System d'exploitation avancé

Semaphores

EXERCICES & CORRIGES

Contenu

I. EXERCICES :

I.1 Synchronisation : le coiffeur

Une illustration classique du problème de la synchronisation est celui du salon de coiffure. Dans le salon de coiffure, il y a un coiffeur C, un fauteuil F dans lequel se met le client pour être coiffé et N sièges pour attendre.

- S'il n'a pas de clients, le coiffeur C somnole dans le fauteuil F.
- Quand un client arrive et que le coiffeur C dort, il le réveille, C se lève. Le client s'assied dans F et se fait coiffer.
- Si un client arrive pendant que le coiffeur travaille :
    - si un des N sièges est libre, il s'assied et attend,
    - sinon il ressort.

Il s'agit de synchroniser les activités du coiffeur et de ses clients avec des sémaphores. Les sémaphores utilisés sont initialisés ainsi :

```
Init (SCF, 0);
Init (SP, 0);
Init (SX, 1);
```

Programme client :	Programme coiffeur :

`Client() {`
`  P(SX);`
`  if(Attend < N) {`
`  Attend = Attend + 1;`
`  V (SP);`
`  V (SX);`
`  P (SCF);`
`  SeFaireCoifferEtSortir();`
`  }`
`  else {`
`  V(SX);`
`  Sortir();`
`  }`
`}`
	`Coiffeur(){`
	`  while (1){`
	`  P(SP);`
	`  P(SX);`
	`  Attend = Attend -1;`
	`  V(SCF);`
	`  V(SX);`
	`  Coiffer();`
	`  }`
	`}`

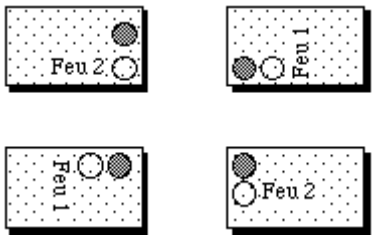
a- Détailler le fonctionnement du coiffeur et de ses clients tels qu'ils sont représentés par les deux fonctions `Coiffeur` et `Client`.

b- Quel est le rôle de chacun des sémaphores SCF, SP et SX ?

I.2 Synchronisation : le carrefour

Le carrefour ou le problème de la gestion des feux de circulation.

La circulation au carrefour de deux voies est réglée par des signaux lumineux (feu vert/rouge). On suppose que les voitures traversent le carrefour en ligne droite et que l**e carrefour peut contenir au plus une voiture à la fois**.



On impose les conditions suivantes :

- toute voiture se présentant au carrefour le franchit en un temps fini,
- les feux de chaque voie passent alternativement du vert au rouge, chaque couleur étant maintenue pendant un temps fini.

Les arrivées sur les deux voies sont réparties de façon quelconque. Le fonctionnement de ce système peut être modélisé par un ensemble de processus parallèles :

- un processus `P` qui exécute la procédure `Changement` de commande des feux;
- un processus est associé à chaque voiture;
- le traversée du carrefour par une voiture qui circule sur la voie i (i = 1, 2) correspond à l'exécution d'une procédure `Traverseei` par le processus associé à cette voiture.

On demande d'écrire le programme `Changement` ainsi que les procédures `Traversee1` et `Traversee2`.

Remarque :
Le processus P doit attendre que la voiture engagée dans le carrefour en soit sortie avant d'ouvrir le passage sur l'autre voie.

I.3 Exécutions atomiques

On suppose que sur Unix on peut définir des variables `x` et `y` communes à deux processus comme suit :

```
shared long x = 0 ;
shared long y = 0 ;
```

Deux processus exécutent les codes suivants :

```
   Processus P1                    Processus P2
***                                ***
x = x + 1;                         x = x * 2;
y = y + 1;                         y = y * 2;
printf("x=%d,y=%d\n", x, y);       printf("x=%d,y=%d\n", x, y);
***                                ***
```

a- Ces processus s'exécutent sur un système UNIX dont la politique d'ordonnancement est du type *round robin* . Quelles peuvent être les couples de valeurs affichées par chacun des deux processus ?

b- En utilisant un sémaphore, modifier le code pour assurer que les `printf` affichent toujours des valeurs identiques pour `x` et `y`.

