Université du Havre, IUT, Dépt. Informatique

s4#, 2014/2015

Cours La programmation système

Nakechbandi M., LITIS, Email : nakech@free.fr

Chapitre 1

1. Objectifs
2. Contenu du cours
3. Rappel : Qu'est ce qu'un système d'exploitation
4. Système de gestion de fichiers : i-nœud, Répertoires, Arborescences et montage, Fichiers réguliers, Droits d'accès
5. Processus.
6. Primitives et commandes pour la gestion des processus : fork(), ps,

Références :

- F. Guinand, Polycopiés de cours, Université du Havre, 2013.
- J. Boukachour, Polycopiés de cours, Université du Havre, 2013.

Objectifs

1. Connaître le rôle et les services d'un système d'exploitation.

2. Pratiquer l'écriture de programmes en C. et savoir programmer un système multi-tâches et multi-utilisateur : Creation de processus, communication entrez processus, …

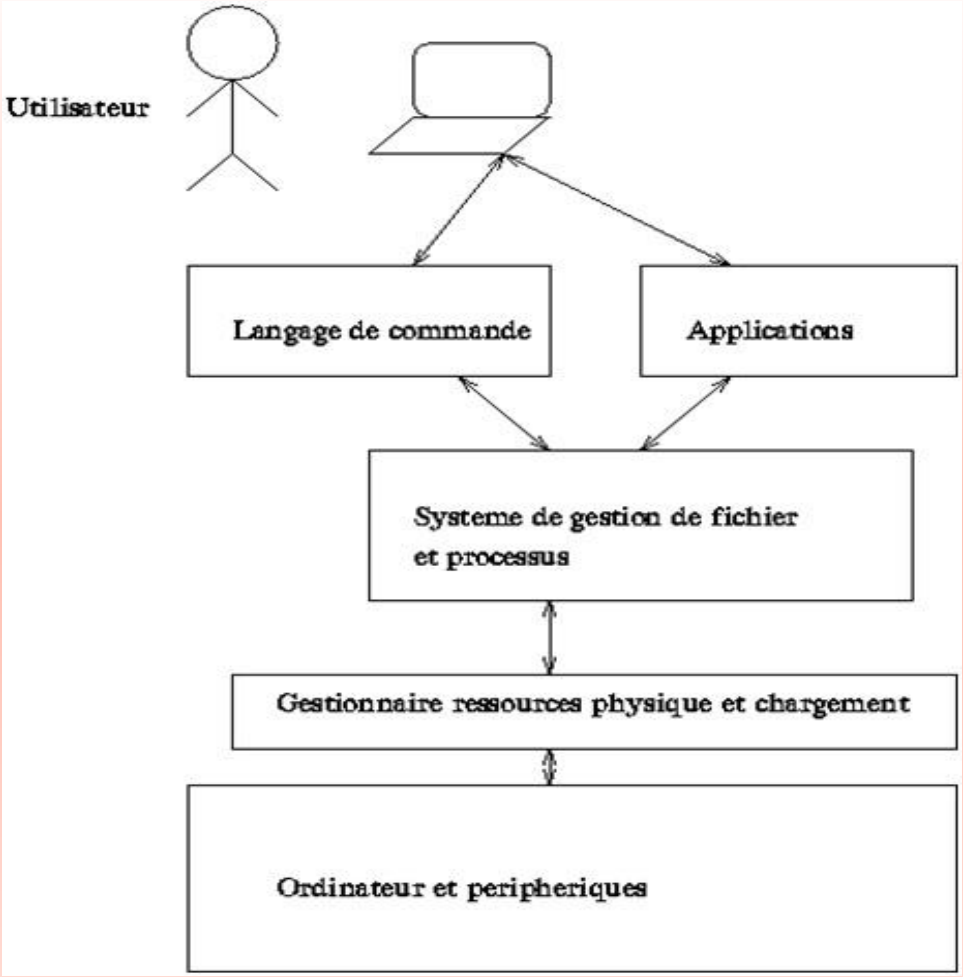
Contenu du cours

1. Principes des systèmes d'exploitation (SE) :
    - du matériel au logiciel : un ensemble de couches,
    - mécanismes d'exécution,
    - communication inter-processus.

2. programmation UNIX (programmation C) :
    - rôle du SE dans la programmation en langage de haut niveau (compilation, édition de liens, débogage...)

SE : Quelle place au sein de l'ordinateur ?

- Le logiciel permet de combler le fossé entre la machine physique et l'utilisateur.



- Les programmes sont classés en deux catégories : les logiciels d'applications, le logiciel de base. le système d'exploitation fait partie du logiciel de base.

Système de gestion de fichiers : Fichier UNIX

- Qu'est-ce qu'un fichier du point de vue du système UNIX ?
  - Un fichier est un ensemble de données (musique, vidéo, texte...), programmes exécutables, librairies, etc.
  - Ressources (physique ou logique) : `/dev/hda1`
- Un fichier possède des caractéristiques : localisation, type, droits.
- Ces caractéristiques sont stockées dans des tables ; à chaque fichier est associée une position dans une table.,

i-noeud

- Sous un système UNIX, un fichier quel que soit son type est identifié par un numéro appelé numéro d'inode
  Pour connaître le numéro d' i-nœud d'un fichier : `ls -i mon-fichier`
  Exemple :
- `ls -l`

```
drwxr-xr-x 2 da130417 users     4096  4 sept. 10:46 hsperfdata_da130417
drwxr-xr-x 3 nakechbm users     4096  2 sept. 13:38 info2$
drwxr-xr-x 4 nakechbm users     4096  3 sept. 11:48 info2#
 -rw-r--r-- 1 dufloh-rw-r--r-- 1 colinj   users 57148005  2 sept. 15:32
VM_Lhinux_TP_0_3.tar.gz
  …
```

- `ls -i`

```
     1081347 hsperfdata_da130417
     868365 info2$
     868411 info2#
     868418 VM_Lhinux_TP_0_3.tar.gz
       …
```

- Caractéristiques des fichiers : Propriétaire, type, droits, localisation.

