Chapitre III : Terminaison et attente du processus

Cours Système Exploitation II

1 LF info ISIMM

Dr. Sana BENZARTI

2024/2025

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

Le processus père et le processus fils s'exécutent d'une façon parallèle.

Impossible de prévoir l'ordre du déroulement. 

→ Il faut synchroniser le déroulement des processus père et fils

Endormir un processus et lancer l'autre.

La primitive sleep() peut assurer une synchronisation d'exécution entre les processus et ordonner le déroulement du programme.

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

Exemple1.c : Utilisation de sleep()

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(){
    int pid;
    printf("Je suis %d le père, je vais enfanter.\n", getpid());
    pid = fork();
    if(pid==0)
    {
        printf("Bonjour. Je suis %d le fils\n",getpid());
       sleep(2);/*Bloque le processus pendant 2 s (endormir)*/
        printf("Quel beau monde !!\n");
       sleep(4);/*Bloque le processus pendant 4 s (endormir)*/
        printf("Je ne vais plus mourir\n");

    }
    else
    {
        sleep(5);/*Bloque le processus pendant 5 s (endormir)*/
        printf("Je vais mourir moi %d\n", getpid());
       sleep(3);/*Bloque le processus pendant 3 s (endormir)*/
        printf("Mon heure est venue: Adieu %d mon fils\n",pid);
    }
}
```

Résultats

Bonjour, Je suis 2019 le père, je v

enfanter.

//Attente de 5s du père

Bonjour, Je suis 2020 le fils.

//Attente de 2s du fils

Quel beau monde !!

//Attente de 4s du fils

Je vais mourir moi 2019

//Attente de 3s du père

Je ne vais plus mourir.

Mon heure est venue. Adieu 2020 mon fil

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

Il faut synchroniser les fins d'exécution des processus afin d'éviter les processus zombies.

→ En communiquant certaines données:

- La notification de la fin du processus fils (code du retour du processus fils)

- L'obligation de l'attente du processus père

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

Un processus se termine :

- Par une demande d'arrêt volontaire (appel système exit() ou retrun)

- Par un arrêt forcé provoqué par un autre processus (appel système kill())

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

Exemple2.c (primitive kill())

```
include <unistd.h> //Pour l'instruction fork();
include <stdio.h>
include <signal.h> //pour l'instruction kill et le signal SIGKILL
nt main()

   int i=0;
   int pid;
   char c;

   pid=fork();
   if (pid==0)
   {
         printf("Je suis le fils Num %d\n", getpid());
         printf("Mon pere %d va me tuer\n",getppid());
         while(1)
             printf("Je suis encore vivant\n");
   }
   else
   {
   sleep(1);
   printf("Je suis le pere Num %d\n",getpid());
   printf("Je vais tuer mon fils %d\n",pid);
   kill(pid,SIGKILL);
   printf("Mon fils est mort, je vais mourir moi %d\n",getpid());
   }
```

Résultats

