I. PROCESSUS

1. Getpid()

Chaque processus s'exécutant sur un système est repéré par un numéro d'identification de type entier. On l'appelle communément *pid* pour processus identification. Ce numéro nous sera nécessaire pour réaliser des communications entre les processus.

La fonction *getpid()* permet de retourner le *pid* du processus en cours.

Soit le code suivant :

```
int main()
{
        int pid ;
        pid=getpid() ;
        printf("pid du processus=%d\n",pid);
}
```

Ce programme affiche le pid du processus. Pour plusieurs exécution du ce même programme le résultat est différent.

2. Fork() : Création de Processus

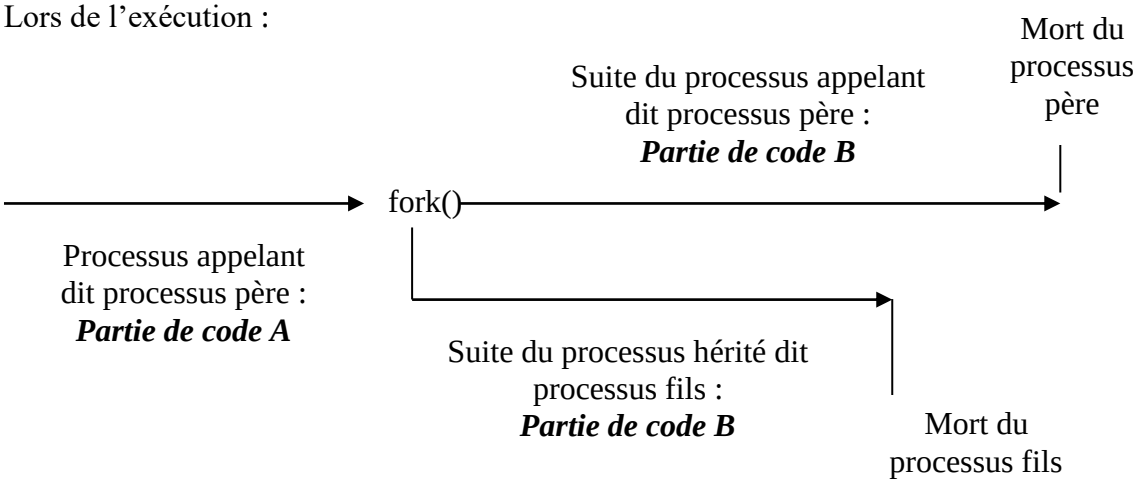
Cette primitive est un appel système qui permet de créer un processus fils hérité du processus qui l'appelle. A partir de ce moment là, l'exécution de la suite code se « dédouble ». En effet, tout le code qui suit le *fork*, s'exécute aussi bien dans le processus fils que dans le processus père.

Schéma chronologique n°1:

Soit le code suivant :

```
int main()
{
      /*  partie de code A */
      fork() ;
      /* partie de code B */
}
```

Lors de l'exécution :



Remarque : Les deux processus se terminent lors de la fin de l'exécution de leur code respectif. Rien ne garantit que l'un ou l'autre se terminera avant. Tout dépend de l'ordonnancement des processus du système.

VALEURS DE RETOUR

Le code s'exécutant deux fois à partir de l'appel système fork(), il est donc normal d'envisager que cette fonction retourne une valeur pour chacun des deux processus. Il y a un résultat pour le processus père, et un résultat pour le processus fils.

Par définition, elle retourne :

-1 si l'exécution a échoué (il faudra vérifier systématiquement le bon déroulement de la commande).

0 pour le processus fils, si la commande a réussi.

le **pid du fils** pour le processus père, si la commande à réussi.

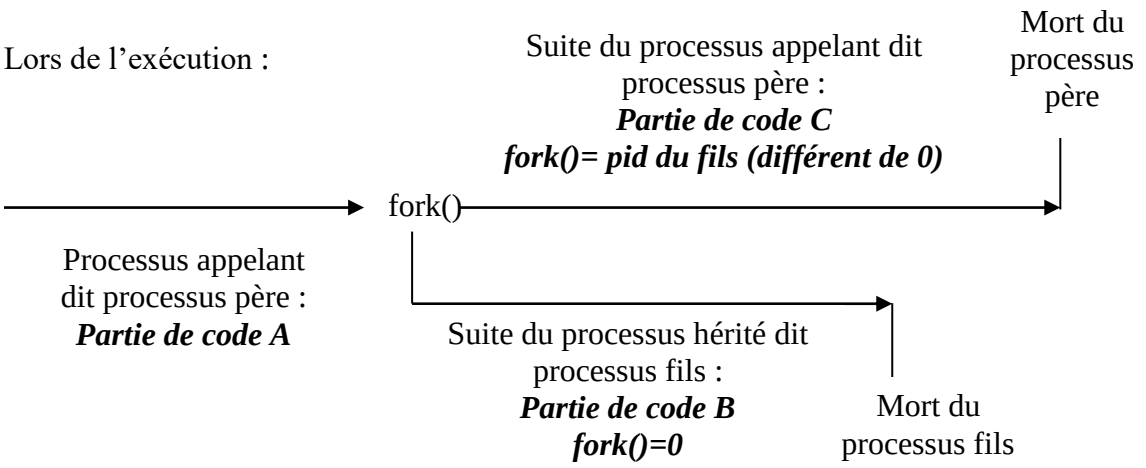
On se sert donc de cette propriété pour différencier les codes pour les parties père et fils.