I.4 Quantum de temps

Dans le cas de la stratégie d'allocation du processeur avec recyclage (algorithme du tourniquet, ou encore algorithme du quantum de temps), indiquer quels sont les effets des choix suivants pour le quantum `q`, sachant que `s` est le temps de changement de contexte et que `t` est le temps moyen d'utilisation du processeur entre deux événements bloquants (t `>>` `s` et [[epsilon]] `<<` `s`) :

1. q = [[infini]]
2. q = [[epsilon]]
3. q = s
4. q = t
5. s < q <t
6. q > t

Pour chaque question, étudier les cas où s compris dans le quantum ou non.

I.5Ordonnancement

Soit **TS** le temps de service d'un travail, c'est à dire le temps écoulé entre la soumission du travail et sa fin. On considère un système de traitement séquentiel (*batch*) dans lequel quatre travaux arrivent dans l'ordre suivant :

NO du travail	Instant d'arrivée	Durée
1	0	8
2	1	4
3	2	9
4	3	5

a- Donner le **TS** moyen dans le cas où l'on adopte la politique PAPS (Premier Arrivé, Premier Servi, ou encore FCFS, *Fist Come Fisrt Served*)

b- Donner le **TS** moyen dans le cas où l'on adopte la politique préemptive : PCA (le plus court d'abord, ou encore SJN, *Shortest Job Next*)

I.6 Synchronisation : barrière

Ecrire les procédures d'accès à un objet de type barrière, c'est à dire dont le fonctionnement est le suivant :

1. la barrière est fermée à son initialisation,
2. elle s'ouvre lorsque N processus sont bloqués sur elle. Définition de la barrière :

```
struct
  {
   Sema Sema1;        /* Sémaphore de section critique */
   Sema Sema2;        /* Sémaphore de blocage sur la barriere */
   int  Count;        /* Compteur */
   int  Maximum;      /* Valeur déclenchant l'ouverture */
  } Barriere;
```

Question 1 :

Complétez le code de la procédure d'initialisation :

```
void Init (Barrière *B; int Maximum)
    {
     Init (B->Sema1,  );
     Init (B->Sema2,  );
     B->Count    =     ;
     B->Maximum  =     ;
    }
```

Question 2 :

Complétez le code de la procédure d'attente donné ci-dessous :

```
void Wait (Barriere *B)
{
Boolean Do_Wait = True;
int I;
***;
B->Count++;
if (B->Count == B->Maximum )
   {
    for (I=0 ; I < (B->Maximum-1) ; I++ ) ***;
    B->Count = 0;
    Do_Wait = ***;
   }
***;
if (Do_Wait) ***;
}
```

I.7 Synchronisation

On se propose ici d'écrire deux procédures `Wait (P,V)` et `Set (P,V)` :

1. `Wait` permet à un processus d'attendre qu'une variable P passe à une valeur V,
2. `Set` positionne P à la valeur V et libère tous les processus bloqués par la procédure `Wait`.

Ces procédures pouvant être utilisées simultanément par plusieurs processus, il faudra être particulièrement attentif aux problèmes de synchronisation.

On donne ci-dessous le canevas (pseudo C) de `Wait` et `Set` ainsi que les structures de données utilisées.

```
/*-------------------------------------------------------------------
  Structure de données utilisée pour gerer P
-------------------------------------------------------------------*/
struct {
   Sema Sema1;        /* Semaphore pour gérer la section critique
*/
   Sema Sema2;        /* Semaphore d'attente de remise a jour */
   int  Count;        /* Nombre de processus en attente */
   int  X;            /* Donnee protegee */
} Protected;
```

