École Polytechnique de Montréal

Département de Génie Informatique et de Génie Logiciel

Cours INF2610

Examen final

Automne 2010

• Date : **16 décembre 2010** de **9h30 à 12h00**	• Pondération : 4**0 %**
• Professeur : **Boucheneb Hanifa**	• Nbre. de questions : **5**
• Documentation permise : Polycopié du cours	• Total : **20 points**
• Calculatrices programmables et cellulaires non permis	

Question 1 (4 pts) : Généralités

Répondez aux questions suivantes (les réponses doivent être justifiées, concises et claires) :

a) [1.5 pt] Donnez l'arborescence des processus créés par le bout de code suivant (supposez que l'appel à « fork » ne retourne pas d'erreur) :

```
int i, pid, n=0;
 for (i=0; i<3; i++)
{          pid = fork();
           if (pid != 0) break;
           else n=n+i;
}
printf("n = %d  pour processus  %d\n", n, pid);
```

Donnez, pour chaque processus (y compris le processus principal), la (ou les) valeur(s) de la variable **n** affichées à l'écran.

b) [1 pt] Sous Windows, deux processus père et fils peuvent-ils partager le même pointeur de fichier ? Justifiez votre réponse.

c) [1.5 pt] Expliquez brièvement comment le langage Java implémente les moniteurs. Cette implémentation de moniteurs correspond-elle à un problème classique de synchronisation (vu en classe) ? Justifiez votre réponse.

Question 2 (4 pts) : Moniteurs et variables de condition

On veut synchroniser *3* threads *th1*, *th2* et *th3* d'un même processus en utilisant un moniteur et des variables de condition. La fonction *Fi, i=1,3*, exécutée par chaque thread *thi*, consiste en une infinité de cycles. Chaque cycle est une section critique qui doit être exécutée en exclusion mutuelle. De plus, chaque cycle de *th3* doit être précédé d'un cycle de *th1* et d'un cycle de *th2*.

Complétez le pseudocode suivant pour synchroniser, en utilisant les variables de condition, les cycles des threads *th1, th2* et *th3*.

```
Moniteur SynCycles
{       /*0*/
        Function F1()  // fonction de th1
        {   while (1)  {       /*1*/
                Sc1();  // section critique de th1
                /*2*/
            }
         }
        Function F2()  // fonction de th2
        {  while (1)   {      /*3*/
                Sc2();  // section critique de th2
                /*4*/
             }
         }
        Function F3()  // fonction de th3
        {  while (1)  {      /*5*/
                Sc3();  // section critique de th3
                /*6*/
            }
         }
}
```

Question 3 (5 pts) : Sémaphore

On veut implanter une pile partagée entre plusieurs threads d'un même processus. Les fonctions *empiler* et *depiler* peuvent être appelées simultanément par plusieurs threads du processus. La fonction *empiler* bloque jusqu'à ce qu'un espace soit libre dans la pile. De son côté, la fonction *depiler* bloque jusqu'à ce qu'une donnée soit disponible dans la pile.

a) [2 pts] Complétez, en utilisant les sémaphores, le code suivant de manière à satisfaire les contraintes de synchronisation des fonctions *empiler* et *dépiler*.

```
int sommet = 0;
int pile [ Size ];

/* 0 */ Semaphore  …  // Utilisez la structure Semaphore et les fonctions P et V.

void empiler (int a)
{
        /* 1 */
        pile[ sommet ] = a;
       sommet++;
        /* 2 */
}

int  depiler()
{
       /* 3 */
       sommet--;
       int tmp = pile[ sommet ];
        /* 4 */
        return tmp;
}
```

b) [3 pts] On veut maintenant permettre aux threads de partager une file circulaire, d'enfiler et de défiler un ou plusieurs éléments de la file. La fonction *enfiler* insère en queue de file, l'un à la suite de l'autre, les éléments d'un tableau. La fonction *defiler* récupère, dans un tableau, un ou plusieurs éléments de la file. Les fonctions *enfiler* et *defiler* peuvent être appelées simultanément par plusieurs threads du processus. La fonction *enfiler (int A[], int m)* bloque jusqu'à ce que l'espace nécessaire pour insérer les éléments du tableau *A* soit libre dans la file. De son côté, la fonction *defiler (int A[], int m)* bloque jusqu'à ce qu'il y ait au moins *m* éléments dans la file. Pour les deux fonctions, m est la dimension du tableau A. Sa valeur est supposée inférieure ou égale à la taille de la file. Elle est la même pour tous les appels aux fonctions *enfiler* et *defiler*.