Répertoires

- les répertoires jouent un rôle essentiel :
  - Structuration
  - désignation extérieur, indépendamment de leur position dans les tables du système et de leur localisation sur le disque.
- dans les répertoires, une association est réalisée entre la chaîne de caractères désignant le fichier et son i-noeud, c'est le lien physique,
- le même mécanisme prévaut pour les atures fichiers,
- la structure est arborescence et le point de départ est la racine absolue
- **remarque :** un répertoire n'est jamais vide car il contient toujours les liens **. et ..**

Arborescences et montage

- Pour chaque disque logique (partition), il existe une arborescence

- Pour pouvoir depuis la racine absolue accéder aux fichiers situés sur l'une de ces arborescences, il faut préalablement qu'elles soient reliées entres elles par le mécanisme du montage.

Exemple : sous corton je lance `ls -il /`

```
1810433 drwxr-xr-x   2 root root  4096 24 oct.   2013 bin
1728513 drwxr-xr-x 121 root root 12288  4 sept. 09:30 etc
1638401 drwxr-xr-x  13 root root  4096 31 août  23:55 home
…
```

on constate que `/home` `/bin` et `/etc` sont des répertoires racines d'un disque logique (une partition) et sont donc des points de montage.

Fichiers : Fichiers réguliers , Fichiers spéciaux

Fichiers réguliers

- Fichier sur disque dont le contenu est une suite de caractères.
- Caractérisé par sa longueur.
- L'interprétation du contenu est de la compétence des applications, mais pas du système.
- Exemple :

```
-rwxr-xr-x 1 toto users 7977 sep 21 2005 moyenne   fichier exécutable
-rw-r-r-   1 toto users 436 sep 21 2005 moyenne.c  fichier texte
-rw-r-r-   1 toto users 91277 sep 22 2005 tp1.pdf  fichier binaire
-rw-rw-r-  1 toto users 23942 sep 23 2005 tp1.tex  fichier texte
```

Fichiers spéciaux

- Sockets
- Liens
- tubes

Rappel : Répertoires UNIX

On retrouve presque toujours au niveau de la racine certains répertoires :

- `/bin et /usr/bin` : commandes UNIX pas spécifiques aux langages de commandes.
- `/etc` : fichiers système, fichiers de configuration.
- `/dev` : fichiers spéciaux associés aux ressources
- `/home` : répertoires des utilisateurs,
- `/tmp (resp. /usr/tmp)` : fichiers temporaires utilisés par les applications système (resp. applications des utilisateurs)
- `/var` : boîtes aux lettres, traces, queues d'impression et pages web.

Rappel : Droits d'accès

- `-rwxr-x-x 1 toto cadre 7735 sep 19 16 :43 moy.pl`
- est un fichier régulier, le propriétaire est toto, le groupe est cadre
- sur ce fichier, seul le propriétaire a les droits en écriture,

  par contre les membres du même groupe peuvent lire et exécuter le programmme, alors que tous les autres utilisateurs ne peuvent qu'exécuter ce programme, mais ils ne peuvent ni le lire, ni le modifier
- Modification des droits d'accès : Les droits représentent 3 vecteurs de 3 bits :
  - `r w x  r - x  - - x`
  - `1 1 1  1 0 1  0 0 1`
  - `7 5 1`

on peut modifier les droits à l'aide de la commande `chmod`

par exemple : `chmod 751 moyenne.pl`

autre syntaxe : `chmod u+x moyenne.pl`

Montage d'une clef USB (1)

- Connecter une clef USB c'est ajouter une branche à l'arborescence de fichiers.

Où ?

- Dans mon arborescence, j'ai prévu un point de montage pour ma clef dans le répertoire `/mnt`

  ⟹ en tant que superutilisateur : `mkdir /mnt/clefUSB`

- dans ce répertoire, il n'y a rien :

```
> ls -al /mnt/clefUSB/
total 8
drwxr-xr-x 2 root root 4096 avr 29 18 :07 ./
drwxr-xr-x 6 root root 4096 sep 8 11 :11 ../
```

- Détection
  - La clef une fois introduite est détectée par le système.
  - Elle fait partie des device (ressources)

Montage d'une clef USB (2)

Ressources avant introduction

> ls -la /dev/sd*

```
brw-r-- 1 root disk 8, 0 sep 18 11 :26 /dev/sda
brw-r-- 1 root disk 8, 1 sep 18 11 :26 /dev/sda1
brw-r-- 1 root disk 8, 2 sep 18 11 :26 /dev/sda2
brw-r-- 1 root disk 8, 5 sep 18 11 :26 /dev/sda5
brw-r-- 1 root disk 8, 6 sep 18 11 :26 /dev/sda6
brw-r-- 1 root disk 8, 7 sep 18 11 :26 /dev/sda7
brw-r-- 1 root disk 8, 8 sep 18 11 :26 /dev/sda8
brw-r-- 1 root disk 8, 9 sep 18 11 :26 /dev/sda9
```

Ressources après introduction

> ls -la /dev/sd*

```
brw-r-- 1 root disk 8, 0 sep 18 11 :26 /dev/sda
brw-r-- 1 root disk 8, 1 sep 18 11 :26 /dev/sda1
brw-r-- 1 root disk 8, 2 sep 18 11 :26 /dev/sda2
brw-r-- 1 root disk 8, 5 sep 18 11 :26 /dev/sda5
brw-r-- 1 root disk 8, 6 sep 18 11 :26 /dev/sda6
brw-r-- 1 root disk 8, 7 sep 18 11 :26 /dev/sda7
brw-r-- 1 root disk 8, 8 sep 18 11 :26 /dev/sda8
brw-r-- 1 root disk 8, 9 sep 18 11 :26 /dev/sda9
brw-r-- 1 root disk 8, 16 sep 19 12 :28 /dev/sdb
brw-r-- 1 root disk 8, 17 sep 19 12 :28 /dev/sdb1
```

Montage d'une clef USB (3)

