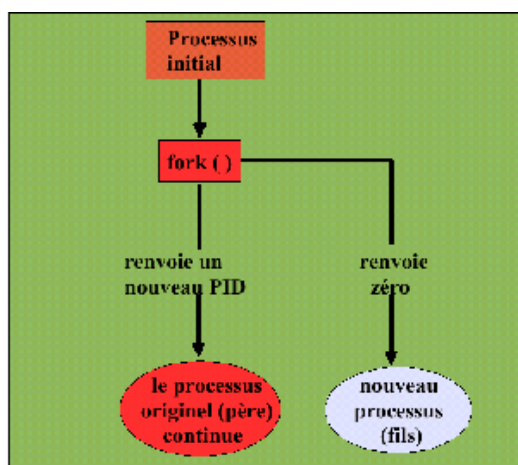


Création et synchronisation de processus. Communications par signaux, par tubes.

- Un processus est un ensemble d'octets (en langage machine) en cours d'exécution, en d'autres termes, c'est l'exécution d'un programme.
- Les processus sous Unix apportent :
 - La multiplicité des exécutions. Plusieurs processus peuvent être l'exécution d'un même programme.
 - La protection des exécutions. Un processus ne peut exécuter que ses instructions propres et ce de façon séquentielle ; il ne peut pas exécuter des instructions appartenant à un autre processus.
- Les processus sous Unix communiquent entre eux et avec le reste du monde grâce aux appels système.

La création d'un processus - fork()

Sous Unix, la création d'un processus est réalisée par l'appel système `fork()`. Cet appel système permet de créer dynamiquement (en cours d'exécution) un nouveau processus qui s'exécute de façon concurrente avec le processus qui l'a créé.



Le nouveau processus exécute le même code que le processus parent et démarre comme lui au retour du `fork()`. Le seul moyen de distinguer le processus fils du processus père est que la valeur de retour de la fonction `fork` est 0 dans le processus fils créé et est égale au numéro du processus fils nouvellement créé, dans le processus père et -1 en cas d'échec.

```
# include <unistd.h>
pid_t fork(void);
```

L'utilisation classique de la fonction `fork` est la suivante :

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

int pid;
pid = fork ( );
if (pid == - 1)
{ /* code si échec : printf ( " le fork ( ) a échoué\n " ) */
```


```

if (pid == 0)
  /* code correspondant à l'exécution du processus fils */
else
  /* code correspondant à l'exécution du processus père */

```

Exercice

Ecrire un programme "C" qui crée deux processus (un père et son fils). Chacun d'eux affichera à l'écran qui il est ("je suis le père de PID : ??", je suis le fils de PID : ?? et de père de PID : ??). Afin de connaître le PID (=numéro d'identification d'un processus) du processus en cours, la fonction *int getpid()* s'impose. Pour le processus père, c'est la commande *int getppid()*.

Rappel : pour compiler sous Unix, la commande à réaliser dans le terminal est : *cc -o proc.exe proc.c* . Où *proc.c* est le fichier où le code est écrit et *proc.exe* l'exécutable.

La fonction " fork () " dans une boucle

Pour mieux comprendre le fonctionnement de la fonction de création de processus (" fork () "), nous allons l'utiliser dans une boucle itérative.

Ecrire un programme en " C " qui fait appel à la fonction " fork () " dans une boucle " for (i=0; i<3; i++) ". A l'intérieur de chaque itération le programme affichera les informations suivantes :

```

(i : valeur_de_i) je suis le processus : pid, mon pere est : ppid, retour : retour
où
" valeur_de_i " est la valeur du compteur de la boucle
" pid " est le PID du processus courant,
" ppid " est le PID du processus père du processus courant,
" retour " est la valeur du code retour de l'appel à la fonction " fork ( ) ".

```

La fonction - execl ()

La fonction *execl()* illustre le mécanisme complémentaire à celui de la création de processus implanté via la fonction *fork()*, dont dispose UNIX. Le principe que nous allons voir ici revient à modifier le code du programme à exécuter pendant l'exécution du code en cours.

Ecrire un programme appelé " bonjour " dont le code affiche à l'écran la chaîne suivante : " bonjour ". Ecrire et lancer le programme suivant qui fait appel au programme " bonjour " via la méthode " execl () " :

