Rappels et notions importantes sur les Processus

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

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

   - 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. 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 ?

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

   - 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. 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 ?

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

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

   - 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. 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 ?

   - 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

10. Dans le cas d'UNIX, la création de processus est réalisée par duplication. Citez un avantage et un inconvénient.

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

11. Expliquez pourquoi, dans UNIX, lorsqu'un processus exécute l'appel système « exit », ses ressources ne sont pas libérées tout de suite.

    - 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.

12. Expliquez pourquoi les processeurs ont deux modes de fonctionnement (noyau et utilisateur).

    - 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.

Exercice N°1: « Manipulation des processus dans le Terminal »

Pour voir quels processus tournent sur une machine à un moment donné, il faut utiliser la commande ps.
1) Ouvrir deux terminaux. Dans le premier terminal, lancer 2 applications, par exemple *firefox* et *gedit* à l'aide des commandes firefox & et xemacs &. Dans le deuxième terminal, tapez la commande ps.
Que se passe-t-il ? Pourquoi firefox et gedit n'apparaissent-ils pas dans la liste ?

ps permet d'obtenir la liste des processus qui tournent au moment où on lance la commande, c'est pour cette raison que les processus utilisateur bash (l'interpréteur de commandes du terminal) et ps (la commande qui vient d'être lancée) apparaissent. Par contre, les processus firefox et gedit n'apparaissent pas car ce ne sont pas des processus utilisateur mais des processus système.

Quelle option utiliser avec ps pour les voir ?

Si l'on souhaite lister tous les processus, il faut taper la commande ps -e ou ps -ef. La colonne UID (User ID) indique le nom de l'utilisateur qui a lancé la commande.

2) Utilisez la commande ps pour déterminer le PID (*Process ID*) du firefox que vous avez lancé. Tapez kill -9 lepiddefirefox.
Que se passe-t-il ?

La fenêtre du navigateur firefox se ferme.

Déterminez le PID d'une des commandes bash et arrêtez-la à l'aide de la commande kill -9. Pourquoi la fenêtre du terminal disparaît-elle ?

La fenêtre du terminal se ferme car elle correspond à l'interpréteur bash que nous venons de tuer.

3) Tapez firefox dans le premier terminal.
Pouvez-vous exécuter d'autres commandes dans ce terminal ? Pourquoi ? Faites un Ctrl-C.
Quel processus a été tué ?

Il n'est pas possible de lancer d'autres commandes car firefox a été lancé en avant-plan. L'appui CTRL-C ferme la fenêtre firefox en indiquant que le processus a été arrêté.

Exercice N°2: « La fonction fork()»

La fonction fork() permet de dupliquer un processus existant afin d'accéder à de la programmation parallèle. Il existe tout un tas d'autres techniques pour faire ceci, mais dans le cadre de ce cours nous ne nous intéresserons uniquement à la fonction fork et ses utilisations.

Exercice 2a:

Réaliser un programme qui utilise la fonction fork(). Vous stockerez le résultat de cette commande dans une variable et vous l'afficherez.

- Que constatez-vous ?
- Comment expliquer ce résultat ?
- À votre avis à quoi correspond l'affichage ?
- Est-il possible d'avoir deux morceaux de codes exécutés qu'une seule fois ?

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

int main(int argc, char**argv) {
      int retval;
      int status;
      int pchild = fork();
      if (-1 == pchild) {   // Erreur au fork
            fprintf(stderr, "ERREUR AU FORK\n");
            exit(-1);
      }
      if (0 == pchild) { // code du fils
            printf("CHILD : Enter an exit value : ");
            scanf("%d", &retval);
            sleep(1);
            exit(retval);
      }
      else {     // code du père
            printf("PARENT : JE SUIS TON PERE et tu as le PID %d\n", pchild);
            wait(&status);
            printf("PARENT : Child has terminated with exit code %d\n",
WEXITSTATUS(status));
            exit(0);
      }
}
```

Exercice 2b:

Dans un programme qui utilise la fonction fork, vous créerez des variables globales et locales de types primitifs et de types pointeurs. Vous les initialisez avant l'utilisation de fork, puis vous les affichez dans le code du père, et dans le code du fils. Pour les variables de types pointeurs, vous afficherez l'adresse du pointeur ainsi que sa valeur.

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

int g=66;
int *pg;

int main(int argc, char**argv) {

      // Variables de test
      int i=5;
      int *pi = (int *) malloc(sizeof(int));
      *pi=10;
      pg = (int *) malloc(sizeof(int));
      *pg=55;

      int pchild = fork();
      if (-1 == pchild) {   // Erreur au fork
            fprintf(stderr, "ERREUR AU FORK\n");
            exit(-1);
      }
      if (0 == pchild) { // code du fils
            printf("CHILD : g=%d, pg=%d, &pg=%x, i=%d, pi=%d, &pi=%x\n",
g,*pg,pg,i,*pi,pi);
            exit(0);
      }
      else {     // code du père
            printf("PARENT : g=%d, pg=%d, &pg=%x, i=%d, pi=%d, &pi=%x\n",
g,*pg,pg,i,*pi,pi);
            exit(0);
      }
}
```

Exercice 2c:

Dans un programme qui utilise la fonction fork, vous créerez plusieurs fils à partir du même père. Mettez le père en attente jusqu'à la terminaison de tous les fils (avec wait ou waitpid) et, ensuite, continuez l'exécution du père.