```
//Endormir le père 1s
Je suis le fils Num 8173
Mon pere 8172 va me tuer
Je suis encore vivant
Je suis encore vivant
Je suis le pere Num 8172
Je suis encore vivant
Je suis encore vivant
Je vais tuer mon fils 8173
//le processus père tue le processus fils
```

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

#include <stdlib.h>
Void exit (int status);

La primitive **exit(status)** permet d'arrêter explicitement l'exécution d'un processus.

Le paramètre **status** spécifie un code de retour, entre 0 et 255, à renvoyer au processus père.

Par convention, en cas de **terminaison normale**, un processus doit retourner la **valeur 0**.

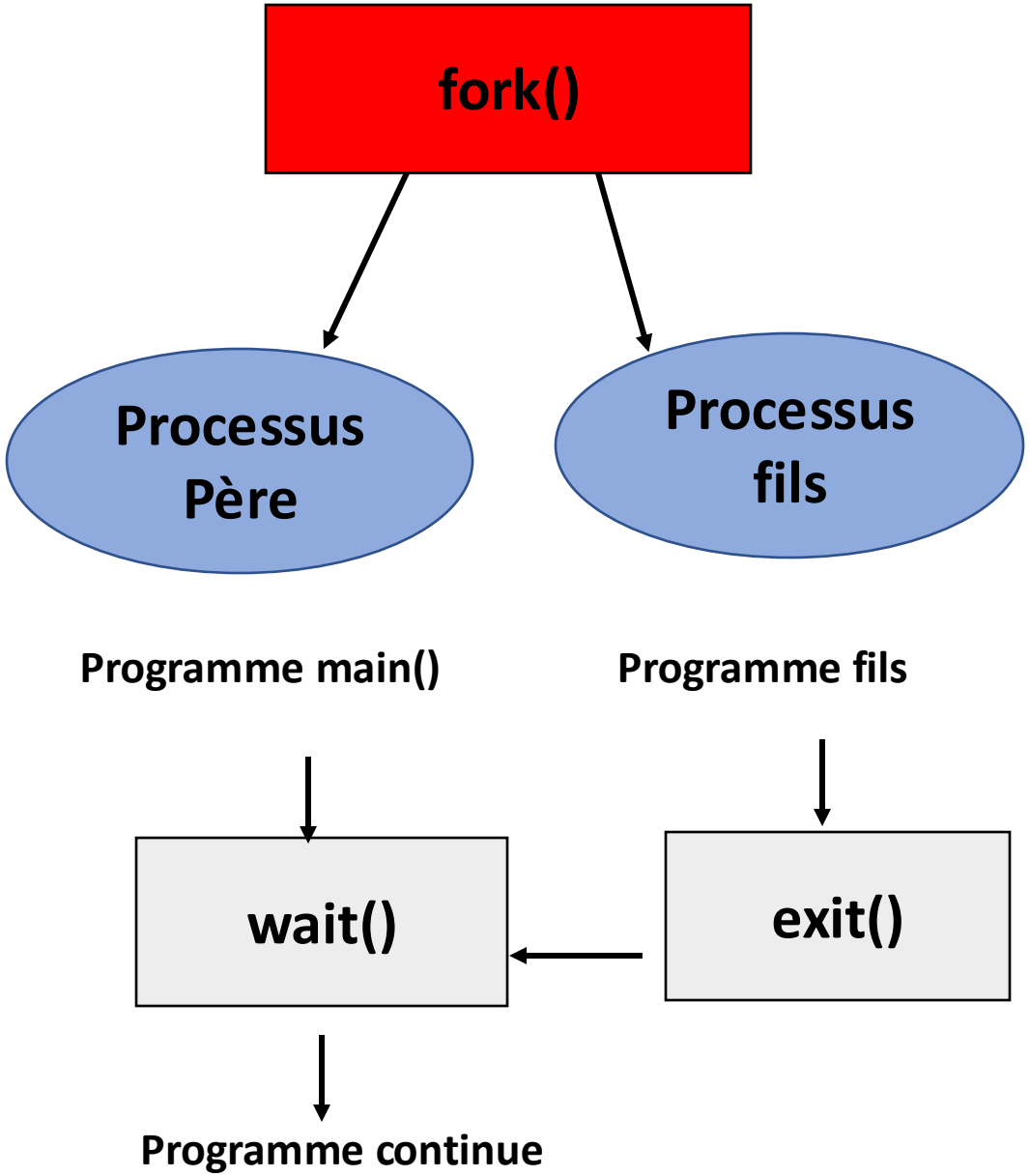
Avant de terminer l'exécution du processus, exit() exécute les fonctions de « nettoyage » des librairies standards : toutes les ressources systèmes qui lui ont été allouées sont libérées.

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

Le processus père, par un appel de la primitive wait(), « récupère » la mort de son fils, supprime l'entrée de la table des processus concernant son fils achevé.

→ Le processus fils à l'état zombie disparaît complétement.

→ Il faut prendre garde de l'appel wait qui et bloquant.
→ L'exécution du père est suspendue jusqu'à ce qu'un fils se termine
→ Il faut mettre autant d'appels de wait qu'il y a de fils



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

Attente d'un processus fils : 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()

Attente d'un processus fils : **wait**()

Les informations à propos de la bonne ou mauvaise terminaison d'un processus peuvent être accédées facilement à l'aide des macros suivantes définies dans **sys**/**wait**.**h**

- WIFEXITED (status) : elle renvoie vrai si le fils s'est terminé normalement par un appel à exit() ou bien un retour de main().
- WEXITSTATUS (status) : (si WIFEXITED (status) renvoie vrai) renvoie le code de retour du processus passé à exit() ou la valeur retournée par la fonction main().
- WIFSIGNALED (status) : renvoie vrai si le fils s'est terminé à cause d'un signal.
- WTERMSIG (status) : (si WIFSIGNALED (status) renvoie vrai) renvoie la valeur du signal qui a provoqué la terminaison du processus fils.

