# Partie 1 : Processus et Thread

## Le corrigé

## Solution 1

1) Il gère et contrôle le matériel et offre aux utilisateurs une machine virtuelle plus simple d'emploi que la machine réelle (appels systèmes). Non, les interpréteurs et les compilateurs ne font pas parties du système d'exploitation. 2) Un système multiprogrammé gère le partage des ressources (mémoire, processeur, périphériques…) de l'ordinateur entre plusieurs programmes chargés en mémoire. Dans un système de traitement par lots, les processus sont exécutés l'un à la suite de l'autre selon l'ordre d'arrivée. Dans un système en temps partagé, le processeur est alloué à chaque processus pendant au plus un quantum de temps. Au bout de ce quantum, le processeur est alloué à un autre processus.

3) A partir de la bibliothèque standard des appels système (instruction TRAP). Ils sont exécutés en mode superviseur (Leurs codes constituent le système d'exploitation).

4) Les fichiers sont organisés dans des répertoires. Chaque répertoire peut contenir des fichiers ou des répertoires (une structure arborescente). Pour contrôler les accès aux fichiers, chaque fichier a son propre code d'accès sur 9 bits. Un utilisateur peut accéder à un fichier d'un autre utilisateur si le code d'accès du fichier le permet. Le chemin d'accès est absolu.

5) Oui à l'exception du processus INIT. Le processus INIT devient son père. Un processus devient Zombie lorsqu'il effectue l'appel système exit et envoie donc un signal à son père puis se met en attente que le père ait reçu le signal.

6) tht_create ( ) car le fork( ) consomme beaucoup d'espace (duplication de processus). Mais il faut faire attention au conflit d'accès aux objets partagés.

7) 1) fin d'un quantum, 2) demande d'une E/S, 3) arrivée d'un signal, 4) mise en attente par l'opération sem_wait d'un sémaphore…

8) L'ordonnanceur gère l'allocation du processeur aux différents processus. L'ordonnanceur d'UNIX est un ordonnanceur à deux niveaux, à priorité qui ordonnance les processus de même priorité selon l'algorithme du tourniquet. L'ordonnanceur de bas niveau se charge de sélectionner un processus parmi ceux qui sont prêts et résidents en mémoire. Cette restriction permet d'éviter lors de commutation de contextes qu'il y ait un chargement à partir du disque

d’un processus en mémoire (réduction du temps de commutation). L’ordonnanceur de haut niveau se charge de ramener des processus prêts en mémoire en transférant éventuellement des processus sur disque (va-et-vient).

Oui, il favorise les processus interactifs car ces derniers font beaucoup d’E/S et à chaque fin d’E/S, ils se voient attribuer une priorité négative.

9) Un autre processus peut accéder aux données partagées avant qu’un processus n’est fini de les utiliser (modifier). Oui, par exemple les sémaphores. Une suite d’instructions qui accèdent à des objets partagés avec d’autres processus.

## Solution 2

1) Le père crée des processus fils tant qu’il n’y a pas d’échec. Le père et les processus créés se transforment en prog.

2) Le processus père tente de créer un fils et rentre dans la boucle. Si la création a échoue, le processus père se termine (p<0). Sinon il sort de la boucle car l’expression (--i) devient égale à 0. Il exécute ensuite j+=2 ; i*=2 ; j*=2 ; et enfin, il imprime les valeurs 0 et 24.

Le fils ne rentre dans la boucle car i=1 mais p=0. Il exécute ensuite j+=2 ; i*=3 ; j*=3 ; et enfin, il affiche les valeurs 3 et 36.

3) pour i=2, le père crée un pipe puis un fils. Il dirige sa sortie standard vers le pipe puis il se met en attente de données du clavier pour les déposer sur le pipe. Ensuite, il se termine.

Le fils dirige son entrée standard vers le pipe puis crée un autre pipe et son propre fils (i=1). Il dirige sa sortie standard vers le deuxième pipe créé puis se met en attente de lecture de données du premier pipe. Les données lues sont déposées sur le deuxième pipe.

Ensuite, il se termine.

Le petit fils dirige son entrée standard vers le pipe. Il se met en attente de lecture de données du second pipe. Il sort de la boucle car i devient nul. Les données lues sont affichées à l’écran. Enfin, il se termine.

4) Le père tente de créer un fils. S’il ne parvient pas il se termine. Sinon, il affiche i =8, j=24. Le fils affiche i=12, j=36

5) Le père tente de créer 5 fils. S'il parvient, il se transforme en prog. Sinon il se termine. Les fils créés se transforment en prog.

