

Gestion des processus système

Samia Bouzefrane

CNAM

`Samia.bouzefrane@cnam.fr`

`http://cedric.cnam.fr/~bouzefra`

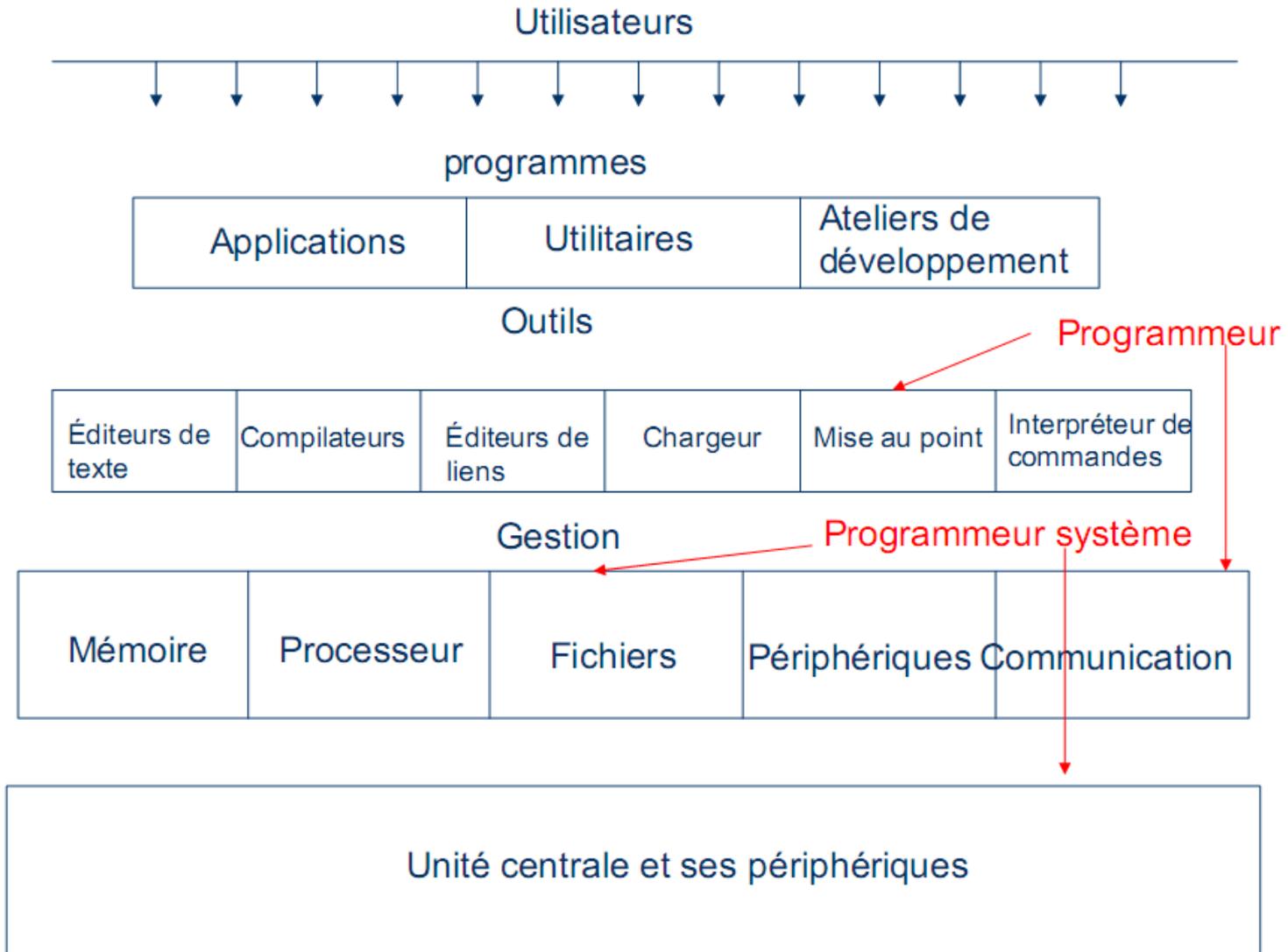
Introduction

Définition d'un système informatique

➤ *Un système informatique*

- Matériel
 - Unité centrale, disques, réseau, périphériques
- Système d'exploitation
 - Gestion utilisateurs, SGF, ressources (accès, partage, distribution, échange...)
- Programmes d'application
- Utilisateurs
 - Login, droits, accès