   Complétez, en utilisant les sémaphores, le code suivant de manière à satisfaire les contraintes de synchronisation des fonctions *enfiler* et *defiler*.

```
int t = 0, q=0;
int file [ Size ];

/* 0 */ Semaphore … // Utilisez la structure Semaphore et les fonctions P et V.

void enfiler (int A[], int m)
{       int i;
        /* 1 */
        for(i=0; i<m;i++)
        {       file[q] = A[i];
                q=(q+1)%Size;
        }
        /* 2 */
}

void defiler (int A[], int m)
{        int i;
        /* 3 */
        for(i=0; i<m;i++)
        {       A[i] = file[t];
                t=(t+1)%Size;
        }
         /* 4 */
}
```

Question 4 (3.5 pts) : Interblocage

Soient 3 processus concurrents P1, P2 et P3 qui utilisent en exclusion mutuelle 6 ressources différentes (de R1 à R6). Les processus exécutent respectivement les codes suivants :

```
P1()
{
    while(1){
        prendre (R4);
        prendre (R5);
        prendre (R3);
     // Utiliser R4, R5, R3
        liberer(R4);
        liberer(R5);
        liberer(R3);
    }
}
```

```
P2()
{
    while(1){
        prendre (R3);
        prendre (R2);
        prendre (R6);
      //Utiliser R3, R2, R6
        liberer(R6);
        liberer(R2);
        liberer(R3);
    }
}
```

```
P3()
{
    while(1){
        prendre (R1);
        prendre (R2);
        prendre (R5);
      // Utiliser R1, R2, R5
        liberer(R5);
        liberer(R2);
        liberer(R1);
    }
}
```

La fonction *prendre* permet d'allouer une ressource au processus appelant, si cette dernière est libre. Dans le cas contraire, elle bloque le processus appelant jusqu'à l'obtention de la ressource demandée. Une ressource allouée est libérée par le processus détenteur lorsqu'il invoquera la fonction *liberer*.

Ces processus peuvent-ils entrer en interblocage ? Si oui, peut-on prévenir les interblocages ? Peut-on les éviter ? Justifiez vos réponses.

Question 5 (3.5 pts) : Ordonnancement de processus

a) [1.5 pts] Considérez un système monoprocesseur et les 5 processus P1, P2, P3, P4 et P5 décrits dans le Tableau 1 :

Processus	Date d'arrivée	Priorité	Temps d'exécution
P1	0	3	5 (2) 3
P2	1	3	6
P3	2	2	2 (3) 2
P4	4	1	2
P5	5	2	2

Tableau 1

Supposez que :

- le système dispose d'un seul périphérique d'E/S partagé entre les processus,
- le temps de commutation est égal à 0, et
- la priorité 1 est la plus basse.

Donnez le diagramme de Gantt montrant l'ordonnancement des processus dans le cas d'un ordonnancement préemptif à priorités fixes. L'ordonnancement des processus de même priorité est circulaire avec un quantum égal à 3.

b) [2 pts] Considérez les tâches décrites par les caractéristiques suivantes, partageant la même ressource R :

Processus	Date d'arrivée	Temps d'exécution	Deadline = Période
P1	3	3 ERE	6
P2	2	2 EE	8
P3	0	4 ERRE	12

Ces processus sont-ils ordonnançables RMA dans le cas où PIP (protocole d'héritage de priorités) est utilisé pour traiter les inversions de priorité ? L'intervalle d'étude à considérer est [0,27].

Le corrigé

Question 1 :

a)

P0 crée un processus F1 puis affiche n=0

F1 crée un processus F11 puis affiche n=0

F11 crée un processus F111 puis affiche n=1

F111 ne crée pas de processus mais affiche n=3

b)

Oui, il suffit, d'une part, de spécifier lors de la création ou l'ouverture du fichier que le handle est héritable par les processus fils et, d'autre part, d'indiquer lors de la création du fils qu'il hérite tous les objets marqués héritables.

c)

Il permet d'exécuter en exclusion mutuelle certaines méthodes (les méthodes de type synchronized) d'un même objet. Il associe à un chaque objet une variable de condition et deux files d'attente : Entry queue et wait queue. La première file sert à gérer les demandes d'accès aux méthodes de l'objet alors que la seconde est utilisée pour se mettre en attente de la variable de condition de l'objet. Le modèle de synchronisation utilisé est similaire à celui des lecteurs / rédacteurs d'une base de données. L'objet représente la base de données. Ses méthodes de type Synchronized sont les rédacteurs. Ses autres méthodes sont des lecteurs.

