Partie 1 : Processus et Thread

Exercice 1 :

1) Quel est le rôle d'un système d'exploitation ? Les interpréteurs de commandes et les compilateurs font-ils parties du système d'exploitation ?

2) Qu'est ce qu'un système multiprogrammé ? Un système de traitement par lots ? Un système en temps partagé ?

3) Dans le système UNIX, les véritables appels système sont effectués à partir :

- d'un programme utilisateur
- d'une commande shell
- d'une procédure de la bibliothèque standard

Sont-ils exécutés en mode superviseur ou en mode utilisateur ?

4) Comment sont organisés les fichiers dans le système UNIX ? Un utilisateur peut-il accéder à un fichier d'un autre utilisateur ? Si oui, comment ?

5) Dans le système UNIX, est-ce que tout processus a un père ? Que se passe-t-il lorsqu'un processus devient orphelin (mort de son père) ? Quand est-ce un processus passe à l'état Zombie ?

6) Pour lancer en parallèle plusieurs traitements d'une même application, vous avez le choix entre les appels système fork( ) et pthread_create( ). Laquelle des deux possibilités choisir ? pourquoi ?

7) Citez quatre événements qui provoquent l'interruption de l'exécution d'un processus en cours, dans le système UNIX.

8) Quel est le rôle de l'ordonnanceur ? Décrire brièvement l'ordonnanceur du système UNIX ? Favorise-t-il les processus interactifs ?

9) Pourquoi le partage de données pose des problèmes dans un système multiprogrammé en temps partagé ? Le système UNIX permet-il de contrôler les accès aux données partagées ? Qu'est-ce qu'une section critique ?

Exercice 2 :

Que fait chacun des programmes suivants :

1)

```
int main( )
{       int p=1 ;
        while(p>0) p=fork() ;
        execlp("prog", "prog", NULL) ;
        return 0 ;
}
```

2)

```
int i=2 ;
int main ( )
{       j=10;
        int p ;
        while(--i && p = fork())
        if(p<0) exit(1) ;
        j += 2;
        if (p == 0)
        {       i *= 3;
                j *= 3;
        }else
        {       i *= 2;
                j *= 2;
        }
        printf(« i=%d, j=%d », i,j) ;
        return 0 ;
}
```

3)

```
#include <stdio.h>
#include <unistd.h>
int main ( )
{
        int fd[2], i=2;
        char ch[100];
        while (i)
        {       pipe(fd);
                if( fork())
                {       close(fd[0]);
                        dup2(fd[1],1);
                        close(fd[1]);
                        break;
                } else
```

```
                    {       close(fd[1]);
                            dup2(fd[0],0);
                            close(fd[0]);
                    }
                    i--;
            }
            scanf("%s", ch);
            printf("%s\n",ch);
            exit(0);
    }
```

4)

```
    int i=4, j=10;
    int main ( )
    {       int p ;
            p = fork();
            if(p<0) exit(1) ;
            j += 2;
            if (p == 0)
            {       i *= 3;
                    j *= 3;
            }
            else
            {       i *= 2;
                    j *= 2;
            }
            printf("i=%d, j=%d", i,j) ;
            return 0 ;
    }
```

5)

```
    int main ( )
    {       int p=1 ;
            for(int i=0 ; i<=4 ; i++)
            if (p>0) p=fork( ) ;
            if(p !=-1) execlp("prog", "prog", NULL) ;
            else exit(1) ;
            while( wait(NULL) !=-1) ;
            return 0 ;
}
```

Exercice 3 :

1) Dans le cas d'UNIX, la création de processus est réalisée par duplication.

a) Citez un avantage et un inconvénient.

b) Citez en deux ou trois lignes les avantages des processus légers (threads) par rapport aux processus.

2) Considérez un fichier nommé COURS. Pour accélérer la recherche du mot INF3600 dans le fichier COURS, le processus de départ crée quatre processus. Chaque processus fils créé effectue la recherche dans une des quatre parties du fichier en appelant la fonction Recherche suivante :

bool Recherche (char * Fichier, char * Mot, int Partie) où :
- **Fichier** est le nom du fichier, c'est-à-dire COURS,
- **Mot** est le mot recherché, c'est-à-dire INF3600 et
- **Partie** est la partie 1, 2, 3 ou 4 du fichier.

