon en écoute sur le port P traite (où fait traiter) la demande et retourne (où fait retourner) le résultat : `(From : RPC port P, To : client, Port : noyau, résultat)`
6. le noyau transmet le résultat au client : `envoyer(RPC, procédure X, résultat)`.

2 Algorithme du banquier : 4 points

Un système est composé de trois processus P_1, P_2, P_3 et de quatre types de ressource.
Initialement, le système est décrit par $Existe = (5, 6, 6, 5)$, et $Max = \begin{pmatrix} 4 & 3 & 3 & 1 \\ 2 & 3 & 3 & 4 \\ 2 & 2 & 3 & 3 \end{pmatrix}$.
Soit la situation $Tenue = \begin{pmatrix} 2 & 1 & 2 & 0 \\ 1 & 1 & 2 & 1 \\ 1 & 2 & 1 & 1 \end{pmatrix}$.

Question 2.1 *La situation est-elle saine ? Justifiez.*

Réponse à la question 2.1 (2 points) : Une situation saine est caractérisée par :
`situationSaine(E,Max,T) = non (trouverAttenteCirculaire(E,T,Max-T));`
L'algorithme `trouverAttenteCirculaire(E,T,R)` donne :

1. $D = (1, 2, 1, 3)$
2. $O < R_2 = (1, 2, 1, 3) \leq D$, donc $R_2 = O$ et $D = (2, 3, 3, 4)$
3. $O < R_1 = (2, 2, 1, 1) \leq D$, donc $R_1 = O$ et $D = (4, 4, 5, 4)$
4. $O < R_3 = (1, 0, 2, 2) \leq D$, donc $R_3 = O$ et $D = (5, 6, 6, 5)$
5. $(D = E) \Rightarrow$ pas d'attente circulaire

La situation est donc saine.

Question 2.2 *On suppose que le processus P_1 émet la requête $R_1 = (0,1,1,0)$. La requête va t'elle être honorée de suite, ou bien va t'elle être mise en attente ? Justifiez.*

Réponse à la question 2.2 (2 points) : La requête sera honorée de suite si et seulement si la situation après la requête est saine, c'est à dire si `situationSaine(E,Max,T+R1)`.
L'algorithme `trouverAttenteCirculaire(E,T,R)` donne :

1. $D = (1,1,0,3)$
2. $O < R_1 = (2,1,0,1) \leq D$, est faux,
3. $O < R_2 = (1,2,1,3) \leq D$, est faux,
4. $O < R_3 = (1,0,2,2) \leq D$, est faux,
5. $(D \neq E) \Rightarrow$ attente circulaire

La situation n'est pas saine ; la requête est donc mise en attente.

3 Algorithme de remplacement de pages : 6 points

L'idée de cet algorithme est d'évincer la page la moins fréquemment utilisée. Pour cela cet algorithme associe à chaque cadre un compteur qui comptabilise le nombre d'accès à la page contenue dans ce cadre.

Question 3.1 *Définissez un algorithme basé sur cette idée. Vous devez décrire précisément :*

1. *la valeur initiale des compteurs,*
2. *les événements qui provoquent les modifications (incrémentation, décrémentation, affectation...),*
3. *la fonction de sélection de la page à évincer.*

Réponse à la question 3.1 (2 points) : Voici l'une des réponses possibles : On note `valeur[i]` la valeur du compteur associé au cadre `i`. Ces compteurs sont initialisés à `0`.
– accès sans défaut de page à la page du cadre `c` : `valeur[c]++;`
– chargement, suite à un défaut de page, d'une page dans le cadre `c` : `valeur[c] = 1;`
– lors d'un défaut de page, le cadre libéré est celui retourné par la fonction suivante. Le paramètre est le résultat de l'appel précédent (0 pour le premier appel), afin d'avoir un algorithme de type FIFO si tous les compteurs sont égaux.

```
int cadreLibere(int ref){
  int  i;
  int min=(ref+1) % nombreCadre;
  for(i=(ref+2) % nombreCadre; i<nombreCadre; i++)
     if (valeur[i] < valeur[min]) min =i;
  for(i=0; i<=ref; i++)
     if (valeur[i] < valeur[min]) min =i;
  return min;
}
```

Question 3.2 *Construisez, pour une mémoire comportant quatre cadres initialement vides, la suite des configurations de la mémoire, pour la suite de références :*
`1, 2, 3, 4, 5, 3, 4, 1, 6, 7, 8, 7, 8, 9, 7, 8, 9, 5, 4, 5, 4, 2.`

Réponse à la question 3.2 (1,5 point) :

		1	2	3	4	5	3	4	1	6	7	8	7	8	9	7	8	9	5	4	5	4	2
cadre 0	page	1				5				6		8											
	valeur	1				1				1		1		2			3						
cadre 1	page		2						1		7				9								2
	valeur		1						1		1		2		1			2					1
cadre 2	page			3												7			5				
	valeur			1			2									1			1		2		
cadre 3	page				4																		
	valeur				1			2												3		4	

Question 3.3 *Donnez une suite infinie de références qui après avoir chargée une page dans le premier cadre, ne provoquera plus jamais de défaut de page sur ce cadre, bien que la page ne soit pas infiniment souvent utilisée. Donnez une interprétation possible à votre suite de références.*