Question 2 : (4pts)

```
Moniteur SynCycles
{       /*0*/ boolc c1, c2, c3;
        bool  t1=1, t2=1, t3=0;   // ti =0 si ce n'est pas le tour de thi, ti=1 sinon.
        Function F1()  // fonction de th1
```

```
        {   while (1)
          {
                /*1*/ while (t1!=1)  c1.wait();
                Sc1();  // section critique de th1
                /*2*/    t1=0;
                        if (t2==0)  {
                                        t3=1;
                                        c3.signal();
                                    }
          }
         }
        Function F2()  // fonction de th2
        {   while (1)
          {
                /*3*/ while(t2!=1) c2.wait();
                Sc2();  // section critique de th2
                /*4*/  t2=0;
                    if( t1==0) {   t3 =1;
                                   c3.signal();
                                }
          }
         }
        Function F3()  // fonction de th3
        {   while (1)
          {
                /*5*/ while (t3!=1) c3.wait();
                Sc3();  // section critique de th3
                /*6*/ t3=0; t1=1; t2=1; c1.signal(); c2.signal();
          }
        }
}
```

Question 3 :

a)

```
int sommet = 0;
int pile [ Size ];

/* 0 */ Semaphore libre= Size, Occupe=0, mutex =1;

void empiler (int a)
{
        /* 1 */
        P(libre);
        P(mutex);
        pile[ sommet ] = a;
        sommet++;
        /* 2 */
        V(mutex);
        V(occupe);
}

int  depiler()
{
        /* 3 */
        P(occupe);
        P(mutex);
        sommet--;
        int tmp = pile[ sommet ];
         /* 4 */
        V(mutex);
        V(libre);
        return tmp;
}
```

b)

```
int t = 0, q=0;
int file [ Size ];

/* 0 */ Semaphore libre = Size, occupe=0, mutex1=1, mutex2=1;

void enfiler (int A[], int m)
{       int i;
        /* 1 */ P(mutex1);
        for(i=0; i<m; i++)  P(libre);
        for(i=0; i<m;i++)
        {       file[ q ] = A[i];
                q=(q+1)%Size;
        }
        /* 2 */ V(mutex1);
```

```
        for(i=0; i<m; i++) V(occupe);
}

void defiler (int A[], int m)
{
        /* 3 */ P(mutex2);
        for(i=0; i<m; i++)  P(occupe);
        for(int i=0; i<m;i++)
        {       A[i] = file[ t ];
                t=(t+1)%Size;
        }

         /* 4 */
        V(mutex2);
        for(i=0; i<m;i++) V(libre);
}
```

Question 4 : 3.5 pts

Oui, le scénario suivant mène vers un interblocage :
P1 prend R4 et R5;
P3 prend R1 et R2;
P2 prend R3;
P1 attend R3; (détenue par P2)
P3 attend R5; (détenue par P1)
P2 attend R2; (détenue par P3)
Oui, on peut les prévenir en ordonnant les ressources R1<R2<R3<R4<R5<R6 et en modifiant le code de manière à demander les ressources dans un ordre croissant :

P1()	P2()	P3()
{	{	{
while(1){	while(1){	while(1){
prendre (R3);	prendre (R2);	prendre (R1);
prendre (R4);	prendre (R3);	prendre (R2);
prendre (R5);	prendre (R6);	prendre (R5);
// Utiliser R4, R5, R3	//Utiliser R3, R2, R6	//Utiliser R1, R2, R5
liberer(R4);	liberer(R6);	liberer(R5);
liberer(R5);	liberer(R2);	liberer(R2);
liberer(R3);	liberer(R3);	liberer(R1);
}	}	}
}	}	}

Oui, on peut les éviter en appliquant l'algorithme du banquier car on connait les besoins maximaux des processus. Au départ toutes les ressources sont libres et la matrice Alloc est nulle :

A = (1,1,1,1,1,1),

0 0 0 0 0 0
Alloc = 0 0 0 0 0 0
0 0 0 0 0 0

0 0 1 1 1 0
Req = 0 1 1 0 0 1
1 1 0 0 1 0

Question 5 :

a)

P1	P1	P1	P2	P2	P2	P1	P1	P2	P2	P2	P1	P1	P1	P3	P3	P5	P5	P4	P3	P3	P4
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

b)

			E		R	E			E	R	E				E	R	E				E		R	E			
		E					E					E	E					E	E								
E	R			R				E						E						R		R		X			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
P3		P2	P1						P1	P2		P3			P1			P2			P1			P3		P2	P1