Suite Session 2

L'héritage du fils

- Le processus fils hérite de beaucoup d'attributs du père mais n'hérite pas:
  - De l'identification de son père
  - Des temps d'exécution qui sont initialisés à 0
  - Des signaux pendants (arrivées, en attente d'être traités) du père
  - De la priorité du père; la sienne est initialisée à une valeur standard
  - Des verrous sur les fichiers détenus par le père

- Le fils travaille sur les données du père s'il accède seulement en lecture. S'il accède en écriture à une donnée, celle-ci est alors recopiée dans son espace local.

L'héritage du fils: exemple 1

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int m=2;
void main(void) {
   int i, pid;
   printf("m=%d\n", m);
   pid = fork();
   if (pid > 0) {
      for (i=0; i<5; i++) {
         sleep(1);
         m++;
         printf("\nje suis le processus père: %d, m=%d\n", i, m);
         sleep(1);

      }
   }
   if (pid == 0) {
      for (i=0;i<5; i++) {
         m = m*2;
         printf("\nje suis le processus fils: %d, m=%d\n", i, m);
         sleep(2);
      }
   }
   if (pid < 0)  printf("Probleme de creation par fork()\n");

}
```

L'héritage du fils: exemple 2

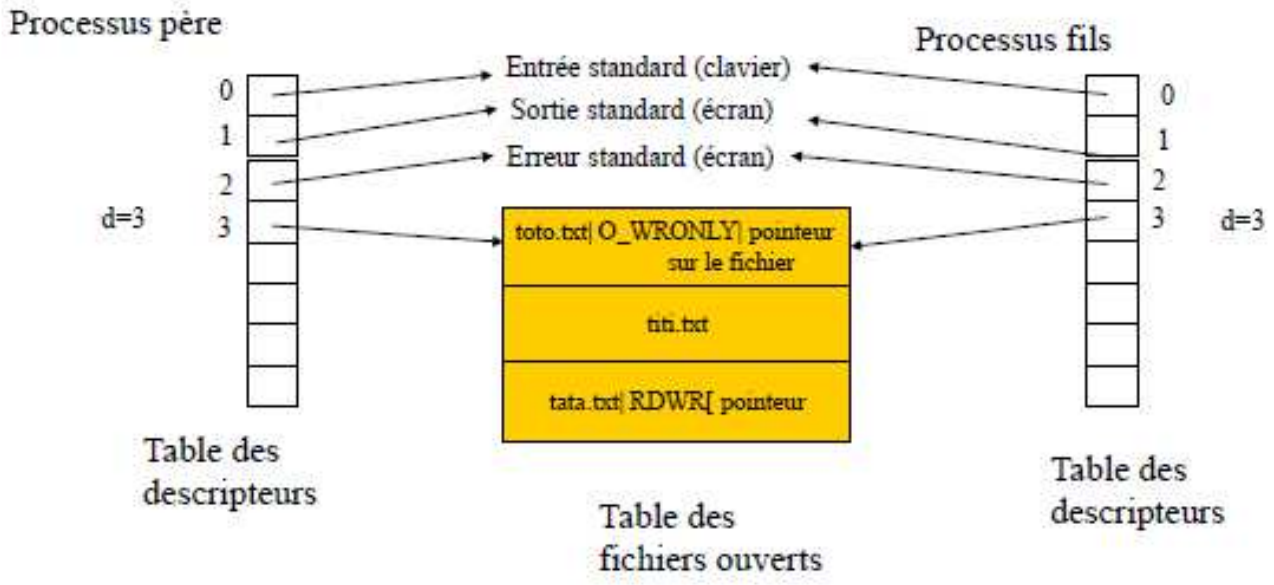
```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <fcntl.h>

int d;

int main(void) {
        d= open("toto.txt", O_CREAT | O_WRONLY , 0640);
        printf("descript=%d\n", d);

        if (fork() == 0) {
                write(d, "ab", 2);
        }
        else {
                sleep(1);
                write(d, "AB", 2);
                wait(NULL);
        }
        return 0;
}
```

Héritage du fils: Tables de descripteurs



Synchronisation père-fils

- Les processus zombis
  - Tout processus qui se termine passe dans l'état zombi jusqu'à ce que le processus père prenne connaissance de sa terminaison; cela pour que le processus père puisse accéder au code retourné par son fils
  - Un processus zombi affiché par la commande *ps* donne Z comme état du processus et 0 pour la taille

Synchronisation père-fils

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

int main(void)
{
        if (fork() == 0) {
                printf("Fin du processus fils de N° %d\n", getpid());
                exit(2);
        }
        sleep(15);
        wait(NULL);
}
```

```
$ p_z &
Fin du processus fils de N°xxx

$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR    SZ WCHAN     TTY      TIME CMD
000 S   501  8389  8325  0  73   0    -   749 wait4   pts/2  00:00:00 bash
000 S   501 11445  8389  0  71   0    -   342 nanosl pts/2  00:00:00 p_z
044 Z   501 11446 11445  0  70   0    -     0 do_exi pts/2  00:00:00 p_z <defunct>
000 R   501 11450  8389  0  74   0    -   789 -       pts/2    00:00:00 ps
```

