École Polytechnique de Montréal

Département de Génie Informatique et Génie Logiciel

Cours INF2610

Contrôle périodique

Automne 2011

Question 1 (3 pts) : Généralités

Répondez aux questions suivantes (maximum 10 lignes par question). Les réponses doivent être justifiées, **concises et claires**.

1) [1 pt] Est-ce qu'un processus UNIX/Linux peut récupérer l'état de terminaison d'un de ses processus fils ? Justifiez votre réponse.

2) [1 pt] Un processus UNIX/Linux préserve-t-il sa table de descripteurs de fichiers suite à un appel à la fonction « execl » ? Justifiez votre réponse.

3) [1 pt] Deux threads appartenant à un même processus peuvent-ils se synchroniser au moyen d'un sémaphore du noyau, s'ils sont implémentés au niveau utilisateur ? Supposez que ce sémaphore est utilisé uniquement par ces deux threads.

Question 2 (6 pts) : Processus

Considérez le programme suivant :

```
#include  "includes.h"
#define MaxJoueurs 4
void jouer(int NJ)    //NJ est le numéro du joueur  0..MaxJoueurs-1
{
   sleep(3);  // simule une partie de jeu
}
```

```
/*0*/
int main( )
{
      // créer MaxJoueurs processus fils du processus principal
      // chaque joueur exécute la fonction jouer puis se termine
      /*1*/
      while(1)
      {
        // attendre la fin d'un processus joueur
        // lorsqu'un joueur  de numéro NJ se termine, un autre joueur de même
        // numéro (NJ) est créé.
       /*2*/
      }

      return 0;
}
```

1) [2.5 pts] Complétez le programme ci-dessus en y ajoutant le code qui réalise exactement les traitements spécifiés sous forme de commentaires.
2) [1 pt] On veut limiter le nombre total de remplacements de processus « joueur » à 1000. **Modifiez/complétez le code obtenu en 1) pour inclure cette directive.**
3) [2.5 pts] On veut maintenant utiliser la fonction ”alarm” pour limiter la durée totale du jeu à 60 secondes (alarm(60);). Cette fonction est appelée juste avant l'instruction ”while”. Au bout de ces 60 secondes, le processus principal doit stopper le remplacement de processus, attendre la terminaison des joueurs en cours puis se terminer. **Modifiez/complétez le code obtenu en 2) pour inclure cette nouvelle directive.**

La fonction *int alarm(int nb_sec)* programme une temporisation pour qu'elle envoie un signal *SIGALRM* au processus appelant dans *nb_sec* secondes. La valeur de retour de la fonction *alarm* ne nous intéresse pas dans le cadre de cette question. Le traitement par défaut du signal *SIGLARM* est : terminer le processus récepteur.

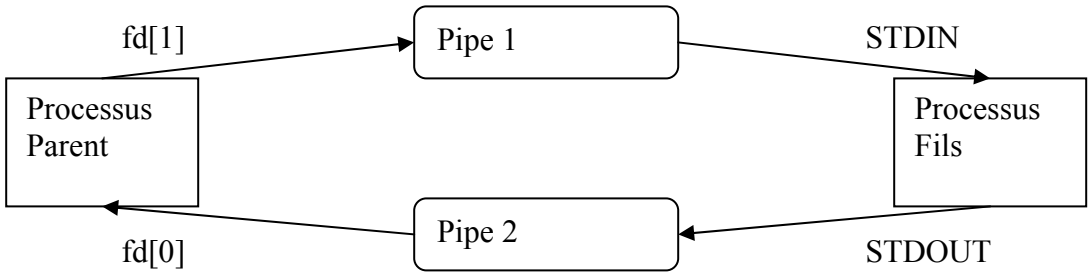
Question 3 (4 pts) : Redirection des entrées/sorties standards et pipes

On veut implémenter la fonction DuplexPipe qui crée un processus fils du processus appelant et deux pipes anonymes pour la communication entre les processus Parent et fils (un pipe pour chaque sens de communication).
La fonction DuplexPipe est définie comme suit :

pid_t DuplexPipe (int fd[2], char * com[ ]) ;
où :

- fd est utilisé pour récupérer les descripteurs nécessaires au processus appelant pour lire et écrire dans les pipes, et
- com est une commande composée d'un nom de fichier exécutable suivi éventuellement de paramètres.

DuplexPipe crée deux pipes anonymes et un processus fils. Le processus fils redirige son entrée et sa sortie standards vers les tubes créés comme le montre le schéma suivant. Il se transforme ensuite pour exécuter la commande com.



Vous supposez que tous les appels système utilisés dans cette fonction réussissent. Vous ne devez pas gérer les erreurs.
La fonction DuplexPipe retourne le pid du fils créé. Les descripteurs nécessaires au père pour communiquer avec son fils sont retournés dans le tableau fd.

```
pid_t DuplexPipe(int fd[2], char * com[ ] )
{    pid_t pid ;
        /*0*/
        pid = fork() ;
        /*1*/
   return pid ;
}
```

Complétez la fonction DuplexPipe.