La fonction `Set` :

```
/*-------------------------------------------------------------------
   Set fait passer la variable P a la valeur V
   et libere ensuite tous les processus qui attendent le changement
   de valeur de P.

-------------------------------------------------------------------*/
void Set (Protected *P; int Valeur)
{
   int I;
    ***;
   P->X = Valeur;
   for (I=0 ; I < P->Count ; I++)    ***;
     P->Count = 0;
   ***;
}
```

La fonction `Wait`:

```
/*-------------------------------------------------------------------
   Wait bloque un processus tant que la variable X ne vaut pas V
   Le nombre de processus bloques est memorise dans Count

-------------------------------------------------------------------*/
void Wait (Protected *P; int Valeur)
{
   Boolean Do_Wait = True;
   while Do_Wait
   {
   ***;
      if (P->X == Valeur) {
         Do_Wait = ***;
      } else {
         P->Count++;
      }
   ***;
   if Do_Wait    ***;      /*  (1)   */
   }
}
```

Question 1 :

Comment doivent être initialisés Sema1 et Sema2 ?

Question 2 :

Compléter `Wait` en remplaçant les `***` par le code adéquat.

Question 3 :
Compléter `Set` en remplaçant les `***` par le code adéquat.

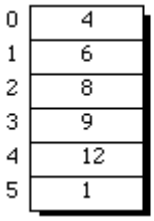
Question 4 :
Que se passe-t-il dans ce scénario **très particulier** :

- un processus P1 se bloque sur l'instruction commentée **(1)** dans `Wait`
- un processus P2 exécute ensuite intégralement `Set`
- un processus P3 fait alors appel à `Wait` plus précisément qu'arrive-t-il à P3 ?

Suggérez une solution en utilisant un troisième sémaphore qui assure le bon fonctionnement de l'instruction commentée **(1).**

I.8 Pagination

Soit la table de pages suivante :



Sachant que les pages virtuelles et physiques font 1K octets, quelle est l'adresse mémoire correspondant à chacune des adresses virtuelles suivantes codées en hexadécimal :
`142A` et `0AF1`

I.9 Gestion mémoire

Comment implanter le partage de code (code réentrant) dans un système qui gère la mémoire par pagination ?

I.10 Pagination 2

Un programme a besoin de 5 pages virtuelles numérotées 0 à 4. Au cours de son déroulement, il utilise les pages dans l'ordre suivant : **`012301401234`**

Question :

1. S'il reste 3 pages libre en mémoire, indiquer la séquence des défauts de page, sachant que l'algorithme de remplacement est FIFO,
2. Même question avec 4 pages disponibles en mémoire. Observation ?

I.11 Gestion de la pile

On donne le programme C suivant :

```
#include
int main (void){
   short i, *P ;
   short *Fonc (void);
   P = Fonc ();
   for (i=0; i <5 ;i++) printf ("P[%d] = %d\n", i, P[i]);
}

short *Fonc (void) {
   short  i, T[5];
   for (i=0; i <5 ;i++) T[i] = i;
   return (T);
}
```

Question :

En exécutant le programme, on obtient l'affichage suivant, pourquoi ?

```
P[0] = 0
P[1] = -2856
P[2] = 1
P[3] = 1764
P[4] = 4
```

II.CORRIGES

II.1 Synchronisation : le coiffeur

a- Coiffeur

- prend le 1er client de la file, s'il y en a un (sinon il se bloque)
- décrémente Attend (accès exclusif)
- libère le siège du client (l'invite à s'asseoir)
- Coiffe

Client
incrémente Attend (accès exclusif)

- Si le nombre de clients en attente est supérieur à N, sort
- s'ajoute à la file d'attente
- attend de pouvoir s'asseoir dans le siège du coiffeur
- se fait coiffer et sort.

b- Rôle de chacun des sémaphores SCF, SP et SX :

- SX : assure l'accès en exclusion mutuelle à la variable `Attend`. `Attend`permet de contrôler la taille maximale de la file d'attente
- SP : gère la file des clients en attente. Indique le nombre de ressources pour le coiffeur, c'est à dire le nombre de clients en attente,
- SCF : gère l'accès au fauteil du coiffeur. Indique si le coiffeur est prêt à travailler ou non (il est la essource unique des clients).

