École Polytechnique de Montréal

Département de Génie Informatique et Génie Logiciel

Cours INF2610

Noyau d'un système d'exploitation

Contrôle périodique

Automne 2015

• Date : **30 octobre** de 18**h à 20h**	• Pondération : **30 %**
• Professeur : Hanifa Boucheneb	• Nombre de questions : **3**
• Toute documentation permise	• Total : **20 points**
• Calculatrices, cellulaires, ordinateurs non permis	

Question 1 : **(7 pts) Généralités**

Supposez un processus principal PP composé d'un ensemble de threads gérés par le noyau. Justifiez vos réponses à toutes les questions suivantes.

a) [**2.5 pts]** Indiquez pour chaque événement suivant l'effet sur le thread à l'origine de l'événement, les autres threads du processus ou/et le processus (la réponse à chaque sous question doit être claire et concise de maximum 4 lignes) :

1- Un thread de PP modifie une variable globale du processus PP.
La variable est partagée par tous les threads. La modification de cette variable par un thread sera visible par tous les autres.

2- Un thread de PP se termine par exit.
Le processus et tous ses threads vont se terminer (exit termine un processus).

3- Un thread de PP crée un pipe, en récupérant dans une variable globale « int fd[2]; » les deux descripteurs de fichiers du pipe, puis ferme le descripteur de lecture du pipe.
Le pipe devient non accessible en lecture.

4- Un thread de PP crée un pipe, en récupérant dans une variable globale « int fd[2]; » les deux descripteurs de fichiers du pipe puis lit du pipe :
« while(read(fd[0],&c,1) >0) write(1,&c,1) ; ».
L'appel read(fd[0],&c,1) va bloquer le thread car il y a toujours un écrivain.

5- Un thread de PP crée un pipe, en récupérant dans une variable globale « int fd[2]; » les deux descripteurs de fichiers du pipe, ferme le descripteur d'écriture puis lit du pipe :
« while(read(fd[0],&c,1) >0) write(1,&c,1) ; ».
Le pipe est vide et n'a aucun écrivain. L'appel read(fd[0],&c,1) va retourner 0 (une fin de fichier).

6- Un thread de PP crée un pipe, en récupérant dans une variable globale « int fd[2]; » les deux descripteurs de fichiers du pipe, ferme le descripteur de lecture fd[0] puis écrit dans le pipe : « write(fd[1], &c,1); ».
Le pipe n'a aucun écrivain. L'appel write(fd[1], &c,1) va générer un signal SIGPIPE qui mettra fin au processus et ses threads.

7- Un thread de PP crée un pipe, en récupérant dans une variable globale « int fd[2]; » les deux descripteurs de fichiers du pipe, puis exécute l'instruction « dup2(fd[1],1); ».
La sortie standard du processus et de ses threads devient le pipe car tous les threads d'un processus partagent la table des descripteurs de fichiers du processus.

8- Un thread th1 de PP crée un autre thread th2 en lui passant comme paramètre l'adresse d'une variable locale v du thread th1. Le thread th2 modifie la variable

v.
Tous les threads d'un processus partagent l'espace d'adressage du processus. La modification de v par th2 sera donc visible par th1 (th2 pourra accéder à v tant que th1 n'a pas terminé son exécution).

9- Un thread appelle la fonction « pause() ; ».
Seul le thread appelant passe à l'état bloqué car le noyau alloue du temps CPU aux threads noyau.

b) **[2.25 pts]** Supposez maintenant que les threads sont implémentés au niveau utilisateur, indiquez parmi les événements précédents, ceux dont l'effet est différent.

Tous les threads utilisateur du processus partagent l'espace d'adressage et la table de descripteurs de fichiers du processus. Par contre, dans ce cas, le noyau alloue du temps CPU au processus. Si un thread du processus effectue un appel système bloquant, le processus passe à l'état bloqué.
Les événements 4 et 9 (qui se traduisent en un appel système bloquant) vont bloquer tout le processus. Les autres événements ont le même effet qu'en a).

c) **[2.25 pts]** Supposez finalement que les threads sont remplacés par des processus fils du processus principal. Indiquez, pour chacun des événements précédents, à l'exception de l'événement 8, l'effet sur le processus fils à l'origine de l'événement et les autres.
Les processus fils ne partagent ni l'espace d'adressage, ni la table de descripteurs de fichiers du processus père. Lorsqu'un processus fils est créé, l'espace d'adressage et la table de descripteurs de fichiers du père « sont clonés » pour le fils, selon le principe « copie-on-write ». Le noyau alloue du temps CPU à chaque processus (père et fils). Par conséquent :

1- Un processus modifie une variable globale.
La modification d'une variable globale du processus sera visible que pour le processus lui même.