Exécution dans le même Terminal pour pouvoir voir l'état

ps -aux

C'est l'instruction `sleep(15)` qui en endormant le père pendant 15 secondes, rend le fils zombi (état " Z " et intitulé du processus: " <defunct>")

Primitives wait et waitpid

```
#include <sys/types.h> #include <sys/wait.h>
pid_t wait(int *etat);
pid_t waitpid(pid_t pid, int *etat, int options);
```

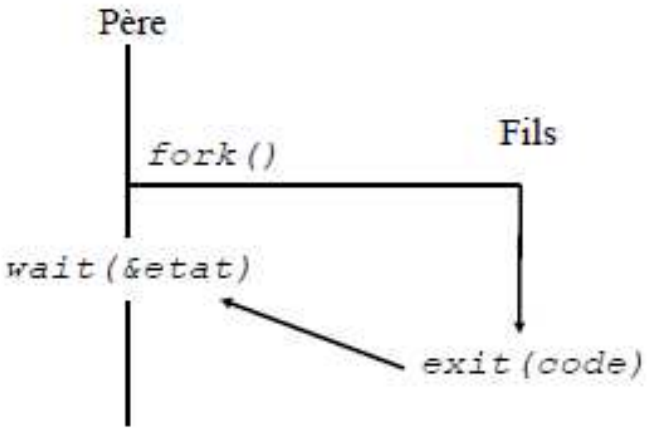
- Ces deux primitives permettent l'élimination des processus zombis et la synchronisation d'un processus sur la terminaison de ses descendants avec récupération des informations relatives à cette terminaison

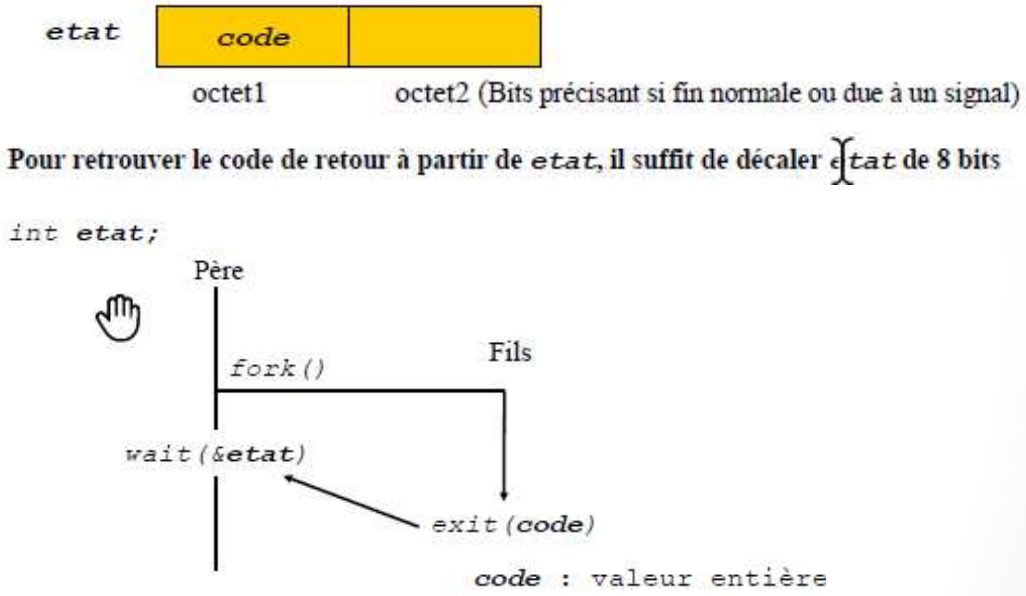
- Elles provoquent la suspension du processus appelant jusqu'à ce que l'un de ses processus fils se termine

- La primitive waitpid permet de sélectionner un processus particulier parmi les processus fils (pid)