La clef a donc été détectée, mais que trouve-t-on dans /mnt/clefUSB ? Rien ! !

```
>       ls -la /mnt/clefUSB/
        total 8
        drwxr-xr-x 2 root root 4096 sep 19 11 :40 .
        drwxr-xr-x 3 root root 4096 sep 19 11 :41 ..
```

En fait, le système de fichiers de la clef n'a pas encore été rattaché à l'arborescence principale. Il faut donc monter le système de la clef pour le rattacher à l'arborescence générale.

```
> mount /dev/sdb1 /mnt/clefUSB
```

C'est fait ! !

```
ls -la /mnt/clefUSB/
total 8942
drwxr-xr-x 4 root root 16384 jan 1 1970 .
drwxr-xr-x 3 root root 4096 sep 19 11 :41 ..
-rwxr-xr-x 1 root root 1864 sep 6 2006 configureWlan
-rwxr-xr-x 1 root root 111833 jan 8 2007 ipc.pdf
drwxr-xr-x 5 root root 2048 sep 21 2006 javadoc
-rwxr-xr-x 1 root root 9012380 nov 15 2006 java.tgz
-rwxr-xr-x 1 root root 265 sep 6 2006 wlan-UBIMOB.conf
-rwxr-xr-x 1 root root 119 sep 6 2006 wpa_supplicant.conf
```

Occupation des systèmes de fichiers

utilisation de l'espace par les différents systèmes de fichiers :

```
> df
 Sys. de fich. 1K-blocs Occupé Dispo Capacité Monté sur
 /dev/sda5 6016568 1208276 4502660 22% /
 udev 1038124 96 1038028 1% /dev
 /dev/sda9 40330028 34118260 4163084 90% /home
 /dev/sda1 30233896 27686552 1011532 97% /perso
 /dev/sda7 11084636 2868864 7652696 28% /usr
 /dev/sda8 1510032 350736 1082588 25% /var
 /dev/sr0 645640 645640 0 100% /media/CDROM
 /dev/sdb1 128484 12076 116408 10% /mnt/clefUSB
```

Retirer une partie de l'arborescence montée

- en tant que root

> **umount /mnt/clefUSB**

- on peut également faire en sorte que les utilisateurs soient autorisés à monter la clef, en incluant une ligne adéquat dans le fichier qui décrit de quelle manière les ressources doivent être montées sur le SGF : `/etc/fstab`
- Gestion automatique

more /etc/fstab

```
/dev/sda5 / ext3 acl,user_xattr 1 1
/dev/sda9 /home ext3 acl,user_xattr 1 2
/dev/sda1 /perso ext3 acl,user_xattr 1 2
/dev/sda7 /usr ext3 acl,user_xattr 1 2
/dev/sda8 /var ext3 acl,user_xattr 1 2
/dev/sda6 swap swap defaults 0 0
proc /proc proc defaults 0 0
sysfs /sys sysfs noauto 0 0
debugfs /sys/kernel/debug debugfs noauto 0 0
usbfs /proc/bus/usb usbfs noauto 0 0
devpts /dev/pts devpts mode=0620,gid=5 0 0
```

Processus : Définition

•Un processus est un programme en cours d'exécution. Par exemple, chaque fois que l'on lance la commande ls, un processus est créé durant l'exécution de la commande.

- Un processus est identifié par un numéro unique que l'on appelle le PID (identifiant).
- Un processus dispose d'un processus père que l'on appelle le PPID (Parent PID).
- La particularité d'un processus est de s'exécuter avec les droits accordés à l'utilisateur qui a lancé la commande. Ceci participe fortement à la sécurité du système. Ainsi, si un utilisateur contracte un programme malveillant (un virus par exemple), le processus sera soumis au droit d'accès de cet utilisateur, et ne pourra pas effectuer des opérations non autorisées (comme par exemple modifier le fichier de mots de passe).
- Au démarrage de l'ordinateur, le système charge le noyau Linux qui se charge de l'initialisation du matériel et de la détection des périphériques. Ceci fait, il démarre ensuite le processus init qui a comme particularité d'être le premier processus et de toujours utiliser le PID 1. Ce processus démarre ensuite des processus noyaux (dont le nom est noté entre crochets), et les premiers processus systèmes.
- Chaque processus a ainsi un père (sauf init), et peut être à son tour le père d'autres processus, etc.
- La commande `pstree` permet de visualiser l'arbre des processus. L'option -p permet de visualiser les PID de chaque processus.
- La commande ps : `ps -aux`

Exemple sous corton je lance : `pstree -p`

```
init(1)-+-Xprt(2239)
        |-acpid(1340)
        |-apache2(1749)-+-apache2(15635)
        |               |-apache2(15636)
        |               |-apache2(15637)
        |               |-apache2(15638)
        |               |-apache2(15639)
        |               |-apache2(17081)
        |               |-apache2(17082)
        |               |-apache2(17085)
        |               `-apache2(17086)
        |-atd(1759)
        |-automount(2115)-+-{automount}(2116)
        |                 |-{automount}(2117)
        |                 |-{automount}(2127)
        |                 `-{automount}(2132)
        |-console-kit-dae(2392)-+-{console-kit-da}(2393)
        |                       |-{console-kit-da}(2394)
        |                       |-{console-kit-da}(2395)
        |                       |-{console-kit-da}(2396)
...
        |-cron(2154)
        |-cupsd(1899)
        |-dbus-daemon(1415)
        |-exim4(2101)
        |-getty(2368)
        |-getty(2369)
        |-getty(2370)
        |-getty(2371)
        |-getty(2372)
        |-getty(2373)
        |-hald(1812)-+-hald-runner(1813)-+-hald-addon-acpi(2100)
        |            |                   |-hald-addon-inpu(2059)
        |            |                   |-hald-addon-stor(2088)
        |            |                   `-hald-addon-stor(2097)
        |            `-{hald}(1819)
        |-in.tftpd(1386)
        |-nscd(1378)-+-{nscd}(1396)
        |            |-{nscd}(1397)
        |            |-{nscd}(1398)
        |            |-{nscd}(1399)
        |            |-{nscd}(1400)
        |            |-{nscd}(1401)
        |            |-{nscd}(1402)
        |            |-{nscd}(1776)
        |            `-{nscd}(15640)
        |-nslcd(1424)-+-{nslcd}(1431)
        |             |-{nslcd}(1432)
        |             |-{nslcd}(1433)
        |             |-{nslcd}(1434)
        |             `-{nslcd}(1435)
        |-ntpd(1732)
        |-portmap(891)
...
```

2 types de processus : processus système , processus utilisateur

Processus système : propriétaire : superutilisateur (superuser).