2- Un processus se termine par exit.
Le processus passe à l'état zombie. Les autres processus pourront continuer leurs exécutions.

3- Un processus crée un pipe, en récupérant dans une variable globale « int fd[2]; » les deux descripteurs de fichiers du pipe, puis ferme le descripteur de lecture du pipe.
Le pipe devient non accessible en lecture. Il est non accessible pour PP et les autres processus fils.

4- Un processus crée un pipe, en récupérant dans une variable globale « int fd[2]; » les deux descripteurs de fichiers du pipe puis lit du pipe « while(read(fd[0],&c,1) >0) write(1,&c,1) ; ».
L'appel read(fd[0],&c,1) va bloquer le processus appelant (uniquement).

5- Un processus crée un pipe, en récupérant dans une variable globale « int fd[2]; » les deux descripteurs de fichiers du pipe, ferme le descripteur d'écriture puis lit du pipe « while(read(fd[0],&c,1) >0) write(1,&c,1) ; ».
Le pipe est vide et n'a aucun écrivain, l'appel read(fd[0],&c,1) va retourner 0 (une fin de fichier) pour le processus appelant.

6- Un processus crée un pipe, en récupérant dans une variable globale « int fd[2]; » les deux descripteurs de fichiers du pipe, ferme le descripteur de lecture fd[0] puis écrit dans le pipe « write(fd[1], &c,1); ».
L'appel write(fd[1], &c,1) va générer un signal SIGPIPE qui mettra fin au processus appelant (uniquement).

7- Un processus crée un pipe, en récupérant dans une variable globale « int fd[2]; » les deux descripteurs de fichiers du pipe, puis exécute l'instruction « dup2(fd[1],1); ».
La sortie standard du processus appelant (uniquement) devient le pipe.

8-

9- Un processus appelle la fonction « pause() ; ».
Seul le processus appelant passe à l'état bloqué.

Question 2 (6.5 pts) : Processus & tubes de communication

Considérez le code suivant :

```
int i;
char A[4]="ABC";
int main()
{
    for(i=0; i<2; i++)
        if (fork()==0)
            printf("%c \n", A[i]);
        else
            if (i==0)  printf("%c", A[i]);
    exit(0);
}
```

a) **[2 pts]** Donnez le nombre de processus créés par la fonction « main » ci-dessus ainsi que l'arborescence de ces processus. Indiquez, sur cette arborescence, les symboles affichés à l'écran par chacun des processus, y compris le processus principal.

3 processus sont crées.

PP crée deux fils F1 (affiche A avec un saut de ligne) et F2 (affiche AB avec un saut de ligne) PP affiche A. F1 crée F11 (affiche B avec un saut de ligne)

b) **[1.5 pt]** Modifiez le code afin de forcer chaque processus père à attendre la fin de tous ses fils avant de se terminer.

```
int i;
char A[4]=”ABC”;
bool parent;
int main()
{    for(i=0; i<2; i++)
            if (fork()==0)
            {  printf("%c \n", A[i]);  parent=false;}
            else  { if (i==0)  printf("%c", A[i]);  parent = true;}
    if (parent) while(wait(NULL)>0);
     exit(0);
}
```

c) **[3 pts]** Complétez le code initial de manière à ce que :

- les « printf » des différents processus fils soient redirigés vers un tube anonyme, créé par le processus principal et

- le processus principal lit et affiche à l’écran toutes les données insérées dans le pipe par les processus créés, juste avant de se terminer.

```
int i;
char A[4]=”ABC”;
int fd[2];
bool pp=true;  // pp=true pour le processus principal
int main()
{
   pipe(fd);
   for(i=0; i<2; i++)
         if (fork()==0)
         {
                  if (pp)  {dup2(fd[1],1); close(fd[1]); close(fd[0]); pp=false;}
                  printf("%c \n", A[i]);
          } else   if (i==0)  printf("%c", A[i]);
   if (pp)
   {char C; close(fd[1]);
            while(read(fd[0], &C,1))>0)  write(1,&C,1);
            close(fd[0];
   } else { fflush(stdout); close(1);  }
    exit(0);
}
```