Réponse à la question 3.3 (1 point) : Soit la suite `1, 1, (2, 3, 4, 5,)*` , avec 4 cadres. On obtient :

		1	1	2	3	4	5	2	3	4	5	2	3	4	5	2	3	4	5	2	3	4	5
cadre 0	page	1																					
	valeur	1	2																				
cadre 1	page			2			5			4			3			2			5			4	
	valeur			1			1			1			1			1			1			1	
cadre 2	page				3			2			5			4			3			2			5
	valeur				1			1			1			1			1			1			1
cadre 3	page					4			3			2			5			4			3		
	valeur					1			1			1			1			1			1		

Il faut remarquer que l'existence de ce cycle est indépendant de la fonction `cadreLibere`, mais du uniquement à la spécification. Plus généralement, pour N cadres, cet algorithme n'évincera jamais la page du premier cadre utilisé pour la suite `1, 1, (2, 3, 4, ..., N, (N+1))*`.
Une telle suite peut correspondre à l'exécution d'un programme, les deux premières références correspondent à du code qui ne sera jamais utilisé durant l'exécution (par exemple le traitement des exceptions).

Question 3.4 *En utilisant une technique de type* vieillissement*, décrivez les modifications à apporter à votre algorithme afin de corriger le problème précédent.*

Réponse à la question 3.4 (1,5 point) : Les techniques de type *vieillissement* essaient intuitivement de calculer des fréquences (où des moyennes) au lieu de calculer des occurrences. Pour cela, ils décrémentent où bien divisent les compteurs à intervalles réguliers. Une solution possible est de modifier ainsi l'algorithme précédent :

- ajouter les variables :
```
int nombreAcces=0;
int periode = P;
```
- accès sans défaut de page à la page du cadre `c` :
```
valeur[c]++;
vieillirPages();
```
- chargement, suite à un défaut de page, d'une page dans le cadre `c` :
```
valeur[c] = 1;
vieillirPages();
```
- la fonction `vieillirPages()`
```
void vieillirPages() {
  int i;
  nombreAcces++;
  if (nombreAcces == periode) {
    nombreAcces = 0;
    for(i=0; i<=nombreCadre; i++)
      valeur[i] >> 1;
  }
}
```
- la fonction de sélection de la page à évincer est inchangée.

De plus, la taille des compteurs peut être optimisée en fonction du paramètre `periode`. Par exemple, ils seront codés sur un octet si P est égal à 2^8.

4 Moniteurs et sémaphores : 8 points

Definition 4.1 *Un sémaphore* `S` *est une structure de données*

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

manipulée uniquement par les actions atomiques `wait(S)` *et* `signal(S)`*, également nommées* `P(S)` *et* `V(S)`*.*

```
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
```

Definition 4.2 *Un moniteur est constitué d'un ensemble de variables, d'un ensemble de conditions et d'un ensemble de procédures.*

les variables *sont locales au moniteur, donc non accessibles aux utilisateurs.*

les conditions *sont locales au moniteur, donc non accessibles aux utilisateurs. Une condition* `c` *n'est manipulable qu'au moyen de deux primitives :*

- `c.attendre` *: bloque le processus* `P` *et le place en attente de* `c`*,*
- `c.signaler` *: réveille un processus en attente de* `c` *s'il en existe au moins un, ne fait rien sinon.*

les procédures *pour un moniteur* `M` *sont externes ou bien locales.*

- *les procédures externes sont celles qui sont accessibles aux utilisateurs. Un moniteur garantit que ses procédures externes sont exécutées en exclusion mutuelle.*
- *les procédures locales sont :* `init(M)`, `entree(M)`, `sortie(M)`*. La procédure locale* `init(M)` *est exécutée une seule fois à la création du moniteur. La procédure* `entree(M)` *(resp.* `sortie(M)`*) est exécutée au début (resp. à la fin) de chaque procédure externe. Elles y sont ajoutées par le compilateur, et ce sont elles (avec les primitives sur les conditions) qui font l'implémentation du moniteur.*

Comme exemple, le moniteur suivant protège la lecture d'une variable `a` non initialisée.

```
sync : moniteur;
  var fait : boolean;
      fini : condition;
procedure finEcrire;
  debut
  fait := vrai;
  fini.signaler
  fin
procedure debutLire;
  debut
  si (non fait) alors
    fini.attendre
  fin
procedure init;
  debut
  fait := faux
  fin
```

```
processus P
  debut
  ecrire(a);
  sync.finEcrire;
  fin

processus Q
  debut
  sync.debutLire;
  lire(a);
  fin
```