- `init` : assure l'existence d'un processus pour chaque terminal de Commandes, il le fait en invoquant la commande getty processus père de tous les processus shells créés par l'utilisateur.
- `cron` : assure le lancement de commandes à des dates spécifiques
- `inetd` : c'est l'oreille du système, écoute les ports d'entrée (/etc/services) correspondance numéro de port $ service arrivée d'une requête sur un port : inetd supervise tous les services réseau qui utilisent les protocoles internet

processus utilisateur

Création d'un processus :

- lancement d'un programme : création d'un processus par le shell ⇒ `ls`

- création dynamique de processus ? ⇒ primitive `fork`

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

  - fork() créé un nouveau processus à partir d'un processus existant,
  - le nouveau processus est appelé processus fils et l'existant est appelé processus père,
  - la valeur de retour de fork() prend une valeur différente dans le processus père et dans le processus fils :
    - 0 dans le fils
    - le pid du processus fils dans le père
    - -1 en cas d'échec

Un premier exemple de fork en C

```
#include<unistd.h>
#include <sys/types.h>
#include<stdio.h>
main() {
pid_t pid ;
switch(pid=fork()) {
case(pid_t)-1 :
    perror("pb dans le fork") ;
    exit(2) ;
case(pid_t)0 :
    printf("valeur de fork = %d \n",pid) ;
    printf("je suis %d, mon père est %d \n",getpid(),getppid()) ;
    exit(0) ;
default :
    printf("valeur de fork = %d \n",pid) ;
    printf("je suis le père %d et mon père est %d n", getpid(),
    getppid()) ; exit(0) ; }}
```

Primitives relatives à la gestion de processus en C : Synchronisation père/fils

- rien ne garantie que le fils se termine avant le père
  ⇒ le père doit attendre la fin du processus fils pour accéder aux résultats

- Primitives de synchronisation : `wait ();  waitpid();`

- l'appel à cette primitive par le processus père le bloque tant que le fils n'est pas terminé. Syntaxe :

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

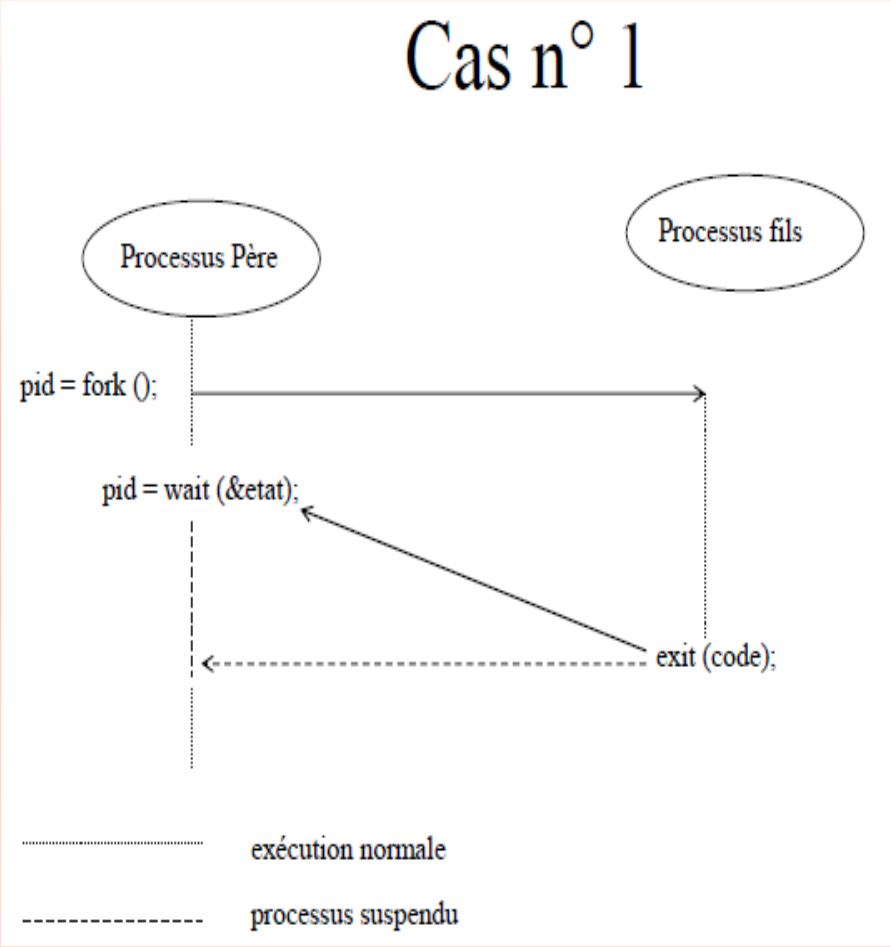
- status apporte des informations sur les conditions d'arrêt du fils
- options permet d'effectuer un appel non bloquant et d'avoir des informations sur les fils bloqués (généralement 0).