Structure générale d'un système informatique



Évolution des systèmes d'exploitation

- Batch
- Multiprogrammé
- Multi-utilisateurs centralisé
 - Temps partagé, transactionnel
- Distribué / Parallèle
- Temps réel
- Embarqué/connecté

Éléments d'un système d'exploitation (SE)

Matériel	Services du SE	Abstraction utilisateur
Processeur	Gestion de processus – ordonnancement – gestion de signaux - synchronisation	processus
Mémoire	Gestion mémoire	Espace d'adressage
Entrée/sortie (disque)	Gestion de fichiers	Fichiers

Gestion des processus

Applications comportant des processus concurrents

- Motivations pour les processus concurrents
 - améliorer l'utilisation de la machine (cas monoprocesseur)
 - multiprogrammation sur des serveurs
 - parallélisme entre le calcul et les E/S sur PC
 - lancer des sous-calculs dans une machine parallèle ou répartie
 - créer des processus réactifs pour répondre à des requêtes ou à des événements externes (systèmes temps réel, systèmes embarqués et/mobiles)
- Construire des applications concurrentes, parallèles ou réparties

Définitions (1)

- **Processus:**
 - Instance d'un programme en exécution
 - et un ensemble de données
 - et un ensemble d'informations (BCP, bloc de contrôle de processus)
- **Deux types de processus**
 - **Système:** lancés par le système d'exploitation (démons) ou le super-utilisateur
 - **Utilisateur:** lancés par les utilisateurs
- **Code de retour d'un processus**
 - **=0:** fin normale
 - **≠0:** comportement anormal (en cours d'exécution ou à la terminaison)

Définitions (2)

- La table des descripteurs de fichiers:
 - Chaque processus peut posséder au maximum OPEN_MAX descripteurs de fichier
 - En plus des descripteurs qu'un processus hérite de son père, il peut en créer d'autres avec open, dup, ...
- Enchaînement d'exécution des processus
 - Si les commandes associées sont séparées par " ; "
 - Exemple: `commande1 ; commande2 ; ... ; commanden`
 - Remarque:
 - Une erreur dans un processus n'affecte pas les suivants
- Processus coopérants et communication par tube (pipe)
 - Communication par l'intermédiaire de tube. Les résultats d'un processus servent de données d'entrée à l'autre
 - Exemple: `commande1 | commande2 | ... | commanden`

Descripteur des processus Linux

- Bloc de contrôle (utilisé par Linux) comporte :
 - État du processus
 - Priorité du processus
 - Signaux en attente
 - Signaux masqués
 - Pointeur sur le processus suivant dans la liste des processus prêts
 - Numéro du processus
 - Numéro du groupe contenant le processus
 - Pointeur sur le processus père
 - Numéro de la session contenant le processus
 - Identificateur de l'utilisateur réel
 - Identificateur de l'utilisateur effectif
 - Politique d'ordonnancement utilisé
 - Temps processeur consommé en mode noyau et en mode utilisateur
 - Date de création du processus, etc.

Visualisation de processus (1)

- La commande « **ps** » permet de visualiser les processus existant à son lancement sur la machine
- Sans option, elle ne concerne que les processus associés au terminal depuis lequel elle est lancée

```
$ ps -cflg
```

```
  F S UID      PID   PPID  PGID   SID  CLS  PRI  ADDR      SZ  WCHAN  STIME TTY      TIME CMD
000 S invite  8389  8325  8389  8389   -   25   -       749  wait4  10:15 pts/2   00:00:00 /bin/bash
000 R invite  11364 8389  11364 8389   -   25   -       789  -      11:22 pts/2   00:00:00 ps -cflj
```

```
$
```

```
$ 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 R   501  11370  8389  0  79   0   -       788  -      pts/2    00:00:00 ps
```

```
$
```

