Les commandes waitpid() et exec()

Synchronisation père/fils:

- ✓ En se terminant avec la fonction *exit* ou *return* dans main, un processus affecte une valeur à son *code de retour* :
  - processus père peut accéder à cette valeur en utilisant les fonctions *wait* et *waitpid*.

Wait ()

- Bloquer le processus père jusqu'à ce que l'un de ses fils termine.
- Le processus père attend n'importe quel fils.

Wait() renvoie soit :

-le PID du processus fils terminé
- En cas d'erreur -1

Le paramètre status correspond au code de retour du processus fils qui va se terminer

Les appels système wait(), waitpid() et exit()

- Dans le cas de plusieurs fils, comment on procède si le père veut attendre la fin d'un processus fils particulier ?

#include <sys/types.h>
#include <sys/wait.h>

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

Les appels système wait(), waitpid() et exit()

La primitive **waitpid()** permet au processus père d'attendre un **fils particulier**

waitpid() renvoie :

Le PID du fils en cas de réussite :

- si PID > 0 il s'agit d'un processus fils particulier
- si PID =-1 il s'agit de n'importe quel fils

-1 en cas d'échec

Interprétation de la valeur de retour - *int*status*

- ✓ Utilisation des ***macros*** pour des questions de portabilité :
  - Type de terminaison
    - WIFEXITED : non NULL si le processus fils s'est terminé normalement.
    - WIFSIGNALED : non NULL si le processus fils s'est terminé à cause d'un signal
    - WIFSTOPPED : non NULL si le processus fils est stoppé (option WUNTRACED de waitpid)
  - Information sur la valeur de retour ou sur le signal
    - WEXITSTATUS : code de retour si le processus s'est terminé normalement
    - WTERMSIG : numéro du signal ayant terminé le processus
    - WSTOPSIG : numéro du signal ayant stoppé le processus

Les appels système wait(), waitpid() et exit()

Exemple4.c : Un programme C qui permet à un processus père de **récupérer le PID et le code de retour** renvoyé par son fils dans la fonction exit.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
int main()
{
  pid_t pid = fork();
  if(pid == 0) {
    printf("Je suis le fils. PID=%d.\n", getpid());
    sleep(1);
    printf("Je retourne la valeur 4.\n") ;
    return 4;
  } else if (pid>0) {
    printf("Je suis le pere. PID=%d.\n", getpid());
    int status;
    pid_t pid_retour = wait(&status); //récupérer le PID du fils et son code de retour
    printf("Mon fils %d se termine avec le code de retour %d\n", pid_retour, WEXITSTATUS(status));

  } else {
    printf("probleme de creation de Fork\n");
  }
  return EXIT_SUCCESS;
}
```

On récupère la valeur 4 dans status

Les appels système wait(), waitpid() et exit()

Exemple6.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
int main()
{
  int x1,x2 ;
  printf("Je suis le pere, je vais enfanter.\n");
  pid_t pid1, pid2 ;
  pid1 = fork();
  if(pid1 == 0){
       sleep(1);
       printf("Je suis le fils1. PID=%d, PPID=%d\n", getpid(), getppid());
       return 3;
   }
    else if (pid1>0) {
        printf("Je suis le pere du fils1\n", getpid(), getppid());
        pid2 = fork();
        if(pid2==0) {
              sleep(2);
              printf("Je suis le fils2 . PID=%d, PPID=%d\n", getpid(), getppid());
              sleep(1);
              return 5;
        }
        else if(pid2>0) {
              printf("Je suis le pere du fils2\n", getpid(), getppid());
              waitpid(pid2,&x2,0);
              printf("Mon fils2 %d se termine avec le code de retour= %d\n", pid2, WEXITSTATUS(x2));
              waitpid(pid1,&x1,0);
              printf("Mon fils1 %d se termine avec le code de retour= %d\n", pid1, WEXITSTATUS(x1));

        }else{
              printf("probleme de creation de Fork\n");
               }
    }else {
         printf("probleme de creation de Fork\n");
           }
    return EXIT_SUCCESS;
}
```