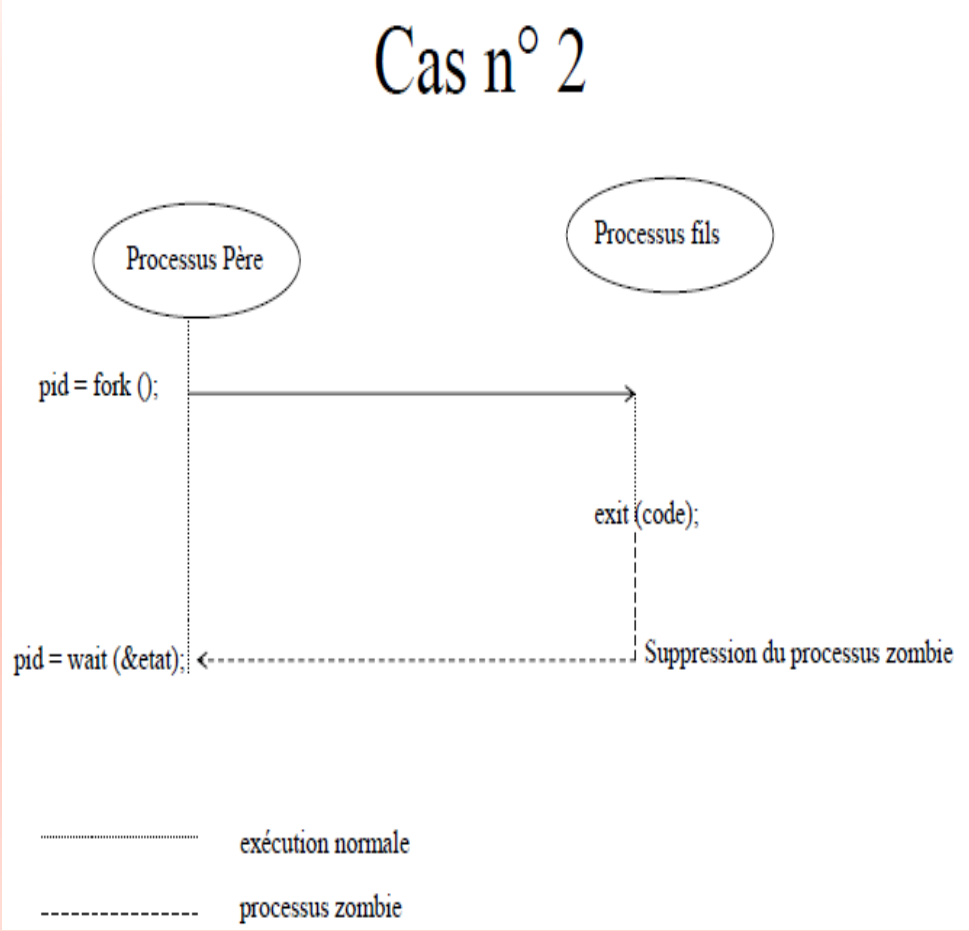
Exemple : fork, **wait**

```
int pid,
code=10,
etat;
pid=fork ( );
if (pid == -1) { perror ("erreur fork"); exit(1);}
else if (pid ==0) { /* code exécuté par le fils */
        ……..
        exit (code);
        }
        else { /*code exécuté par le père */
        ……..
        wait (& etat);
        ……..
        }
```

Actions du exit

- On libère tout sauf l'entrée dans la table des processus (processus en état zombie)
- exit (n) retourne un code de sortie n qui pourra être récupéré par le père par la primitive `wait`