Cette fonction retourne 1 en cas de succès et 0 sinon.
Au retour de la fonction Recherche, le processus fils transmet, en utilisant l'appel système **exit**, au processus père le résultat de la recherche (**exit(0**) en cas de succès, **exit(1)** en cas d'échec). Lorsque le processus père est informé du succès de l'un de ses fils, il tue tous les autres fils.

a) Ecrivez un programme C/C++ qui réalise le traitement ci-dessus.
Attention : n'écrivez pas le code de la fonction **Recherche**.

Exercice 4 :

1) Expliquez pourquoi, dans UNIX, lorsqu'un processus exécute l'appel système « exit », ses ressources ne sont pas libérées tout de suite.
2) Expliquez pourquoi les processeurs ont deux modes de fonctionnement (noyau et utilisateur).
3) Écrivez un programme qui lance, l'un à la suite de l'autre, les fichiers exécutables passés en paramètre. Un fichier exécutable est lancé uniquement si tous ceux qui le précèdent sont terminés. Avant de se terminer, le programme affiche, à l'écran, le nombre de lancements de fichiers exécutables qui ont échoués (parce que le fichier n'existe pas ou n'est pas exécutable).

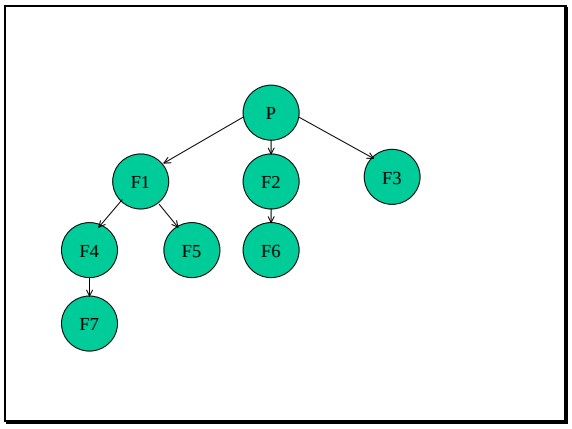
Exercice 5 :

Considérez le programme suivant:

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>
#include <stdlib.h>

int main ( )
{       pid_t pid;
        int i, n=3;
        for (i=0; i<n; i++)
        { /*1*/
        }
        while (wait(NULL) >= 0);
        return 0;
}
```

1. Complétez le code précédent de manière qu'il crée l'arborescence illustrée ci-dessus :



Les processus fils du dernier niveau se transforment en a (a est un fichier exécutable). Les autres processus fils affichent à l'écran le message « je continue » avant de poursuivre la création de processus.

2. Est-ce que le code complété engendre des processus zombies? Justifiez.

3. Tracez l'arborescence des processus créés par le programme si l'on supprime la ligne **while (wait(NULL) >= 0);** et à la fin du code des pères l'on ajoute les lignes :

wait(NULL);

exit(0);

Exercice 6 :

Considérez le programme C suivant :

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
void main()
{
        pid_t p1, p2, p3, p4;
        if ((p1=fork( ))==0)
                if ((p2=fork( ))==0)
                        f2( );
                else
                        f1( );
        else
                if ((p3=fork())==0)
                        f3( );
                else
                        if ((p4=fork())==0)
                                f4( );
        sleep(3);
        while(wait(NULL)>0) ;
}
```

1) Tracez l'arborescence des processus créés par ce programme si les fonctions f1, f2, f3 et f4 se terminent par exit().

2) Est-ce que ce programme peut produire des processus zombies? Justifiez.

Exercice 7 :

Considérez le programme suivant :

```
#include <unistd.h>
```

```
#include <stdlib.h>
#include<stdio.h>
#include <sys/wait.h>

void jouer(int NumJoueur); //NumJoueur est le numéro du joueur  0..MaxJoueurs-1

#define MaxJoueurs 4

/*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 joueur
         //lorsqu'un joueur se termine, un autre joueur de même numéro est créé

        /*3*/

      }

      return 0;
}
```

Complétez le programme en ajoutant le code qui réalise exactement les traitements spécifiés sous forme de commentaires.