Résultats

```
Je suis le pere, je vais enfanter.
//creation de fils1
//endormir fils1 1s
Je suis le pere du fils1
//creation du fils2
//endormir fils2 2s
Je suis le pere du fils2
//pere se bloque sur la fin de fils
Je suis le fils1. PID=7366, PPID=73
//fils1 retourne 3
Je suis le fils2. PID=7367, PPID=73
//fils2 attend 1s
//fils2 retourne 5
Mon fils2 7367 se termine avec
code de retour= 5
Mon fils1 7366 se termine avec
code de retour= 3
```

```c
// Création d'un processus fils
pid = fork();

if (pid == -1) {
    printf("Erreur lors de la création du processus fils\n");
    exit(EXIT_FAILURE);

} else if (pid == 0) {
    // Code exécuté par le processus fils
    printf("Je suis le processus fils (PID = %d)\n", getpid());
    printf("Je m'endors pendant 5 secondes...\n");
    sleep(5);
    printf("Je suis réveillé !\n");
    exit(EXIT_SUCCESS);

} else {
    // Code exécuté par le processus parent
    printf("Je suis le processus parent (PID = %d)\n", getpid());
    printf("Le processus fils a été créé avec le PID %d\n", pid);
    printf("J'attends la fin du processus fils...\n");
    wait(&status);
    if (WIFEXITED(status)) {
        printf("Le processus fils s'est terminé normalement avec le code de sortie %d\n", WEXITSTATUS(status));
    } else if (WIFSIGNALED(status)) {
        printf("Le processus fils a été tué par le signal %d\n", WTERMSIG(status));
    }
    printf("Fin du processus parent\n");
    exit(EXIT_SUCCESS);
```

status contient des informations sur la fin du processus fils

Les macros WIFEXITED() et WEXITSTATUS() pour vérifier si le processus fils s'est terminé normalement et récupérer son code de sortie

Les macros WIFSIGNALED() et WTERMSIG() pour vérifier si le processus fils a été tué par un signal et récupérer le numéro de ce signal.

Les appels système wait(), waitpid() et exit()

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

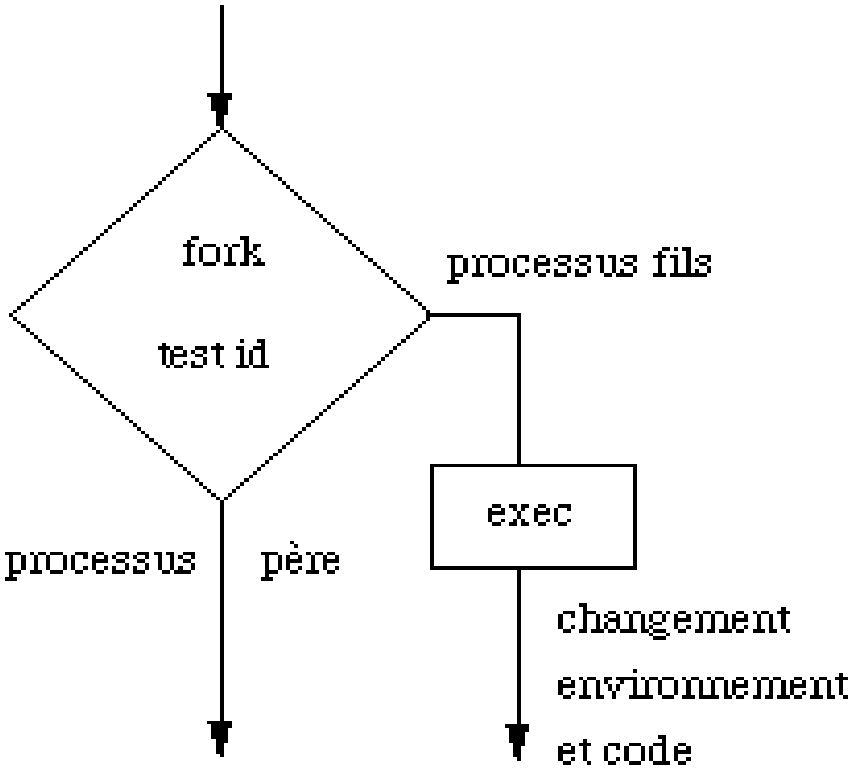
- Option 0 : waitpid() va bloquer le processus appelant jusqu'à ce que le processus enfant spécifié par le PID ait terminé son exécution. Pendant ce temps, le processus appelant est mis en pause et ne reprendra son exécution que lorsque waitpid() aura renvoyé.

- **wait(&status) est équivalent à waitpid(-1,&status,0)**

La primitive exec()