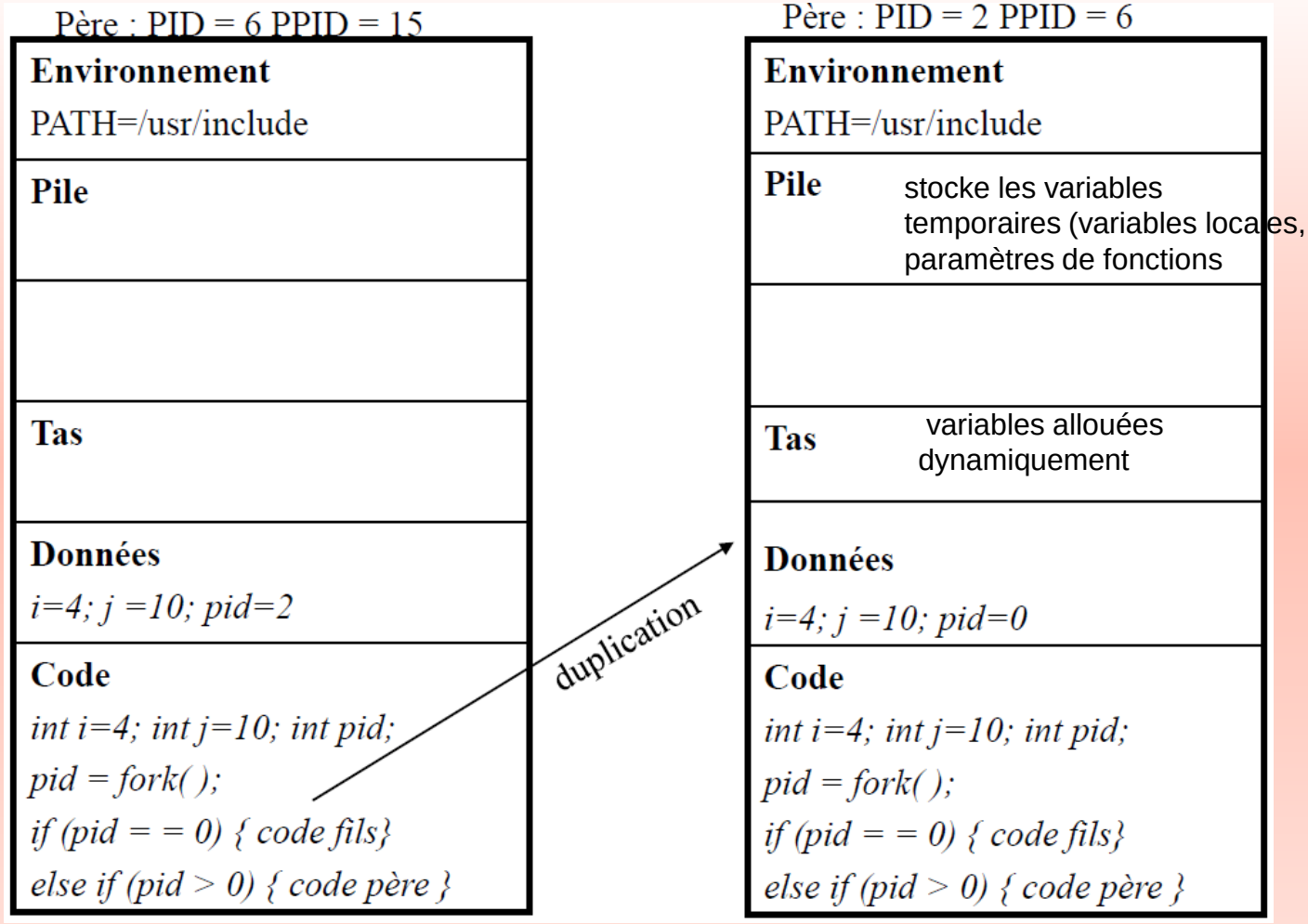


Université du Havre, IUT, Dépt. Informatique
s4#, 2014/2015
Cours La programmation système
Nakechbandi M., LITIS, Email : nakech@free.fr

Chapitre 1 suit
Rappel : Action du fork

- Allocation d'une entrée dans la table de processus
- Duplication du père
  - Duplication de la zone u (descripteurs de fichiers ouverts, répertoire courant, registres, variables d'environnement, ….)
  - Partage du **code** et des **données**
- Retour de la valeur du pid (numéro du fils) dans le processus père
- Retour de la valeur 0 dans le processus fils

Action du fork



Action du fork : Exemple 1

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
int main(){
        int pfils, i=1, nbProcessus=8;
        printf("debut P%d ==> pid = %d \n",i,getpid());
        for(i=2 ; i<=nbProcessus ; i++){
                pfils=fork();
                if(pfils==-1){
                        perror("fork a echoue");
                        exit(1);}
                if(pfils ==0) // fils
                        {printf("processus fils P%d ==> pid =
          %d et ppid = %d \n",i,getpid(),getppid());break;}
                else // pere
                  {printf("----->i=%d,fin du pere dont le pid
                    =%d  \n",i,getpid());  }
                                        }
}
```

Resultat de l'execution :

```
debut P1 ==> pid = 12315
---------->fin de l'iteraion i=2 du pere dont le pid =12315
---------->fin de l'iteraion i=3 du pere dont le pid =12315
processus fils P2 ==> pid = 12316 et ppid = 12315
---------->fin de l'iteraion i=4 du pere dont le pid =12315
processus fils P3 ==> pid = 12317 et ppid = 12315
processus fils P4 ==> pid = 12318 et ppid = 12315
---------->fin de l'iteraion i=5 du pere dont le pid =12315
processus fils P5 ==> pid = 12319 et ppid = 12315
---------->fin de l'iteraion i=6 du pere dont le pid =12315
---------->fin de l'iteraion i=7 du pere dont le pid =12315
processus fils P6 ==> pid = 12320 et ppid = 12315
processus fils P7 ==> pid = 12321 et ppid = 12315
---------->fin de l'iteraion i=8 du pere dont le pid =12315
processus fils P8 ==> pid = 12322 et ppid = 12315
```

On remarque que les 8 fils viennent du même père

On remarque également qu'il y a plusieurs fils par iteration
onc n'y a pas de synchronisation entre le père et les fils.

Action du fork : Exemple 2

```
 #include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

int main(){
        int pfils, i=1, nbProcessus=8;
        printf("debut P%d ==> pid = %d \n",i,getpid());

        for(i=2 ; i<=nbProcessus ; i++){
                pfils=fork();
                if(pfils==-1){
                        perror("fork a echoue");
                        exit(1);
                }
                if(pfils ==0) // fils
                        {printf("processus fils P%d ==> pid = %d
      et ppid = %d \n",i,getpid(),getppid());  break;}
                else // pere
{printf("---------->fin de l'iteraion i=%d du pere dont le pid
      =%d  \n",i,getpid());
}       }
      wait(NULL);
}
```

Resultat de l'execution :

```
 debut P1 ==> pid = 12347
---------->fin de l'iteraion i=2 du pere dont le pid =12347
processus fils P2 ==> pid = 12348 et ppid = 12347
---------->fin de l'iteraion i=3 du pere dont le pid =12347
processus fils P3 ==> pid = 12349 et ppid = 12347
---------->fin de l'iteraion i=4 du pere dont le pid =12347
processus fils P4 ==> pid = 12350 et ppid = 12347
---------->fin de l'iteraion i=5 du pere dont le pid =12347
processus fils P5 ==> pid = 12351 et ppid = 12347
---------->fin de l'iteraion i=6 du pere dont le pid =12347
processus fils P6 ==> pid = 12352 et ppid = 12347
---------->fin de l'iteraion i=7 du pere dont le pid =12347
processus fils P7 ==> pid = 12353 et ppid = 12347
---------->fin de l'iteraion i=8 du pere dont le pid =12347
processus fils P8 ==> pid = 12354 et ppid = 12347
```

On remarque que grâce à l'instruction wait, il y a un fils par itération, donc une synchronisation entre le père et les fils.

Action du fork : Exemple 3

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
int main(){
        int pfils, i=1, nbProcessus=8;
        printf("debut P%d ==> pid = %d \n",i,getpid());
        for(i=2 ; i<=nbProcessus ; i++){
                pfils=fork();
                if(pfils==-1){
                        perror("fork a echoue");
                        exit(1);
                }
                if(pfils ==0) // fils
                        {printf("processus fils P%d ==> pid = %d
        et ppid = %d \n",i,getpid(),getppid());}
                else // pere
{printf("---------->fin de l'iteraion i=%d du pere dont le pid
        =%d  \n",i,getpid());
        break;
}       }
        wait(NULL);
}
```