```

#include <stdio.h>
#include <unistd.h>

int main ( ) {
  printf(" preambule du bonjour\n");
  execl("./bonjour",(char *) 0);
  printf(" postambule du bonjour\n");
}

```

Qu'en conclure ?

La synchronisation entre processus : fork(), wait(), waitpid(), system().

Les processus créés par des *fork()* s'exécutent de façon concurrente avec leur père. Ne pouvant présumer de la politique et de l'état de l'ordonnanceur du système, il sera impossible de déterminer quels processus se termineront avant tels autres (y compris leur père).

Dans certains cas le père meurt avant la terminaison de son fils ; à la terminaison du fils, on dira que ce dernier est dans un état zombi (dans l'affichage de la commande "ps", S=Z). Le noyau rattache ces processus au processus *init* (1). D'où l'existence, dans certains cas, d'un problème de synchronisation.

Primitives de synchronisation : *wait()*, *waitpid()*

La primitive *wait* permet l'élimination des processus zombis et la synchronisation d'un processus sur la terminaison de ses descendants avec récupération des informations relatives à cette terminaison. Elle provoque la suspension du processus appelant jusqu'à ce que l'un de ses processus fils se termine.

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait (int *pointeur_status)
```

Introduite dans la norme POSIX, la primitive *waitpid* permet de sélectionner parmi les fils du processus appelant un processus particulier.

Exercice

Les deux programmes ci-dessous proposent une première approche de synchronisation entre deux processus père et fils. Dans le second, il y a deux fils. Le père est synchronisé de telle sorte qu'il se termine après la terminaison du deuxième fils.

Ecrire ces deux programmes. Les exécuter et en analyser le fonctionnement.

```
// premier programme
#include <stdio.h>
#include <unistd.h>

main() {
    int pid;
    int status;

    switch(pid=fork()){
    case -1: perror("création de processus\n");
            exit(2);
    case 0: printf("Je suis le fils\n");
            printf("valeur de fork = %d\n", pid);
            printf("je suis le processus %d de père %d\n", getpid(),getppid());
            printf("fin du fils\n");
            exit(0);
    default :    wait(&status);
                printf("Je suis le père\n");
                printf("valeur de fork = %d\n", pid);
                printf("je suis le processus %d de père %d\n", getpid(),getppid());
                printf("fin du père\n");
                exit(128);
    }}

// deuxième programme
```

```

#include <stdio.h>
#include <unistd.h>

main() {
int pid_fils1, pid_fils2, pid_courant;
int status;

pid_fils1=fork();
pid_courant=getpid();
switch(pid_fils1){
case -1:
perror("Pb dans la création du 1er fils\n");
exit(2);
case 0:
printf("%d: je suis le 1er fils du processus %d\n", pid_courant,getppid());
printf("%d: fin du processus\n",pid_courant);
exit(0);
default :
switch(pid_fils2=fork()){
case -1:
perror("Pb dans la création du 2ème fils\n");
exit(3);
case 0:
printf("%d: Je suis le 2nd fils du processus %d\n",getpid(),getppid());
sleep(2);
printf("%d: fin du processus\n",getpid());
exit(1);
default :
printf("%d: je suis le père et j'attends mon fils de pid %d\n",pid_courant,pid_fils2);
waitpid(pid_fils2,&status,0);
printf("%d: fin du père\n",pid_courant);
exit(128);
}}}

```

Exercice

Regarder dans le manuel la commande `system()`. Ecrire un programme qui débute par l'affichage de BONJOUR, se poursuit par l'exécution de la commande "ls -l" et se termine par l'affichage de BONSOIR.