II.2 Synchronisation : le carrefour

Sémaphores utilisés :

```
Init (X1, 1), Init (X2, 1), Init (SF1, 1), Init (SF2, 0);

Changement                          Traversee1        Traversee2
{int Feu = 1;                       { P(SX1);         { P(SX2);
 while(1)                             P(SF1);           P(SF2);
 {sleep(Duree_du_feu);                Traversee();      Traversee();
   if  Feu == 1)                      V(SF1);           V(SF2);
   { P(SF1); V(SF2); Feu = 2;}        V(SX1);           V(SX2);
   else                              }                 }
   { P(SF2); V(SF1); Feu = 1;}
 }
}
```

SX1 et SX2 sont introduits pour éviter que les voitures attendent sur SF1 et SF2 et bloquent de ce fait les changements de feu effectués par `Changement`.

II.3 Exécutions atomiques

Réponse :

a- Couples possibles au final :

```
x = 1, y = 1
x = 2, y = 2
x = 1, y = 2
x = 2, y = 1
```

b- On utilise un sémaphore S initialisé ainsi : `Init(S, 1).`

```
   Processus P1
***
 P (S);
x = x + 1;
y = y + 1;
V (S);
printf("x=%d,y=%d\n", x, y);
***
```

```
 Processus P2
***
P (S);
x = x * 2;
y = y * 2;
V (S);
printf("x=%d,y=%d\n", x, y);
***
```

II.4 Quantum de temps

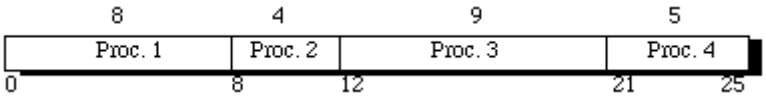
Réponses :

1. Le processus garde le processeur tant qu'il en a besoin (comme FCFS, *Fist Come Fisrt Served*),
2. Le processus ne fait presque rien entre chaque changement de contexte, progression très lente. Si s est compté dans q, aucun processus n'est exécuté.
3. Si s est compris dans q, il ne se passe rien, sinon exécution pendant au plus s,
4. Le quantum a tendance à favoriser les processus orientés entrées-sorties,
5. Le quantum est quelconque,
6. Le quantum favorise les processus qui ne font que du calcul,

II.5 Ordonnancement

Réponses :

a- Avec FCFS, le schéma d'ordonnancement est le suivant :



on obtient : ((8) + ((8 - 1) + 4) + (12 - 2) + 9) + ((21 - 3) + 5)) / 4 = 61 /4

b- Avec SJN, le schéma d'ordonnancement est le suivant :

- 1 commence en 0, est interrompue par 2 au temps 1,
- 2 commence en 1 et fini en 5, sans être interrompue puisqu'elle est toujours la plus courte,
- 4 commence après 2, au temps 5, puisqu'elle est la SJN à cet instant, elle finit au temps 10,
- 1 recommence alors au temps 10 et se termine en 17,
- 3 s'exécute de 17 à 26,

on obtient :
((17 - 0 + 1) + (4 - 1+ 1) + (9 - 3 + 1) + (25 - 2 + 1)) / 4 = 53 /4

II.6 Synchronisation : barrière

Complétez le code de la procédure d'initialisation :

```
void Init (Barriere *B; int Maximum)
{
   Init (B->Sema1, 1);
   Init (B->Sema2, 0);
   B->Count    = 0 ;
   B->Maximum  = N ;
}
```

Complétez le code de la procédure d'attente :

```
void Wait (Barriere *B)
{
Boolean Do_Wait = True;
int I;
P (P->Sema1);
B->Count++;
if (B->Count == B->Maximum )
   {
    for (I=0 ; I < (B->Maximum-1) ; I++ ) V (P->Sema2);
    B->Count = 0;
    Do_Wait = false;
   }
V (P->Sema1);
if (Do_Wait) P (P->Sema2);
}
```