Synchronisation père-fils

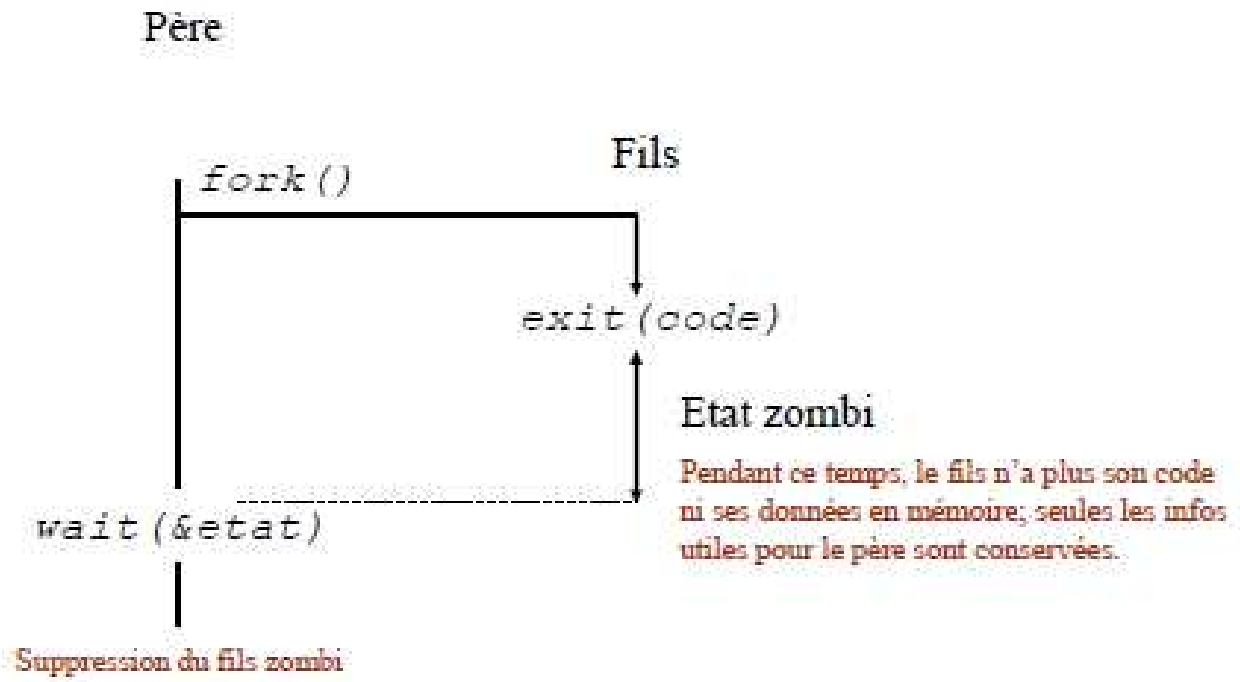
- Cas 1 : A l'exécution, le père attend effectivement le fils





Synchronisation père-fils

- Cas 2 : A l'exécution, le fils se termine avant le wait du père



Exercices

```
/* perefils.c */
#include <stdio.h>
#include <unistd.h>

int main(void)
{
int  PID;
if ((PID= fork()) == 0 )
        {               /* processus fils */
        printf("Le fils: mon pid est %d, le pid de mon papa est %d \n",getpid(), getppid());
        printf("\t\t fork a retourné au fils la valeur %d\n",PID);
        }
        else if (PID > 0)
        {               /*  processus père */
        printf("Le père: mon pid est %d, le pid de mon papa à moi est %d \n", getpid(), getppid());
        printf("\t\t fork a retourné au père la valeur %d\n",PID);
        sleep(2);
        }
        else perror("Erreur dans fork !!!");
return 0;
}
```

Exercices

```
/* perefil1.c */
#include <stdio.h>
#include <unistd.h>

int main(void)
{
int  PID;
if ((PID= fork()) == 0 )
        {               /* processus fils */
        printf("1ere partie du fils \n");
        printf("\t 2eme partie du fils \n");
        printf ("\t\t 3eme partie du fils \n");
        }
        else if (PID > 0)
        {               /*  processus père */
        printf("1 ere partie du pere \n");
        printf("\t 2eme partie du pere \n");
        printf("\t\t 3eme partie du pere \n");
        }
        else perror("Erreur dans fork !!!");
return 0;
}
```

Exercices

```
/* perefil2.c */
#include <stdio.h>
#include <unistd.h>

int main(void)
{
int  PID;
if ((PID= fork()) == 0 )
        {               /* processus fils */
        printf("1ere partie du fils \n");
        printf("\t 2eme partie du fils \n");
        printf ("\t\t 3eme partie du fils \n");
        sleep(15);
        }
        else if (PID > 0)
        {               /*  processus père */
        printf("1 ere partie du pere \n");
        printf("\t 2eme partie du pere \n");
        printf("\t\t 3eme partie du pere \n");
        wait(NULL);
        }
        else perror("Erreur dans fork !!!");
return 0;
}
```