Resultat de l'execution :

```
debut P1 ==> pid = 12570
---------->fin de l'iteraion i=2 du pere dont le pid =12570
processus fils P2 ==> pid = 12571 et ppid = 12570
---------->fin de l'iteraion i=3 du pere dont le pid =12571
processus fils P3 ==> pid = 12572 et ppid = 12571
---------->fin de l'iteraion i=4 du pere dont le pid =12572
processus fils P4 ==> pid = 12573 et ppid = 12572
---------->fin de l'iteraion i=5 du pere dont le pid =12573
processus fils P5 ==> pid = 12574 et ppid = 12573
---------->fin de l'iteraion i=6 du pere dont le pid =12574
processus fils P6 ==> pid = 12575 et ppid = 12574
---------->fin de l'iteraion i=7 du pere dont le pid =12575
processus fils P7 ==> pid = 12576 et ppid = 12575
---------->fin de l'iteraion i=8 du pere dont le pid =12576
processus fils P8 ==> pid = 12577 et ppid = 12576
```

On remarque que :

p1 → p2 est le fils de p1 → p3 fils de p2→ p4 fils de p3→ p5 fils de p4 → p6 fils de p5→ p7 fils de p6 → et p8 est le fils de p7 (tp1 exercice 2a)

La primitive wait et la terminaison d'un fils

- La primitive `wait (int * status)` suspend l'exécution du processus appelant jusqu'à ce qu'un de ses processus fils se termine
- Si un processus fils est déjà terminé, la primitive retourne le résultat immédiatement
- La primitive retourne le pid du processus fils si le retour est déjà la terminaison d'un processus fils et -1 en cas d'erreur

Autres primitives de synchronisation père/fils

- WIFEXITED : vrai si le processus s'est terminé normalement (non interrompu)
- WEXITSTATUS : fournit le code de retour du processus s'il est terminé normalement
- WIFSIGNALED : vrai si le processus fils s'est terminé suite à la réception d'un signal
- WTERMSIG : fournit le numéro du signal ayant provoqué la terminaison du processus

Autres primitives de synchronisation père/fils : Exemple

```
// tp1 Exercice 5
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
main (void)
{ pid_t pid ;  int status,i ;
for (i=0 ; i < 4 ; i++) {
switch (fork ()) {
case -1 : perror (" fork ") ;   exit (1) ;
case 0 : /* le fils */
  printf("  Processus %d fils de %d \n ",getpid(),getppid());
   exit (i+1); /* les entiers de 1 à 4 sont les codes de retour
       de exit */
default : /* le pere */
  printf (" pere a cree processus %d\n ", getpid()) ;
  pid=wait (&status) ;
  if (WIFEXITED (status))
   printf (" fils (%d) envoi l'entier : status = %d, le pere
       l'affiche \n ",pid, WEXITSTATUS(status)                ) ;
           }
}
}
```

Résultat de l'execution :

```
pere a cree processus 24771
   Processus 24772 fils de 24771
   fils 24772 envoi l'entier : status = 1, le pere l'affiche
pere a cree processus 24771
   Processus 24773 fils de 24771
   fils 24773 envoi l'entier : status = 2, le pere l'affiche
pere a cree processus 24771
   Processus 24774 fils de 24771
   fils 24774 envoi l'entier : status = 3, le pere l'affiche
pere a cree processus 24771
   Processus 24775 fils de 24771
   fils 24775 envoi l'entier : status = 4, le pere l'affiche
```

On remarque que : La valeur de la variable status est transmis du fils au père via exit(i +1) et via la primitive WEXITSTATUS(status)

Recouvrement de processus

- Consiste à changer le code d'un processus
- Transforme le processus appelant en un nouveau processus qui exécutera un nouveau programme
- Le nouveau programme se substitue à l'ancien et il n'y a pas de retour vers le processus appelant sauf en cas d'erreur
- L'identité du processus appelant est conservée, il y a simplement remplacement de son code exécutable et ses données par ceux d'un autre programme

Primitives du recouvrement

- `int execl (const char *path, const char *arg,….);`
- `int execlp (const char *file, const char *arg,….);`
- `int execle (const char *path, const char *arg, …., char * const envp[]);`
- `int execv (const char * path, char * argv[]);`
- `int execvp (const char *file, char * const argv[]);`
- `int execve (const char *path, char *const argv[], char *const envp[]);`