II.7 Synchronisation

Compléter `Wait` :

```
/*-----------------------------------------------------------------
   Wait bloque un processus tant que la variable X ne vaut pas V
   Le nombre de processus bloques est memorise dans Count
-----------------------------------------------------------------*/
void Wait (Protected *P; int Valeur)
```

```
{
   Boolean Do_Wait = True;
   while Do_Wait
   {
   P (P->Sema1);
      if (P->X == Valeur) {
          Do_Wait = False;
      } else {
         P->Count++;
      }
   V (P->Sema1);
      if Do_Wait    P (P->Sema2);       /*  (1) */
   }
}
```

Compléter `Set` :

```
/*-------------------------------------------------------------------
   Set fait passer la variable P a la valeur V
   et libere ensuite tous les processus qui attendent le changement
   de valeur de P.

-------------------------------------------------------------------*/
void Set (Protected *P; int Valeur)
{
   int I;



   P (P->Sema1);

   P->X = Valeur;

   for (I=0 ; I < P->Count ; I++)    V (P->Sema2);

   P->Count = 0;

   V (P->Sema1);

}
```

Question I-4 :

Scénario très particulier :

- un processus P1 se bloque sur l'instruction commentée **(1)** dans `Wait`
- un processus P2 exécute ensuite intégralement `Set`
- un processus P3 fait alors appel à `Wait` plus précisément qu'arrive-t-il à P3 ?

Solution :

dans `Set` :

```
...
for (I=0 ; I < P->Count ; I++)
                { V (P->Sema2); P (S3);}
...
```

dans `Wait` :

```
if Do_Wait    { P (P->Sema2); V (S3)} /* (1) */
```

II.8 Pagination

Réponses :
1K = 1024 = 2^{10}, le déplacement dans une page est donc codé sur 10 bits.
page virtuelle 5, octet 2A dans cette page -> page mémoire 1, octet 2A dans cette page
page virtuelle 2, octet 2F1 dans cette page ->page mémoire 8, octet 2F1 dans cette page

II.9 Gestion mémoire

Partage de code :

Réponse :

il suffit de faire pointer les pages virtuelles de deux processus sur les memes pages physiques.

II.10 Pagination 2:

Il utilise les pages dans l'ordre suivant :
`012301401234`

Réponses :

- 3 pages libre en mémoire:

Contenu de page mém. 1 :	0	3	3	3	4	4	4
Contenu de page mém. 2 :	1	1	0	0	0	2	2
Contenu de page mém. 3 :	2	2	2	1	1	1	3
Défaut sur la page virtuelle :		3	0	1	4	2	3

-> 9 défauts de page (3 pour remplir la mémoire, puis 6).

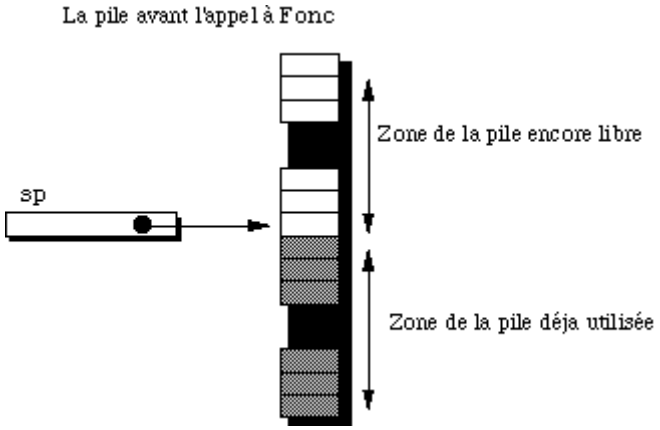
- 4 pages libre en mémoire:

Contenu de page mém. 1 :	0	4	4	4	4	3	3
Contenu de page mém. 2 :	1	1	0	0	0	0	4
Contenu de page mém. 3 :	2	2	2	1	1	1	1
Contenu de page mém. 4 :	3	3	3	3	2	2	2
Défaut sur la page virtuelle :		4	0	1	2	3	4

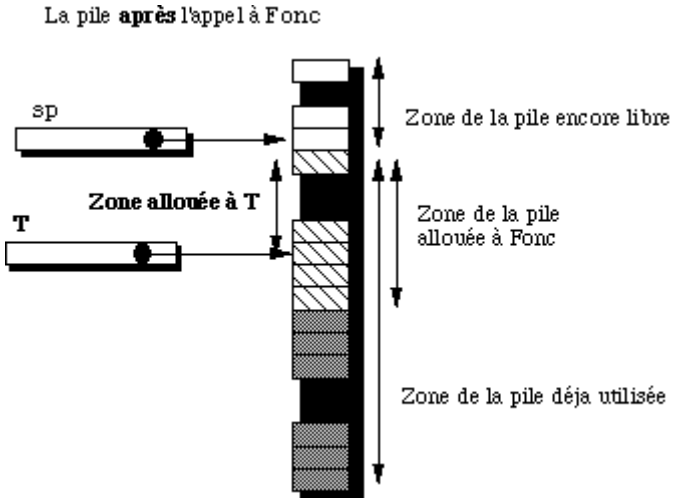
-> 10 défauts de page (4 pour remplir la mémoire, puis 6)

I.11 Gestion de la pile

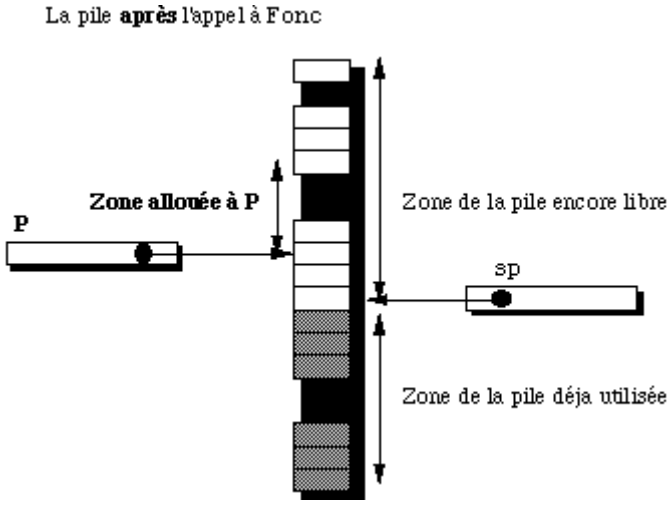
Avant l'appel à la fonction `Fonc`, le pointeur de pile (`stack pointer`), `sp`, contient l'adresse de la première case libre sur la pile.
On schématise ci-dessous l'état de la pile avant l'appelà `Fonc`. Les cases libres sont en blanc, les cases occupées sont en grisé.
Chaque case représente un octet, le pointeur de pile contient donc des adresses d'octets.



T est une variable locale de la fonction `Fonc`, l'espace mémoire réservé à ce tableau est donc alloué sur la pile.



La fonction `Fonc` renvoie donc une adresse dans la pile.
Lorsqu'on sort de la fonction `Fonc`, le système dépile, c'est à dire qu'il remet le pointeur de pile à la valeur qu'il avait avant que l'on rentre dans `Fonc`.



Si l'on a à nouveau besoin de la pile, ici pour l'appel à `printf`, l'espace disponible sur la pile va être réutilisé, et le tableau T, c'est à dire l'espace mémoire dont la taille est :
`5 * sizeof(short)`,
alloué à partir de l'adresse contenue dans T, va donc être "écrasé" par de nouvelles variables temporaires.
T contient donc l'adresse d'une zone dans laquelle on trouvera des valeurs différentes à chaque fois qu'elle est réallouée à de nouvelles variables.