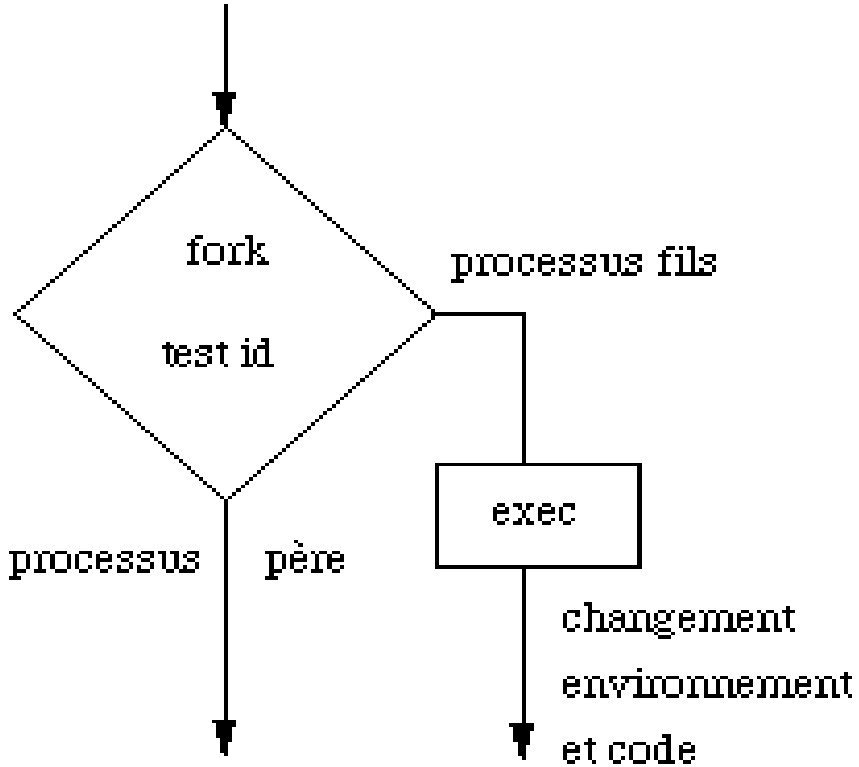
- Lorsqu'un appel exec est effectué, le noyau remplace le code du processus actuel par le code du programme exécutable spécifié, et il réinitialise les registres et les autres données du processus pour correspondre aux nouvelles données de l'image du programme exécutable.

- Cela signifie que l'image mémoire, les variables et les structures de données du processus sont entièrement remplacées par celles du nouveau programme.

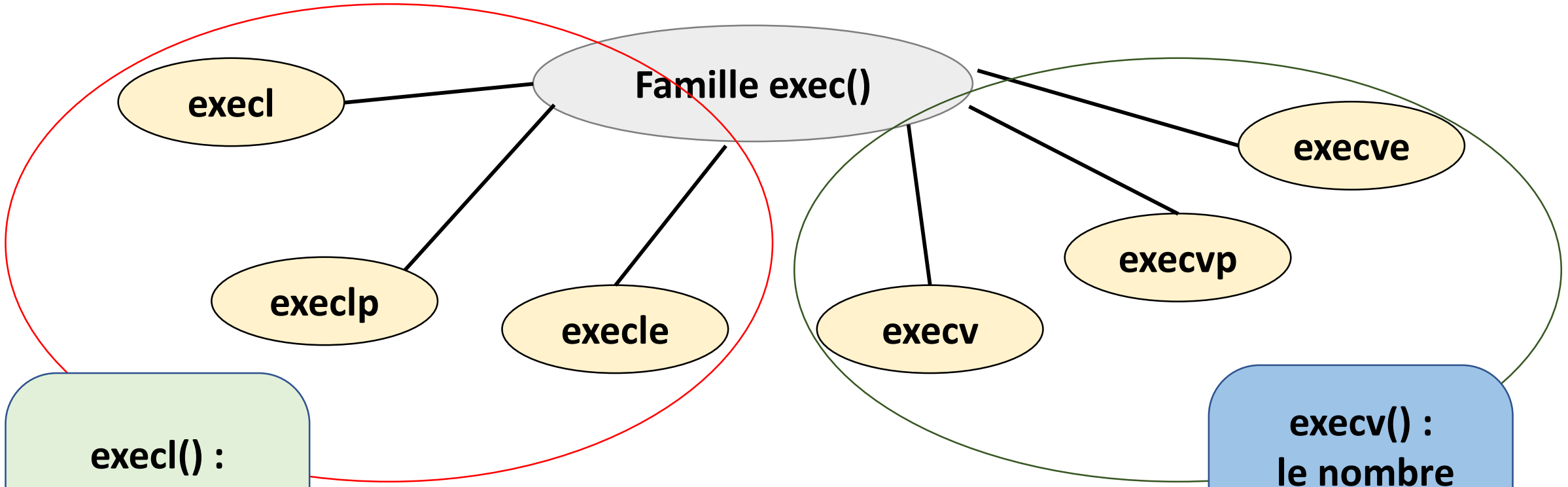


La primitive exec()

• Du point de vue système, il s'agit toujours du même processus : il a conservé le même pid.



La primitive exec()



execl() : le nombre des arguments du programme lancé est connu

execv() : le nombre des arguments du programme lancé n'est pas connu

exec() : remplacement du code d'un processus

```
int execl (char *ref, char *arg0, …, char *argn, 0)
int execv (char *ref, char *argv [ ]);
```

- « l » (resp. « v ») indique que les mots composant la ligne de commande du nouveau code exécutable sont passés séparément l'un à la suite de l'autre (rep. Sont passés via un vecteur de mots).

- « p » : indique que le chemin d'accès au nouveau code exécutable est sauvegardé dans la variable PATH.

- « e » signifie qu'il est possible d'ajouter un vecteur de variables d'environnement.