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

```
int main(int argc, char**argv) {
      int retval;
      int status;
      int pchild1 = fork();
      if (-1 == pchild1) {  // Erreur au fork
            fprintf(stderr, "ERREUR AU FORK\n");
            exit(-1);
      }
      if (0 == pchild1) { // code du fils
            sleep(1);
            exit(0);
      }
      else {      // code du père
            printf("PARENT : JE SUIS TON PERE et tu as le PID %d\n", pchild1);
            int pchild2 = fork();
            if (-1 == pchild2) {  // Erreur au fork
                  fprintf(stderr, "ERREUR AU FORK\n");
                  exit(-1);
            }
            if (0 == pchild2) { // code du fils
                  sleep(1);
                  exit(0);
            }
            else {      // code du père
                  printf("PARENT : JE SUIS TON PERE et tu as le PID %d\n",
pchild2);
                  int pchild3 = fork();
                  if (-1 == pchild3) {  // Erreur au fork
                        fprintf(stderr, "ERREUR AU FORK\n");
                        exit(-1);
                  }
                  if (0 == pchild3) { // code du fils
                        sleep(1);
                        exit(0);
                  }
                  else {      // code du père
                        printf("PARENT : JE SUIS TON PERE et tu as le PID %d\n",
pchild3);
                        waitpid(pchild1, NULL, 0);
                        waitpid(pchild2, NULL, 0);
                        waitpid(pchild3, NULL, 0);
                        printf("PARENT : All Childs have terminated\n");
                        exit(0);
                  }
            }
            exit(0);
      }
}
```

Exercice N°3: « La fonction execl() »

1) Donnez le code source C d'un programme affichez.c qui affiche à l'écran la chaîne de caractères qui lui est passée en paramètre en ligne de commande.
Exemple d'utilisation : affichez coucou
Compiler et tester ce programme.

```
#include <stdio.h>
int main(int argc, char * argv[]) {
if(argc != 2)
     printf("usage: affichez message\n");
else
     printf("%s\n", argv[1]);
return 0;
}
```

2) Ecrire un programme prog1 qui crée un processus fils qui exécute affichez avec l'argument salut. On utilisera la fonction execl.

```
#include <unistd.h> /* necessaire pour les fonctions exec */
#include <sys/types.h> /* necessaire pour la fonction fork */
#include <unistd.h> /* necessaire pour la fonction fork */
#include <stdio.h> /* necessaire pour la fonction perror */

int main() {
pid_t pid;
if ((pid = fork()) < 0)
     perror("fork error");
else if (pid == 0) {
     if (execl("/comptes/queudet-f/affichez","affichez","salut",(char *) 0) < 0)
     perror("execl error");
}
}
return 0;
}
```

Exercice N°4: « La fonction kill() »

Ecrire un programme qui crée un processus fils qui affiche à chaque seconde le nombre de secondes écoulées. Le processus père arrête le processus fils au bout de 10 secondes.

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
int main() {
int i=0;
int pidfils=fork();
if(pidfils!=0)
```

```
{
      sleep(10);
      kill(pidfils,SIGKILL);
} else
{
      while(1) {
      sleep(1); i++;
      printf("%d \n",i);
}
}
}
```

Exercice N°5: « La fonction wait() »

Ecrire un programme qui crée 2 processus, l'un faisant la commande ls -l, l'autre ps -l.
Le père devra attendre la fin de ses deux fils et afficher quel a été le premier processus à terminer.

```
#include <unistd.h> /* necessaire pour les fonctions exec */
#include <sys/types.h> /* necessaire pour la fonction fork */
#include <unistd.h> /* necessaire pour la fonction fork */
#include <stdio.h> /* necessaire pour la fonction perror */
int main(int argc, char * argv[]) {
pid_t pid1,pid2,pid_premier;
int status;

switch(pid1=fork()) {
      case -1:   perror("fork error");
                 break;
      case 0:    execlp("ls","ls",(char *) 0);
                 break;
      default:   switch(pid2=fork()) {
                       case -1:   perror("fork error");
                                  break;
                       case 0:    execlp("ps","ps",(char *) 0);
                                  break;
                       default:   break;
                 }
                 break;
      }
pid_premier = wait(&status);
wait(&status);
if (pid_premier==pid1)
{
      printf("Premier processus a finir : %d\n", pid1);
}
else
{
      printf("Premier processus a finir : %d\n", pid2);
}
return 0;
```

Exercice N°6: « Montrer ce que vous avez compris des exercices précédents »

Rappel :

Pour manipuler des fichiers, il existe deux principaux modes d'accès : bas niveau, haut niveau. Il existe une façon "pratique" de savoir si on utilise une manipulation bas niveau, ou haut niveau : les manipulations bas niveau manipulent des file descriptor (fd) tandis que les manipulations haut niveau manipulent des FILE *. Parmi les fonctions bas niveau on trouve open, read, write, . . .

Que fait chacun des programmes suivants :

- ***Programme 1***

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

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.

- ***Programme 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 ;
}
```

Le processus père tente de créer un fils et rentre dans la boucle. Si la création a échoué, 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.

- ***Programme 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);
}
```

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.

- ***Programme 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 ;
}
```

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

- ***Programme 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 ;
}
```

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.

- ***Programme 6***

Considérons que nous avons 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.

Où COURS est un fichier contenant 4 parties. Que fait le programme suivant ?:

```
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);
}
```

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.
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.

- **Programme 7**

```
#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) ;
}
```

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

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

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

  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.

- ***Programme 8***

```
#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

int main( )
     {
          // créer MaxJoueurs processus fils du processus principal
          // chaque joueur exécute la fonction jouer puis se termine

     while(1)
     {
          // attendre la fin d'un joueur
          //lorsqu'un joueur se termine, un autre joueur de même numéro est
     créé

     }
     return 0;
}
```

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

```
#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++)
                {     if(fork()==0)
                      {
                      jouer(i);
                      exit(i);
                      }
                }
        while(1)
        {       // 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;
}
```