Si "`man system`" ne fonctionne pas :

`system` - appeler une commande shell

```

#include <stdio.h>

int system(commande)
char *commande ;

```

La fonction `system` appelle le shell, et lui passe la commande `commande` comme si elle avait été tapée au terminal. L'exécution est suspendue jusqu'à ce que la commande soit finie. Si `system` ne peut lancer la commande, une valeur négative est renvoyée.

La synchronisation entre processus et le traitement des signaux sous Unix.

Un signal est "**un message très court**" qu'un processus peut envoyer à un autre processus, pour lui dire qu'un **événement particulier est arrivé** (Intuitivement, ils sont comparables à des sonneries, les différentes sonneries indiquant des événements particuliers). Le processus pourra alors mettre en œuvre une réponse décidée et pré-définie à l'avance (*handler*). Ce mécanisme est implanté par un **moniteur**, qui scrute en permanence l'occurrence des signaux. C'est par ce mécanisme que le système communique avec les processus utilisateurs en cas d'erreur (violation mémoire, erreur d'E/S) ou à la demande de l'utilisateur lui-même (caractères d'interruption ^D, ^C ...).

En fait il n'existe qu'une vingtaine de signaux numérotés à partir de 1. Ces signaux portent également des noms (cf. liste des principaux signaux ci-dessous). On les trouve dans "/usr/include/signal.h".

Envoyer un signal revient à envoyer ce numéro à un processus. Tout processus a la possibilité d'émettre à destination d'un autre processus un signal à condition que ses numéros de propriétaires (UID) lui en donnent le droit vis-à-vis de ceux du processus récepteur.

Les principaux signaux :

SIGHUP (1) il est envoyé lorsque la connexion physique de la ligne est interrompue ou en cas de terminaison du processus leader de la session ;

SIGINT (2) frappe du caractère **intr** (interruption clavier) sur le clavier du terminal de contrôle ;

SIGQUIT (3) frappe du caractère **quit** (interruption clavier avec sauvegarde de l'image mémoire dans le fichier de nom **core**) sur le clavier du terminal de contrôle ;

SIGKILL (9) signal de terminaison non déroutable.

SIGTERM (15) signal de terminaison, il est envoyé à tous les processus actifs par le programme **shutdown**, qui permet d'arrêter proprement un système UNIX. Terminaison normale.

SIGILL (4) Instruction illégale.

SIGFPE (8) Erreur arithmétique.

SIGUSR1 (10) Signal 1 défini par l'utilisateur.

SIGSEGV (11) Adressage mémoire invalide.

SIGUSR2 (12) Signal 2 défini par l'utilisateur.

SIGPIPE (13) Écriture sur un tube sans lecteur.

SIGALRM (14) Alarme.

SIGCHLD (17) Terminaison d'un fils.

SIGCONT (18) Reprise du processus.

SIGSTOP (19) Suspension du processus (non déroutable).

SIGTSTP (20) Émission vers le terminal du caractère de suspension.

SIGTTIN (21) Lecture du terminal pour un processus d'arrière-plan.

SIGTTOU (22) Écriture vers le terminal pour un processus d'arrière-plan.

Lorsqu'un processus reçoit un signal, il exécute une **routine spéciale**. Cette routine peut être une routine standard du système ou bien une routine spécifique fournie par l'utilisateur. Il est également possible d'ignorer la réception d'un signal, c'est-à-dire de n'exécuter aucun traitement.