Question 4 (7 pts) : Synchronisation

1) [4 pts] Considérez les 3 processus A, B et C concurrents suivants :

A	B	C
a1; a2;	b1; b2;	c1; c2; c3;

Où ai, bi et ci sont des actions atomiques.

.

a) Synchronisez, en utilisant les sémaphores, les processus A, B et C de manière à empêcher l'entrelacement des actions de A, B et C.
b) Utilisez les sémaphores pour bloquer un processus (A, B ou C) tant que les deux autres exécutent leurs actions (en concurrence).
c) Synchronisez, en utilisant les sémaphores, les processus A, B et C de manière à empêcher l'exécution des actions des processus A et B durant l'exécution des actions de C. Les actions de A et B s'exécutent en concurrence.

d) Utilisez les sémaphores pour forcer l'exécution des actions du processus A avant celles de B et C. Les actions de B et C s'exécutent en concurrence.

2) [3 pts] On désire implémenter une structure de données de type pile partagée. Plusieurs threads peuvent empiler des données sur la pile, en utilisant la fonction *push()* ou d'extraire des données de la pile, par la fonction *pop()*. La pile est représentée par un tableau d'entiers de taille N. Synchronisez les threads voulant accéder à la pile partagée au moyen de sémaphores (complétez le pseudo code ci-dessous).

```
#include "includes.h"
#define N 100
int sommet = 0;
int pile [ N ];
/*0*/
void push (int a)
{
        /*1*/
        pile[ sommet ] = a;
        sommet++;
        /*2*/
}
int pop()
{
        /*3*/
        sommet --;
        int tmp = pile[ sommet ];
        /*4*/
        return tmp;
}
```

Le corrigé

Question 1 :

1) Oui, par l'appel système wait(&status) ou waitpid(pid, &status,0), un processus père peut récupérer dans status l'état de terminaison d'un processus fils. En appliquant des macros sur status, il peut déterminer s'il s'agit d'une fin normale, si oui, récupérer le code de terminaison (exit(c)), etc.
2) Oui, car la fonction exec remplace l'espace d'adressage (table des pages) du processus par un autre sans remplacer la table des descripteurs de fichiers.
3) Non, car le blocage d'un thread par P(S) sur un sémaphore va bloquer tous le processus.

Question 2 :

```
#include <unistd.h>
#include <stdlib.h>
#include<stdio.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <signal.h>
#define MaxJoueurs 4
static void action (int sig)
{      printf("ALARME declenchee\n");
      while(wait(NULL)>0);
       exit(0);
}
void jouer(int NJ)
{       sleep(3);
}
int main( )
{       int status,i, nb=0;
        for(i=0; i<MaxJoueurs; i++)
        {       if(fork()==0)
                {       jouer(i);
                        exit(i);
                }
        }
        signal(SIGALRM,action);
        alarm(60);
        while(nb<1000)
        {       if (wait(&status)>0)
                {       status = WEXITSTATUS(status);
                        if (fork()==0)
                        {       jouer(status);
                                exit(status);
                        }
                 nb++;
                }
        }
        return 0;
}
```

Question 3 :

```
pid_t DuplexPipe (int fd[2], char* com[])
{       int p1[2], p2[2],
        pid_t pid ;
        pipe(p1) ;
        pipe(p2) ;
        pid =fork() ;
        if (pid==0)
        {   dup2(p1[0],0) ;
            dup2(p2[1],1);
            close(p1[0]) ; close(p1[1]) ;close(p2[0]) ; close(p2[1]) ;
           execvp(com[1], com) ;
        }
        close(p1[0]) ; close(p2[1]) ;
        fd[0]=p2[0] ;
        fd[1]=p1[1] ;
        return pid ;
}
```

Question 4 :

1) a)

Semaphore mutex =1;		
A	B	C
P(mutex); a1; a2; V(mutex);	P(mutex); b1; b2; V(mutex);	P(mutex); c1; c2; c3; V(mutex);

b)

Semaphore S =2;		
A	B	C
P(S); a1; a2; V(S);	P(S); b1; b2; V(S);	P(S); c1; c2; c3; V(S);

c)

Semaphore mutex1 =1, mutex2=1;		
A	B	C
P(mutex1); a1; a2; V(mutex1);	P(mutex2); b1; b2; V(mutex2);	P(mutex1); P(mutex2); c1; c2; c3; V(mutex2); V(mutex1);

d)

Semaphore S =0;		
A	B	C
a1; a2; V(S); V(S);	P(S); b1; b2;	P(S); c1; c2; c3;

2) Il s'agit d'une variante du modèle de synchronisation « Producteurs – consommateurs » où le tampon est une pile.

```
#include "includes.h"
#define N 100
int sommet = 0;
int pile [ N ];

/*0*/ Semaphore libre = N, occupe=0, mutex=1;

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

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