Schéma chronologique n°1:

Soit le code suivant :

```
int main()
{
      /*  partie de code A */
      if (fork() ==0)
            {
            /* partie de code B */
            }
      else
            {
            /* partie de code C */
            }
}
```

Lors de l'exécution :



3. VARIABLES

L'appel système *fork* provoque non seulement la naissance d'un nouveau processus, mais aussi la duplication des zones mémoires. En effet, chaque processus a son espace de mémoire propre. A la création du nouveau processus, les variables utilisées dans le processus père restent inchangées et continuent leur évolution. Le processus fils bénéficie des mêmes variables et des mêmes valeurs mais dans des zones différentes (similaires aux variables locales lors d'un appel de fonction). Celles évolueront différemment de celles du processus père. Ce sont des variables distinctes. Elles ne sont pas partagées entre les différents processus.

Soit le code suivant :

```
int main()
{
      /*  partie de commune */
      int n ;
      n=0 ;
      if (fork() ==0)
            {
            /* processus fils*/
            n=n+2 ;
            printf("n=%d\n" , n) ;
            }
      else
            {
            /* processus père */
            n=n+1 ;
            printf("n=%d\n" , n) ;
            }
}
```

Ce programme affichera soit

2 puis 1, si le processus fils s'exécute en premier.

1 puis 2, si le processus père s'exécute en premier.

A aucun moment, il n'affichera 3, car n vaut 0 dans les deux processus après le fork(). Puis les additions portent sur des variables différentes. Le +2 et le +1 portent sur des zones mémoires différentes.

4. GETPPID()

La fonction *getppid()* retourne le pid du processus père d'un processus donné (du processus qui fait l'appel).

5. WAIT()

Une première synchronisation est réalisable par l'intermédiaire de l'appel système *wait()*. Celle-ci a pour effet de bloquer l'exécution du processus en attendant le mort d'un de ses processus fils.

Soit le code suivant :

```
int main()
{
      /*  partie de commune */
      int n ;
      n=0 ;
      if (fork() ==0)
            {
            /* processus fils*/
            printf("n=%d\n" , n+2) ;
            }
      else
            {
            /* processus père */
            wait() ;
            printf("n=%d\n" , n+1) ;
            }
}
```

Ce programme affichera forcément 2 puis 1 car la fonction *wait()* placée au début du processus père force ce processus à attendre la fin du processus fils. Le +2 s'exécute alors en premier. Une fois l'affichage de 2 réalisé, le fils meurt. Il provoque alors le déblocage du processus père. Le +1 peut alors se réaliser et l'affichage de 1 aussi.

6. WAITPID()

L'appel système *wait()* marche forcément du père vers le fils. Il ne faut pas aussi qu'il y ait des ambiguïtés sur les processus qui meurent. En effet, si un processus génère deux fils (deux appels *fork()*). Le *wait()* attendra la mort d'un des fils. Il est impossible de savoir lequel des deux fils a débloqué le père.

waitpid(pid_t** pid**, int ***status**, int** options**);

La fonction *waitpid()* suspend l'exécution du processus courant jusqu'à ce que le processus fils numéro *pid* se termine, ou jusqu'à ce qu'un signal à intercepter arrive. Si le fils mentionné par *pid* s'est déjà terminé au moment de l'appel (il est devenu "zombie"), la fonction revient immédiatement. Toutes les ressources utilisées par le fils sont libérées.
La valeur de *pid* peut également être l'une des suivantes :

- < -1 attendre la fin de n'importe quel processus fils appartenant à un groupe de processus d'ID pid.
- -1 attendre la fin de n'importe quel fils. C'est le même comportement que wait.
- 0 attendre la fin de n'importe quel processus fils du même groupe que l'appelant.
- > 0 attendre la fin du processus numéro pid.

La valeur de l'argument options est un OU binaire entre les constantes suivantes :
WNOHANG : ne pas bloquer si aucun fils ne s'est terminé.
WUNTRACED : recevoir l'information concernant également les fils bloqués si on ne l'a pas encore reçue.
Si status est non NULL, *wait* et *waitpid* y stockent l'information sur la terminaison du fils.

En cas de réussite, le PID du fils qui s'est terminé est renvoyé, en cas d'échec -1 est renvoyé et *errno* contient le code d'erreur.