Signification des arguments

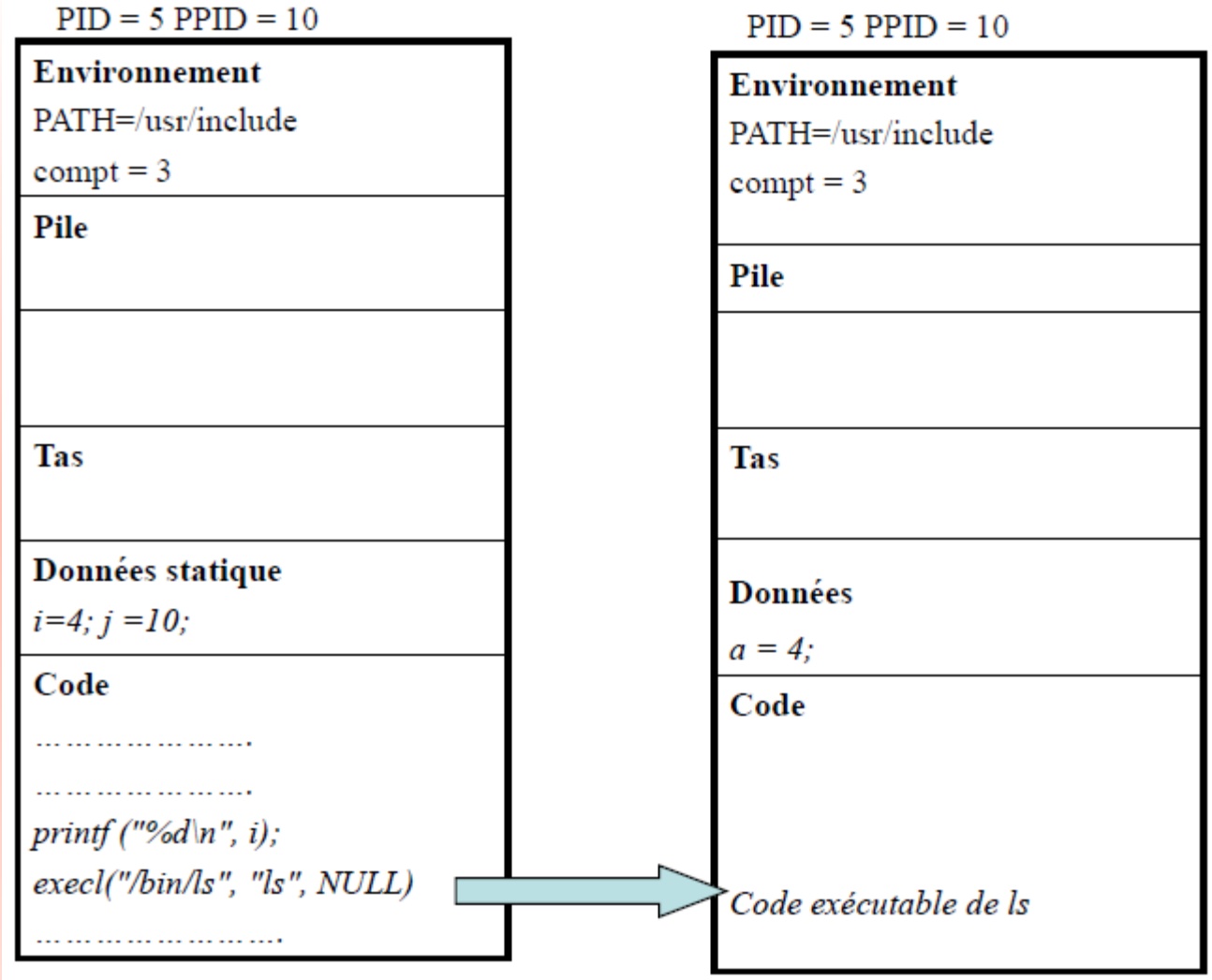
```
Exec?? (arg1, arg2, arg3 );
```

- Le premier correspond à un programme à exécuter
- Le second correspond aux arguments du programme à exécuter
  - soit sous forme de liste de pointeurs sur chaînes de caractères
  - soit sous forme de tableau
  - Le premier élément de la liste (ou du tableau) est le nom du fichier à exécuter et le dernier est un pointeur NULL
- Le troisième argument éventuel est une liste (ou un tableau) de pointeur d'environnement.

Recouvrement de processus : Décodage des noms des primitives

- v inclus dans le nom : les arguments sont passés sous forme de tableau
- l inclus dans le nom : les arguments sont passés sous la forme de liste
- p inclus dans le nom : le fichier à exécuter est recherché dans les répertoires du PATH
- p non inclus dans le nom : le fichier a exécuter est recherché au niveau du répertoire courant
- e inclus dans le nom : un nouvel environnement est transmis au nouveau processus
- e non inclus dans le nom : l'environnement du nouveau processus est déterminé à partir de la variable environ

Le principe de exec



Exemple

Ecrire un programme qui appelle le compilateur cc pour compiler un fichier (prog.c) et génère un fichier exécutable (prog.exe). Les noms du fichier à compiler et de l'exécutable sont passés en paramètres au programme

Donner :

–Une version avec execl

–Une version avec execlp

–Une version avec execv

Version avec execl

```
int main(int argc, char *argv[]){
  /* vérification du nombre de paramètres */
  if (argc != 3)  { fprintf (stderr, "Usage : %s fichier.c fichier.exe \n", argv[0]);
                    exit(1);
                  }
  /* appel de execl avec le chemin absolu de la commande cc */
  execl("/usr/bin/cc",  "cc",  argv[1], "-o",  argv[2],  NULL);
    perror("execl");
  exit (2);
}
```

Version avec execlp

```
Int main(int argc, char *argv[]) {
   /* vérification du nombre de paramètres */
   if (argc != 3)  { fprintf (stderr, "Usage : %s fichier.c fichier.exe \n", argv[0]);
                     exit(1);
                   }
  /* appel de execlp avec le nom de la commande cc */
  execl("cc",  "cc",  argv[1],"-o",  argv[2],  NULL);
  perror("execlp");
  exit (2);
}
```

Version avec execv

```
int main(int argc, char *argv[]) {
    char * args[5];
    /* vérification du nombre de paramètres */
   if (argc != 3)  {
                fprintf (stderr, "Usage : %s fichier.c fichier.exe \n", argv[0]);
                exit(1);
                    }
            args[0]="cc"; args[1]=argv[1]; args[2]="-o";
            args[3]=argv[2]; args[4]=NULL;
            execv("cc", agrs);
            perror("execv");
            exit (2);
}
```

Fork + exec

- exec réinitialise un processus à partir d'un programme déterminé, le programme change et le processus demeure
- A l'exception du noyau UNIX, exec est la seule manière d'exécution des programmes sous UNIX, et la fonction fork est la seule manière de créer un nouveau processus
- L'utilisation simultanée de fork et de exec permet d'exécuter un nouveau programme tout en conservant le processus appelant suivant le mécanisme suivant
  - Le processus père appelle fork pour lancer un processus fils
  - Le processus fils exécute le nouveau programme grâce à exec
  - Et le processus père se substitue au fils
- Après duplication, le fils se charge de l'exécution d'un programme différent de celui de son père grâce aux primitives exec

Exemple : fork + exec

Le programme suivant crée un processus fils.

Le fils exécute la commande date. Le père attend la terminaison du fils et teste si oui ou non s'est terminé normalement.

```
int main ( )
{ int status, pid, pid1;switch (pid = fork ( )){
case -1 : printf("le fork n'a pas marché \n");exit(1);
case 0 : printf(" je suis le fils du processus %d\n",
   getppid());printf(" mon pid est %d\n", getpid());
   execl("/bin/date","date",0);  /*code exécuté par le */ perror("erreur
   sur exec");
default : pid1= wait (&status); *code exécuté par le père*/
   if (WIFEXITED(status)) printf (" le processus fils %d s'est terminé
   normalement avec code de retour du fils %d \n", pid1,
   WEXITSTATUS(status));
}}
```