L'Envoi de signaux

Pour envoyer un signal à un processus, on utilise la commande appelée **kill**. Celle-ci prend en option (c'est-à-dire précédée du caractère '-') le numéro du signal à envoyer et en argument le numéro du (ou des) processus destinataire(s).

```
$ kill -TERM pid_process
```

La commande **kill** utilisée avec l'argument '-l' renvoie la liste des signaux du système :

```
$ kill -l
HUP INT QUIT ILL TRAP IOT EMT
FPE KILL BUS ... TTOU
$
```

En langage C, une fonction permet aussi de réaliser ceci :

```
int kill (pid_t pid, int sig);
```

Un tel appel à cette primitive a pour effet d'envoyer le signal de numéro **sig** au(x) processus déduit(s) de la valeur de **pid**. Les processus destinataires sont les suivants en fonction de la valeur de **pid** :

- <-1 tous les processus du groupe **|pid|**
- 1 tous les processus du système (sauf 0 et 1)
- 0 tous les processus dans le même groupe que le processus
- > 0 processus d'identité **pid**.

Voici un exemple d'utilisation de *handler* définis par l'utilisateur :

```
#include <stdio.h>
#include <signal.h>

/* handler utilisateur "bonjour" et "bonsoir" */
void bonjour (int sig) {
    printf ("bonjour à tous\n");
}
void bonsoir (int sig) {
    printf ("bonsoir à tous\n");
    signal (SIGUSR2, bonsoir);
}

int i;
main () {
    /* "armement" des signaux */
    signal (SIGUSR1, bonjour);
    signal (SIGUSR2, bonsoir);
    for (;;)
    }
```

Sur l'exemple précédent, on pourra envoyer dans le shell, les signaux correspondants aux processus à l'aide de la commande kill :

```
$ kill -USR1 pid
$ kill -USR2 pid
```

Exercice

Ecrire le programme précédent, et l'exécuter.

Au retour du *handler* appelé lors de la délivrance d'un signal capté, l'exécution du processus reprend, sauf demande contraire, au point où le processus a été interrompu. Il existe d'autres primitives permettant la synchronisation des processus. C'est le cas des primitives *sleep* et *pause*.

- La primitive *sleep* (*int durée*) permet de bloquer (état endormi) le processus courant sur la durée passée en paramètre.

- La primitive *pause* permet de mettre le processus courant en attente de l'arrivée de signaux.

```
#include <unistd.h>
int pause (void);
```

Exercice

Ecrire un programme dans lequel un processus crée un fils et initialise un handler (afficher BONJOUR) sur SIGUSR1. Le fils affiche des informations à l'écran puis envoie le signal SIGUSR1 à son père. Attention le programme fils doit se terminer avant le processus père. La sortie (les messages sur l'écran) du programme devra ressembler à ceci :

```
père : Je suis le processus 27406
fils : Je suis le processus 27407
père : j'attends un signal
BONJOUR
fils : fin du processus
père : fin du processus
```

La communication par tubes.

Les tubes sont un mécanisme de communication qui permet de réaliser des communications entre processus sous forme d'un flot continu d'octets. Dans ce TP, nous n'aborderons que la notion de tubes ordinaires (pipes).

Un tube est matérialisé par deux entrées de la table des ouvertures de fichiers, une de ces entrées est ouverte en écriture (l'entrée du tube), l'autre en lecture (la sortie du tube).



Création de tubes ordinaires

Un processus ne peut utiliser que les tubes qu'il a créés lui-même par la primitive pipe ou qu'il a hérités de son père grâce à l'héritage des descripteurs à travers fork() et exec().

```
#include <unistd.h>
int pipe (int p[2]);
```

p[0] correspond au descripteur en mode lecture
p[1] correspond au descripteur en mode écriture

Lecture dans un tube

On utilise l'appel système *read*.

```
int nb_lu; /* nb de caractères lus */
nb_lu = read(p[0], buffer, TAILLE_READ);
```

Remarquer que la lecture se fait dans le descripteur p[0].