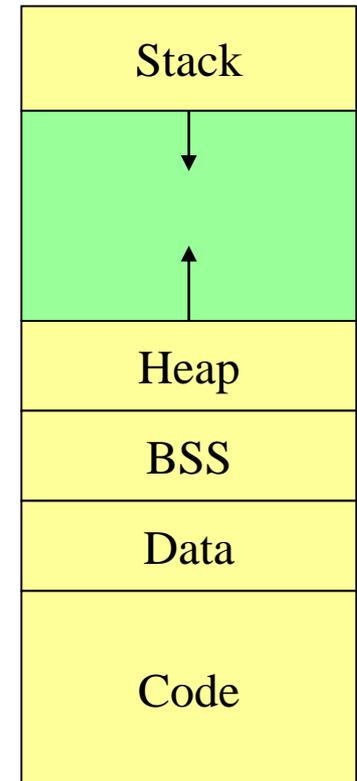
Informations de la commande *ps*

Nom	Interprétation	Option(s)
S	Etat du processus S: endormi, R: actif, T: stoppé, Z: terminé (zombi)	-l
F	Indicateur de l'état du processus en mémoire (swappé, en mémoire, ...)	-l
PPID	Identificateur du processus père	-l -f
UID	Utilisateur propriétaire	-l -f
PGID	N° du groupe de processus	-j
SID	N° de session	-j
ADDR	Adresse du processus	-l
SZ	Taille du processus (en nombre de pages)	-l
WCHAN	Adresse d'événements attendus (raison de sa mise en sommeil s'il est endormi)	-l
STIME	Date de création	-l
CLS	Classe de priorité (TS → temps partagé, SYS → système, RT → temps réel).	-cf -cl
C	Utilisation du processeur par le processus	-f -l
PRI	Priorité du processus	-l
NI	Valeur « nice »	-l

Processus en mémoire (1)

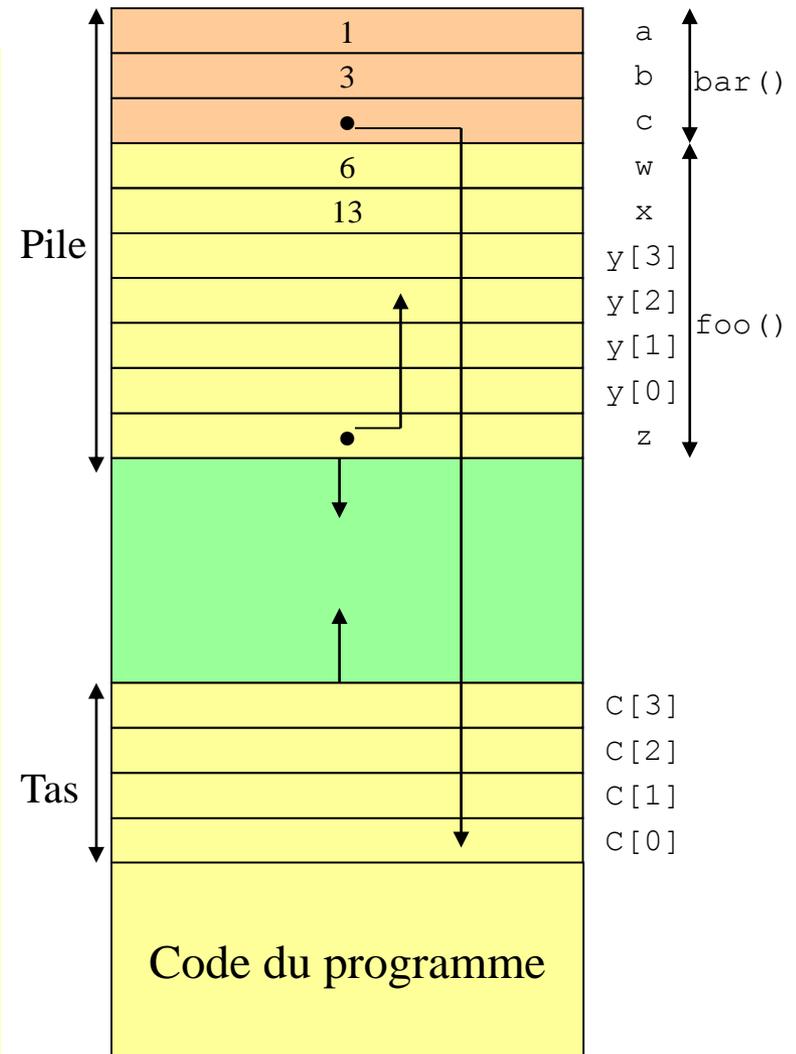
- Pile (stack):
 - Les appels de fonctions et variables locales aux fonctions exigent une structure de donnée dynamique
 - Variables locales et paramètres
 - Adresse de retour de fonction
- Tas (heap)
 - Allocation dynamiques de variables
- Code (Text)
 - Code des fonctions
 - Lecture seulement (read-only)
- Données (Data+BSS)
 - Constantes
 - Variables initialisées / non-initialisées

Image d'un processus



Processus en mémoire (2)

```
void foo(int w) {  
    int x = w+7;  
    int y[4];  
    int *z = &y[2];  
}  
  
void bar(int a) {  
    int b = a+2;  
    int *c = (int *) malloc(sizeof(int)*4);  
    foo(b+3);  
}  
  
main() {  
    bar(1);  
}
```