Communication avec l'environnement

- Int main(int argc, char *argv[], char *argp[])

- Le lancement d'un programme C provoque l'appel de sa fonction principale main() qui a 3 paramètres optionnels:
  - argc: nombre de paramètres sur la lignes de commande (y compris le nom de l'exécutable lui-même)
  - argv: tableau de chaînes contenant les paramètres de la ligne de commande
  - envp: tableau de chaînes contenant les variables d'environnement au moment de l'appel, sous la forme variable=valeur

- Les dénominations argc, argv et envp sont purement conventionnelles

Communication avec l'environnement : exemple 1

```
/* env.c */
 #include <stdio.h>
 #include <stdlib.h>
 int main(int argc, char *argv[], char *envp[])
 {
     int k;
     printf("Appel avec %d parametres\n",argc);
     printf("Le nom de la commande est %s\n",argv[0]);
     printf("Les arguments de la commande sont les suivants : ");
     for (k=1;k<argc;k++)  printf("%s  ",k,argv[k]);
     printf("\n Les variables d'environnement sont :\n");
     for(k=0;envp[k]!=NULL;k++)  printf("%s\n",envp[k]);
     exit(0);
 }
```

Communication avec l'environnement : exemple 2

```
/* getenv.c */
 #include <stdlib.h>
 #include <stdio.h>
 int main() {
      char *terminal; char * login;
      terminal=getenv("TERM");
      login=getenv("LOGNAME");
              printf("Vous vous etes connectes sous le nom de %s \n", login);
      printf("Le terminal est ");
      if (terminal==NULL)
      printf("inconnu\n");
      else
      printf("un %s\n",terminal);
      exit(0);
 }
```

Communication avec l'environnement: exemple

- Résultat

Primitives de recouvrement

- Objectif: charger en mémoire **à la place du processus** fils un autre programme provenant d'un **fichier binaire**

- Il existe 6 primitives exec. Les arguments sont transmis:
  - sous forme d'un tableau (ou vecteur, d'où le v) contenant les paramètres:
    - execv, execvp, execve
  - Sous forme d'une liste (d'où le l): arg_0, arg_1, …, arg_N, NULL:
    - execl, execlp, execle
  - La lettre finale *p* ou *e* est mise pour *path* ou *environnement*

- La fonction system("commande…") permet aussi de lancer une commande à partir d'un programme mais son inconvénient est qu'elle lance un nouveau Shell pour interpréter la commande.

Primitives de recouvrement –exemples

- L'exécution d'un fichier binaire
  - Envoi d'un mail par exemple

- La description des primitives exec

```
#include <stdio.h>

int main(void)
{
   char *argv[4]={"ls", "-l", "/", NULL};

   execv ("/bin/ls", argv);
   execl ("/bin/ls", "ls", "-l", "/", NULL);
   execlp("ls", "ls", "-l", "/", NULL);

   return 0;
}
```

Primitives de recouvrement –exemples

Com.c

```
#include <stdio.h>    /* prog qui affiche ses arguments et ses variables d'environnement */
main( int argc, char *argv[], char *arge[])
{ int i;
                for (i=0; i<argc ;i++) printf("%s \n",argv[i]);
                while(*arge != NULL)   printf("%s \n",*arge++);
}
```

Env_com.c

```
#include <stdio.h>
char *env[2];
main() {

if (fork() == 0)
                {
                                //env[0] = "TERM=vt320";
                                env[0] = "HOME=/home/bouda";
                                env[1] = NULL;
                                execle("com", "com","a",NULL,env);
                                perror ("erreur execle");

                }
                }
```

Exercices – fichier.c

- Créer un fichier toto.txt qui est ouvert par un processus père, le fils hérite du descripteur de toto.txt
- Cet exercice montre que les 2 processus accèdent au même fichier en utilisant le même descripteur
- Le fichier.c a été écrit de manière à laisser le temps au processus fils d'écrire la chaine « ab » dans le fichier toto.txt avant de permettre au processus père de lire à partir de ce fichier. Cette synchronisation étant réalisée grâce à l'insertion de la fonction sleep() à des endroits bien déterminés dans le programme. Seulement que le résultat nous monter que le père ne lit pas la chaine « ab » écrite au préalable par le fils car les deux processus partagent le même descripteur et par conséquent le même pointeur de fichier. C'est pourquoi lorsque le père tente de lire, il a son pointeur positionné en fin de fichier c'est-à-dire après la chaine « ab ».
  - A quoi sert le sleep() ici ?
  - Est-ce que le père lit la chaine « ab » ? Et pourquoi ?