Question 4.1 *Donnez le code des procédures externes* `debutEcrire` *et* `finEcrire` *du moniteur suivant qui procure une solution aux problèmes des lecteurs rédacteurs. La priorité est donnée aux lecteurs, une écriture n'est pas autorisée si des lecteurs sont en attente.*

```
fichier : moniteur;
  var : ecrire : booleen;
        nbLecteurs : entier;
        cEcrire, cLire : condition;
procedure debutLire;                  | processus lecteur;
  debut                               |   debut
  nbLecteurs := nbLecteurs + 1;       |   fichier.debutLire;
  si ecrire alors                     |   < accès aux données protégées >
     debut                            |   fichier.finLire;
     cLire.attendre;                  |   fin
     cLire.signaler;                  |
     fin                              |
  fin                                 | processus redacteur;
procedure finLire;                    |   debut
  debut                               |   fichier.debutEcrire;
  nbLecteurs := nbLecteurs - 1;       |   < accès aux données protégées >
  si (nbLecteurs = 0) alors           |   fichier.finEcrire;
     cEcrire.signaler                 |   fin
  fin
procedure init;
  debut
  ecrire := faux;
  nbLecteurs := 0;
  fin
```

Réponse à la question 4.1 (2 points) :

```
procedure debutEcrire;
  debut
  si (ecrire ou (nbLecteurs > 0)) alors
     cEcrire.attendre;
  ecrire := vrai;
  fin

procedure finEcrire;
  debut
  ecrire := faux;
  si (nbLecteurs > 0) alors
     cLire.signaler;            // priorité aux lecteurs.
  sinon
     cEcrire.signaler;
  fin
```

On se propose maintenant d'implémenter les moniteurs à partir des sémaphores.

Question 4.2 *Dans la cas ou la primitive* `signaler` *n'est admise que comme dernière instruction d'une procédure externe, donnez une implémentation : d'une part des primitives* `signaler` *et* `attendre` *pour une condition ; d'autre part des procédures locales* `init(M)`, `entree(M)`, `sortie(M)`. *Vous pourrez utiliser :*
– un sémaphore pour gérer l'exclusion entre les procédures externes,
– pour chaque condition, un sémaphore et un compteur du nombre de processus en attente.

Réponse à la question 4.2 (3 points) :

```
procedure locale init();
  debut
  mutex : semaphore = (1,{});
  cond[1..n] : semaphore = (0,{});   // le premier wait doit être bloquant
  attente[1..n] : entier = 0;
  fin

primitive signaler(condition i);
  debut
  si attente[i] > 0 alors
    signal(cond[i]);          // on reveille un processus en attente de la condition
  sinon
    signal(mutex);            // on permet à un processus d'entrer dans le moniteur
  fin

primitive attendre(condition i);
  debut
  attente[i]++;
  signal(mutex);              // on permet à un processus d'entrer dans le moniteur
  wait(cond[i]);              // car on sait que l'on va se bloquer.
  attente[i]--;
  fin

procedure locale entree();
  debut
  wait(mutex);                // on attend l'autorisation
  fin

procedure locale fin();
  debut
  fin
```

Dans la cas ou la primitive `signaler` est admise partout dans une procédure externe, se pose le problème suivant : Lorsque `P` exécute l'instruction `c.signaler`, cela réveille un processus `Q` ; il y a alors deux processus actifs en même temps dans deux procédures externes du moniteur. Il faut donc en endormir un des deux, il existe deux possibilités :

1. `P` attend que `Q` abandonne le moniteur ou attend une autre condition.
2. `Q` attend que `P` abandonne le moniteur ou attend une autre condition.

Question 4.3 *Pour la sémantique que vous préciserez, donnez une implémentation : d'une part des primitives* `signaler` *et* `attendre` *pour une condition ; d'autre part des procédures locales* `init(M)`, `entree(M)`, `sortie(M)`. *Vous pourrez utiliser :*
– un sémaphore pour gérer l'exclusion entre les procédures externes,
– un sémaphore et un compteur pour gérer la priorité entre `P` *et* `Q`,
– pour chaque condition, un sémaphore et un compteur du nombre de processus en attente.

Réponse à la question 4.3 (3 points) : Une solution pour que `P` attende que `Q` abandonne le moniteur.

```
procedure locale init();
  debut
  mutex : semaphore = (1,{});
  cond[1..n] : semaphore = (0,{});   // le premier wait doit être bloquant
  attente[1..n] : entier = 0;
  priorite : semaphore = (1,{});     // pour décider de la reprise entre P et Q
  nb : entier = 0;                   // pour compter les processus
  fin
primitive signaler(condition i);
  debut
  si attente[i] > 0 alors
    debut
    nb++;                     // un processus de plus en attente dans le moniteur
    signal(cond[i]);          // on reveille un processus en attente de la condition
    wait(priorite);           // j'attend que le processus reveillé termine.
    nb--;
    fin
  fin
primitive attendre(condition i);
  debut
  attente[i]++;
  si nb > 0 alors
    signal(priorite);         // on reveille un processus dont la condition est vraie
  sinon
    signal(mutex);            // on permet à un processus d'entrer dans le moniteur
  wait(cond[i]);              // car on sait que l'on va se bloquer.
  attente[i]--;
  fin
procedure locale entree();
  debut
  wait(mutex);                // on attend l'autorisation
  fin
procedure locale fin();
  debut
  si nb > 0 alors
    signal(priorite);         // on reveille un processus dont la condition est vraie
  sinon
    signal(mutex);            // on permet à un processus d'entrer dans le moniteur
  fin
```