Écriture dans un tube

On utilise l'appel système *write*.

```
int nb_écrit; /* nb de caractères écrits */
nb_écrit = write(p[1], buf, n);
```

Exemple d'utilisation des tubes ordinaires (fichier "prog.c") :

```
#include <stdio.h>
#include <unistd.h>
int tube[2];
char buf[20];
main() {
    pipe(tube);
    if (fork()) { /* pere */
        close (tube[0]);
        write (tube[1], "bonjour", 8);
    } else { /* fils */
        close (tube[1]);
        read (tube[0], buf, 8);
        printf ("%s bien reçu \n", buf);
    }
}
```

L'exécution de ce programme ("prog") donne le résultat suivant :

```
$ prog
$ bonjour bien reçu
$
```

Exercice

Écrire le programme précédent, et l'exécuter.

Exercice(s)

Analyser les programmes donnés dans la suite, de sorte à comprendre leur fonctionnement. Si le temps est disponible, les écrire et les exécuter.

```
/*
* premier programme
*/

#include <stdio.h>
extern printf();
extern pipe();
extern write();
extern read();
extern close();

int main(void)
{
    int tab_numero_fd[2];
    int xpipe;
    int valeur=12;
```



```

int valeur_lue;

if ((xpipe = pipe(tab_numero_fd)) == -1)
{ printf("pipe: error\n"); exit(1); }

write(tab_numero_fd[1], &valeur, sizeof(int));
printf("Apres write(1)\n");
valeur++;
write(tab_numero_fd[1], &valeur, sizeof(int));
printf("Apres write(2)\n");
valeur++;
read(tab_numero_fd[0], &valeur_lue, sizeof(int));
printf("Apres read(1)\n");
printf("valeur_lue= %d\n", valeur_lue);
write(tab_numero_fd[1], &valeur, sizeof(int));
printf("Apres write(3)\n");
valeur++;
read(tab_numero_fd[0], &valeur_lue, sizeof(int));
printf("Apres read(2)\n");
printf("valeur_lue= %d\n", valeur_lue);
write(tab_numero_fd[1], &valeur, sizeof(int));
printf("Apres write(4)\n"); valeur++;
write(tab_numero_fd[1], &valeur, sizeof(int));
printf("Apres write(5)\n");
read(tab_numero_fd[0], &valeur_lue, sizeof(int));
printf("Apres read(3)\n");
printf("valeur_lue= %d\n", valeur_lue);
read(tab_numero_fd[0], &valeur_lue, sizeof(int));
printf("Apres read(4)\n");
printf("valeur_lue= %d\n", valeur_lue);
read(tab_numero_fd[0], &valeur_lue, sizeof(int));
printf("Apres read(5)\n");
printf("valeur_lue= %d\n", valeur_lue);

close(tab_numero_fd[0]);
close( tab_numero_fd[ 1]);

return(0);
}

/*
* deuxième programme
*/

#define N 1
#define TRUE 1
#define FALSE 0
#define bits0a7 255
#include <stdio.h>
#include <signal.h>
#include <sys/wait.h>
extern printf();
extern fork();
extern getpid();
extern getppid();
extern pause();
extern sleep();
extern usleep();
extern pipe();
extern write();
extern read();
extern close();
int numero;

```

```

int tabPID[N+1];
int pere_fils_fd[2];
int FILS = TRUE;

void handler_SIGUSR2(int sig)
{
    if (numero == 0) FILS = FALSE;
}

void handler_SIGINT(int sig)
{
    if (numero != 0)
        { kill(getppid(), SIGUSR2); exit(16 + numero);
        }
}

int main(void)
{
    int xfork, xwait, xpipe;
    int status;

    signal(SIGUSR2, handler_SIGUSR2);
    signal(SIGINT, handler_SIGINT);

    printf("PERE: %d, GRAND-PERE: %d\n", getpid(), getppid());
    if ((xpipe = pipe(pere_fils_fd)) == -1)
        { printf("pipe: error\n"); exit(1); }

    for (numero=1; numero<=N; numero++)
        {
            if ((xfork = fork()) == -1)
                { printf("fork: error\n"); exit(2); }
            if (xfork == 0) break;
            else {
                tabPID[numero] = xfork;
                printf("PERE: fils: %d, PID: %d\n", numero, xfork); }
        }

    if (xfork == 0)
    {
        int j;
        int valeur_lue;
        int somme = 0;
        while (TRUE)
        {
            for (j=1; j<=5; j++)
                {
                    read(pere_fils_fd[0], &valeur_lue, sizeof(int));
                    printf("FILS: valeur_lue\n");
                    somme = somme + valeur_lue;
                }
            printf("FILS: somme= %d\n", somme); somme = 0;
        }
    }
    else
    {
        int valeur = 1;
        numero = 0;
        while (FILS)
            {
                write(pere_fils_fd[1], &valeur, sizeof(int));
                valeur++;
                usleep(300000);
            }
    }
}