Question 3 (6.5 pts) : Synchronisation des processus

a) **[3 pts]** Considérez la solution, utilisant les sémaphores, au problème producteurs/consommateurs, pour le cas de 2 producteurs P1 et P2, et un consommateur C.

```
const int N=100 ;
int tampon [N];
int ip=0;
Semaphore libre=N, occupe=0, mutex=1;
```

```
Producteur_Pi ( ) // i=1,2
{
   while (1)
  {      P(libre) ;
         P(mutex);
         produire(tampon, ip);
         ip = Mod(ip +1,N);
         V(mutex);
         V(occupe);
    }
}
```

```
Consommateur_C ( )
  {  int ic=0;
     while (1)
     {   P(occupe);
         P(mutex);
         consommer(tampon,ic);
         ic= Mod(ic+1, N);
         V(mutex);
         V(libre);
     }
 }
```

1- **[1.5 pt**] Modifiez ce code pour que les producteurs P1 et P2 produisent en alternance dans le tampon (une production de P1 suivie d'une production de P2, …).

Le code du consommateur est le même. Pour les producteurs, deux sémaphores supplémentaires sont utilisés pour forcer l'ordre des productions.

```
const int N=100 ;
int tampon [N];
int ip=0;
Semaphore libre=N, occupe=0, mutex=1, S1=1, S2=0;
```

```
Producteur_P1( )
{  while (1)
  {      P(S1) ;
         P(libre) ;
         P(mutex);
         produire(tampon, ip);
         ip = Mod(ip +1,N);
         V(mutex);
         V(occupe);
          V(S2) ;
    }
}
```

```
Producteur_P2( )
{  while (1)
  {      P(S2) ;
         P(libre) ;
         P(mutex);
         produire(tampon, ip);
         ip = Mod(ip +1,N);
         V(mutex);
         V(occupe);
          V(S1) ;
    }
}
```

2- **[1.5 pt]** Est-il plus intéressant d'utiliser deux tampons (un tampon pour chaque producteur) et de forcer le consommateur à consommer en alternance des tampons de P1 et P2 (en commençant par celui de P1) ? Justifiez votre réponse.

Dans la solution précédente, il ne peut y avoir de productions ou consommations en parallèle (une à la fois). L'utilisation de deux tampons permet aux producteurs de produire en parallèle. Pendant qu'un producteur (ou le consommateur) utilise un tampon, l'autre producteur pourrait produire dans l'autre tampon. Les producteurs et le consommateur passeront ainsi moins de temps à l'état bloqué. De ce point de vue, cette solution est plus intéressante. Elle nécessite par contre plus de ressources : 2 tampons, 4 sémaphores compteurs (libre1, libre2, occupe1, occupe2) et 2 sémaphores binaires (mutex1, mutex2) vs. 1 tampon, 2 sémaphores compteurs (libre, occupe) et 3 sémaphores binaires (mutex, S1, S2).

b) **[3.5 pts]** On veut implémenter les sémaphores compteurs sem_c en utilisant les sémaphores binaires sem_b. Complétez la structure sem_c et les fonctions Init, P et V suivantes. Supposez que vous disposez des fonctions Initb (sem_b S, int v), Pb(sem_b S) et Vb(sem_b S) qui permettent respectivement d'initialiser un sémaphore binaire et d'appliquer les fonctions P et V à un sémaphore binaire.

```
Struct sem_c {   int val ;  // valeur du sémaphore   …. }
Init (sem_c S, int v0) {  … }
P(sem_c S)  { … }
V(sem_c S)  { … }
```

```
Struct sem_c
{   int val ;  // valeur courante du sémaphore
  int nbwait ;   //  nombre de processus/threads en attente du sémaphore
  sem_b mutex;  // pour protéger les sections critiques de P et V
  sem_b swait;  // pour bloquer des processus/threads (les mettre en attente de S)
}

Init (sem_c S, int v0)
{  S.val=v0 ; S.nbwait= 0 ; initb(S.mutex,1) ; initb(S.swait,0) ; }

P(sem_c S)
 {   Pb(S.mutex) ;
     if (S.val >0)  { S.val-- ; Vb(S.mutex) ;}
    else {S.nbwait++ ; Vb(S.mutex) ; Pb(S.swait) };
}

V(sem_c S)
 {   Pb(S.mutex) ;
     if (S.nbwait >0) {S.nbwait--; V(S.swait);}
     else S.val++;
    Vb(S.mutex) ;
}
```