## Solution 3

1. a. Facilite la duplication des processus exécutant un même programme. Complique la création des processus exécutant des programmes différents.

1. b. Meilleur partage des ressources Gain en temps et en espace Meilleur réactivité

2. a.
```
int main ( )
{
        int pid[4], status, x;
        for (int i=0; i<4; i++)
        { // creation du (i+1) ième fils
                if ((pid[i] = fork() ) == 0)
                        if( Recherche( "COURS", "INF3600", i+1 )
                        exit(0);
                else exit(1);
        }
        while ((x=wait(&status))>0)
        if ( status>>8 ==0)
        {       for(i=0;i<4 ; i++)
                if(pid[i]!=x) kill (pid[i], SIGKILL);
        exit(0);
        };
        exit(1);
}
```

## Solution 4

1. Le processus passe à l'état zombie et reste dans cet état jusqu'à ce que le père exécute un « wait (&vp) » et récupère dans vp la valeur du paramètre de retour de exit ainsi que d'autres informations concernant la terminaison du fils par exemple fin normale ou anormale.

2. Pour protéger le système d'exploitation contre les intrusions et les erreurs des autres. Les instructions privilégiées du système d'exploitation s'exécutent en mode noyau. Le mode utilisateur permet d'isoler les processus utilisateur et de fournir une meilleure protection aux processus du noyau face à ces processus utilisateur.

3.

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

int main (int argc, char* argv[])
{       char * fich[argc-1];
        int status,i, j=0;
        for (i=1 ; i<=argc; i++)
        {       if ( fork() ==0)
        {       execlp (argv[i], argv[i], NULL);
                exit(3);
        }

        wait(&status);
```

```
        if( (status>>8)==3)
        {       fich[j] = new char[strlen(argv[i])+1];
                strcpy(fich[j],argv[i]);
                j++;
        }
}

        // Affichage des programmes non exécutés
        cout << "Nombre de programmes non exécutés :" << j << endl;

        if(j!=0)
        {       cout << "Liste de programmes non exécutés : \n";
                for (i=0; i<j; i++) cout << fich[i] << endl;
        }
        return 0;

}
```

## Solution 5

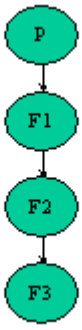
1.

```
#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;
       printf("processus racine %d\n", getpid());
```

```
for (i=0; i<n; i++)
{       /*1*/// printf("creation d'un processus par %d\n",  getpid());
        pid = fork();
        if(pid==0)
        {       printf("creation d'un processus %d par %d\n", getpid(),getppid());
                if( i == n-1)
                        execlp("a", "a", NULL);
                else
                        printf(" Je retourne %d\n", getpid());
        }else {         if(pid==-1)  exit(1);
                        printf(" Je suis un père %d\n", getpid());
                }
}
while (wait(NULL) >= 0);
return 0;
}
```

2. Oui. Si un processus fils termine avant son père, il reste dans l'état zombie jusqu'au moment où son père qui fait wait récupère son état de terminaison et le détruit.

3.




## Solution 6

1. Le processus qui exécute main crée 3 fils (p1, p3 et p4). Le processus p1 crée un fils p2.

2. Oui, les processus p1, p3 et p4 peuvent devenir des zombies pour une certaine période de temps, s'ils se terminent avant leur père (le processus qui exécute main). Le processus p2 est un bon candidat à zombie si son père p1 ne l'attend pas (ne fait pas de wait à la fin de la fonction f1) ou s'il se termine avant son père.

## Solution 7

```
#include <unistd.h>
#include <stdlib.h>
#include<stdio.h>
#include <sys/wait.h>
void jouer(int NumJoueur;

#define MaxJoueurs 4
int main( )
{       int status,i;
        // créer MaxJoueurs processus fils du processus principal
        // chaque joueur exécute la fonction jouer puis se termine
        for(i=0; i<MaxJoueurs; i++)       /* 2 pts */
        {       if(fork()==0)
                {
                        jouer(i);
                        exit(i);
                }
        }
        while(1)       /*2 pts */
        { // attendre la fin d'un joueur
          //lorsqu'un joueur se termine, un autre joueur de même numéro est créé
                if (wait(&status)>0)
                {       status = WEXITSTATUS(status);
                        if (fork()==0)
```

```
                {
                    jouer(status);
                    exit(status);
                }

            }

        }
        return 0;
}
```