```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()

Le père par un wait(), récupère la mort de son fils.

- 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

- -1 en cas d'échec

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

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

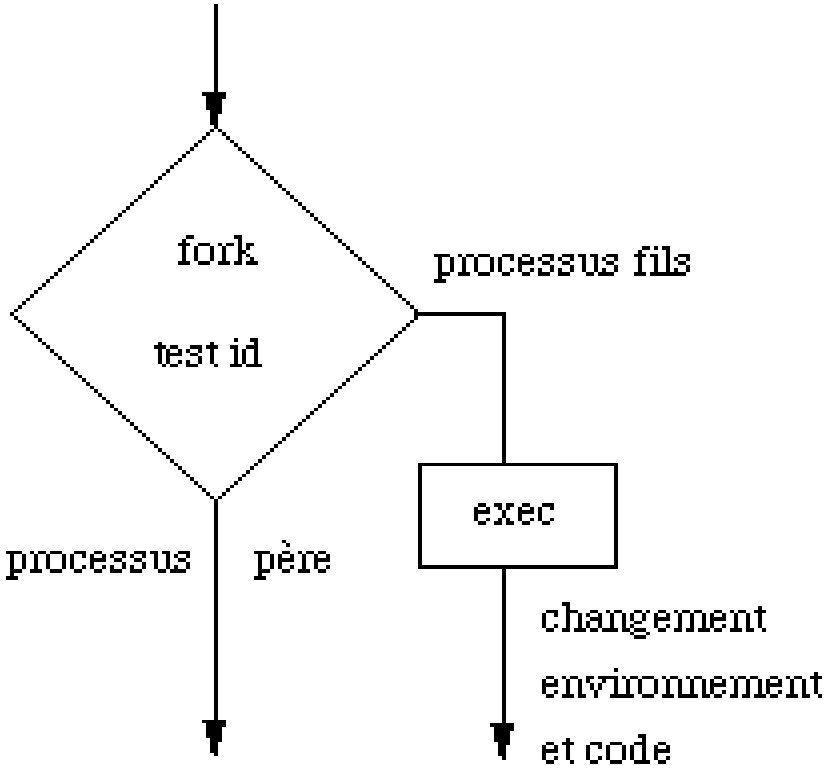
- Option 0 : waitpid() va bloquer le processus appelant jusqu'à ce que le processus enfant spécifié par le PID ait terminé son exécution. Pendant ce temps, le processus appelant est mis en pause et ne reprendra son exécution que lorsque waitpid() aura renvoyé.