```

```

    while ((xwait = wait(&status)) != -1)
    {
        printf("PERE: fils %d termine', y= %X, x= %X\n", xwait, status &
        bits0a7, (status>> 8) & bits0a7);
    }

    close(pere_fils_fd[0]);
    close(pere_fils_fd[1]);
    return 0;
}

/*
* troisième programme
*/

#define N 1
#define TRUE 1
#define FALSE 0
#define bits0a7 255
#include <stdio.h>
#include <signal.h>
#include <sys/wait.h>
extern printf();
extern fork();
extern getpid();
extern getppid();
extern pause();
extern sleep();
extern usleep();
extern pipe();
extern write();
extern read();
extern close();
int numero;

int tabPID[N+1]; int pere_fils_fd[2]; int fils_pere_fd[2];
int FILS = TRUE;

void handler_SIGUSR2(int sig)
{ if (numero == 0) FILS = FALSE;
}
void handler_SIGINT(int sig)
{ if (numero != 0) { kill(getppid(), SIGUSR2); exit(16 + numero); } }

int main(void)
( int xfork, xwait, xpipe; int status;

signal(SIGUSR2, handler_SIGUSR2); signal(SIGINT, handler_SIGINT);

printf("PERE: %d, GRAND-PERE: %d\n", getpid(), getppid());
if ((xpipe = pipe(pere_fils_fd)) = -1)
    { printf("pipe: error\n"); exit(1); }
if ((xpipe = pipe(fils_pere_fd)) == -1)
    { printf("pipe: error\n\n"); exit(1); }

for (numero=1; numero<=N; numero++)
{ if ((xfork = fork()) == -1)
    { printf("fork: error\n"); exit(2); }
  if (xfork == 0) break;
  else
    { tabPID[numero] = xfork;

```

```
        printf("PERE: fils: %d, PID: %d\n", numero, xfork); }
    }
if (xfork == 0)
{ int j;
  int valeur_lue;
  int somme = 0;
  while (TRUE)
  { for (j=1; j<=5; j++)
    { read(pere_fils_fd[0], &valeur_lue, sizeof(int));
      printf("FILS: valeur_lue\n");
      somme = somme + valeur_lue;
    }
  write(fils_pere_fd[1], &somme, sizeof(int)); somme = 0;
}
}
else
{ int i;
  int valeur = 1;
  int somme_lue;
  numero = 0;
  while (FILS)
  { for (i=1; i<=5; i++)
    { write(pere_fils_fd[1], &valeur, sizeof(int));
      valeur++; }
    read(fils_pere_fd[0], &somme_lue, sizeof(int));
    printf("PERE: somme= %d\n", somme_lue);
    usleep(300000); }
  while ((xwait = wait(&status)) != -1)
  printf("PERE: fils %d termine', y= %X, x= %X\n", xwait,
        status & bits0a7, (status>> 8) & bits0a7);
  close(pere_fils_fd[0]); close(pere_fils_fd[1]);
  close(fils_pere_fd[0]); close(fils_pere_fd[1]);
  return 0;
}}
```