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

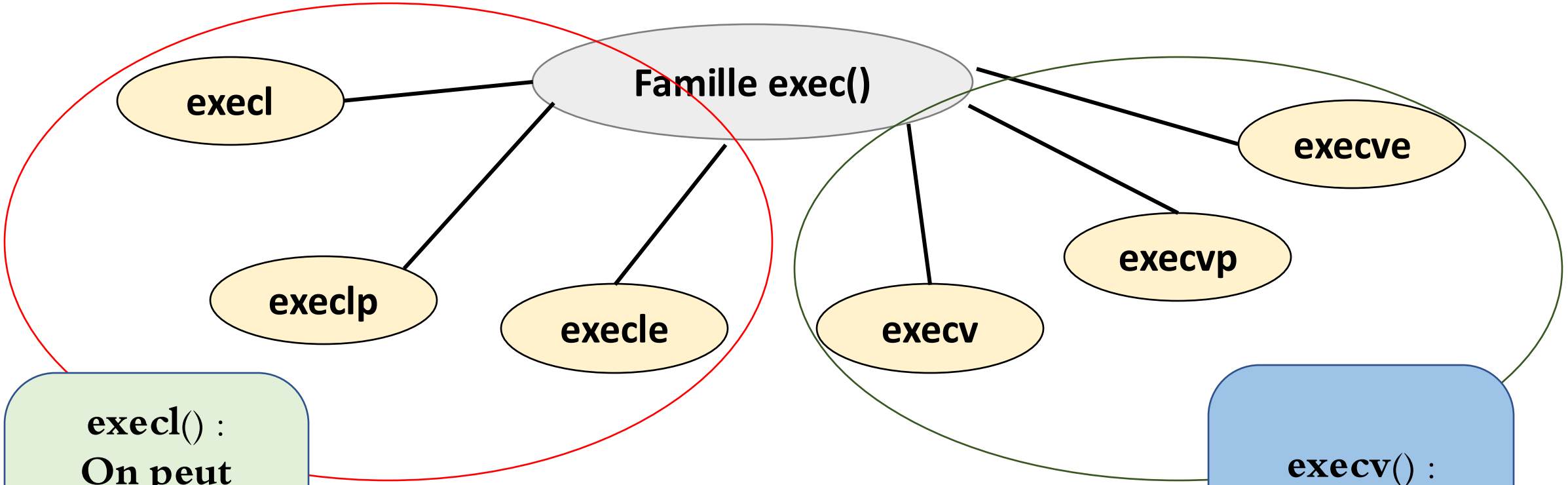
La primitive exec()

- 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()



execl() : On peut passer une liste d'arguments séparés par des virgules

execv() : on peut passer un vecteur d'arguments

Autres exemples avec exec

- execl ("/bin/mkdir", "mkdir", nom_du_dossier, NULL) ;

- /bin/mkdir : est le chemin de la commande exécutable
- mkdir : nom de la commande
- Nom_du_dossier : syntaxe relative a la commande, options de la commande, …
- NULL : indique la fin de la liste des arguments

Autres exemples avec exec

- execlp ("mkdir", nom_du_dossier, NULL) ;

- p dans l'execl désigne que le PATH (chemin) de la commande est inclus dans exec
- mkdir : nom de la commande
- Nom_du_dossier : syntaxe relative a la commande, options de la commande, …
- NULL : indique la fin de la liste des arguments

Autres exemples avec exec

- execle ("/bin/mkdir", "mkdir", nom_du_dossier, NULL, NULL) ;

- e dans l'execl designe qu'on peut inclure des variables d'environnements avec exec. Ici, la variable d'environnement est NULL.
- /bin/mkdir : est le chemin de la commande exécutable
- mkdir : nom de la commande
- Nom_du_dossier : syntaxe relative a la commande, options de la commande, …
- NULL : indique la variable d'environnement qui est vide
- NULL : indique la fin de la liste des arguments

Autres exemples avec exec

- char * args[] = {"mkdir", nom_du_dossier, NULL}
- execv ("/bin/mkdir", args) ;

- Avec execv, on peut passer un vecteur d'arguments

- /bin/mkdir : est le chemin de la commande exécutable
- args : le vecteur d'arguments

Autres exemples avec exec

- char * args[] = {"mkdir", nom_du_dossier, NULL}
- execvp (args) ;

- p dans l'execv désigne que le PATH (chemin) de la commande est inclus dans exec

- args : le vecteur d'arguments

Autres exemples avec exec

- char * args[] = {"mkdir", nom_du_dossier, NULL}
- execve ("/bin/mkdir", args, NULL) ;

- e dans l'execv désigne qu'on peut inclure des variables d'environnements avec exec. Ici, la variable d'environnement est NULL.

- /bin/mkdir : est le chemin de la commande exécutable
- args : le vecteur d'arguments
- NULL : indique la variable d'environnement qui est vide