Ordonnancement de processus

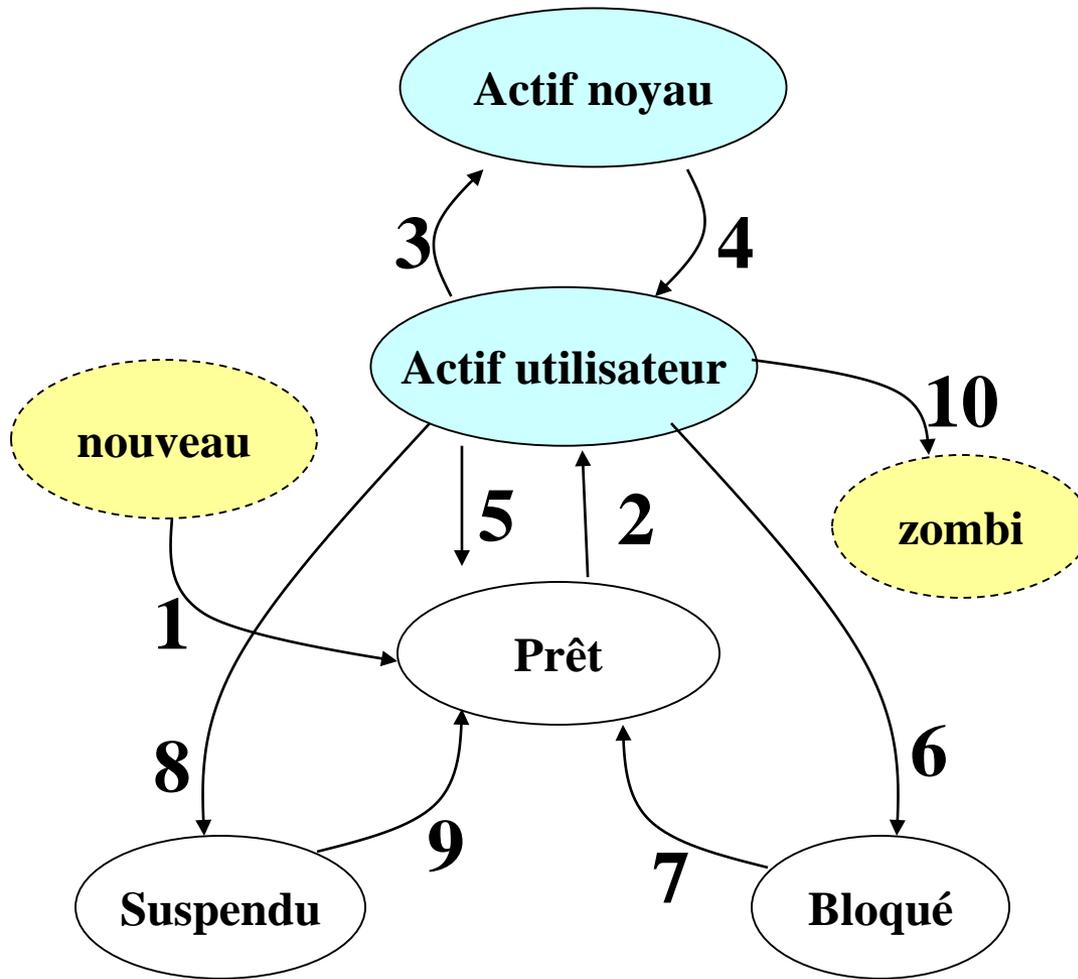
- L'ordonnanceur (scheduler) d'Unix:
 - Associe une priorité dynamique à chaque processus, recalculée périodiquement
 - Utilise la stratégie du tourniquet (round robin) pour les processus de même priorité
- La fonction de calcul de la priorité courante utilise les paramètres suivants:
 - Une priorité de base: correspondant au niveau d'interruption du processus
 - Le temps CPU consommé récemment
 - Une priorité ajustable par le processus lui-même grâce à:

```
#include <unistd.h>  
int nice(int incr);
```

qui permet d'augmenter la valeur de **incr** [0-39] et donc de diminuer sa priorité

- Le super-utilisateur peut augmenter la priorité en donnant une valeur <0 à **incr**.

État d'un processus



1. Allocation de ressources
2. Élu par l'ordonnanceur
3. Le processus revient d'un appel système ou d'une interruption
4. Le processus a fait un appel système ou a été interrompu
5. A consommé son quantum de temps
6. Se met en attente d'un événement
7. Événement attendu est arrivé
8. Suspendu par un signal SIGSTOP
9. Réveil par le signal SIGCONT
10. fin

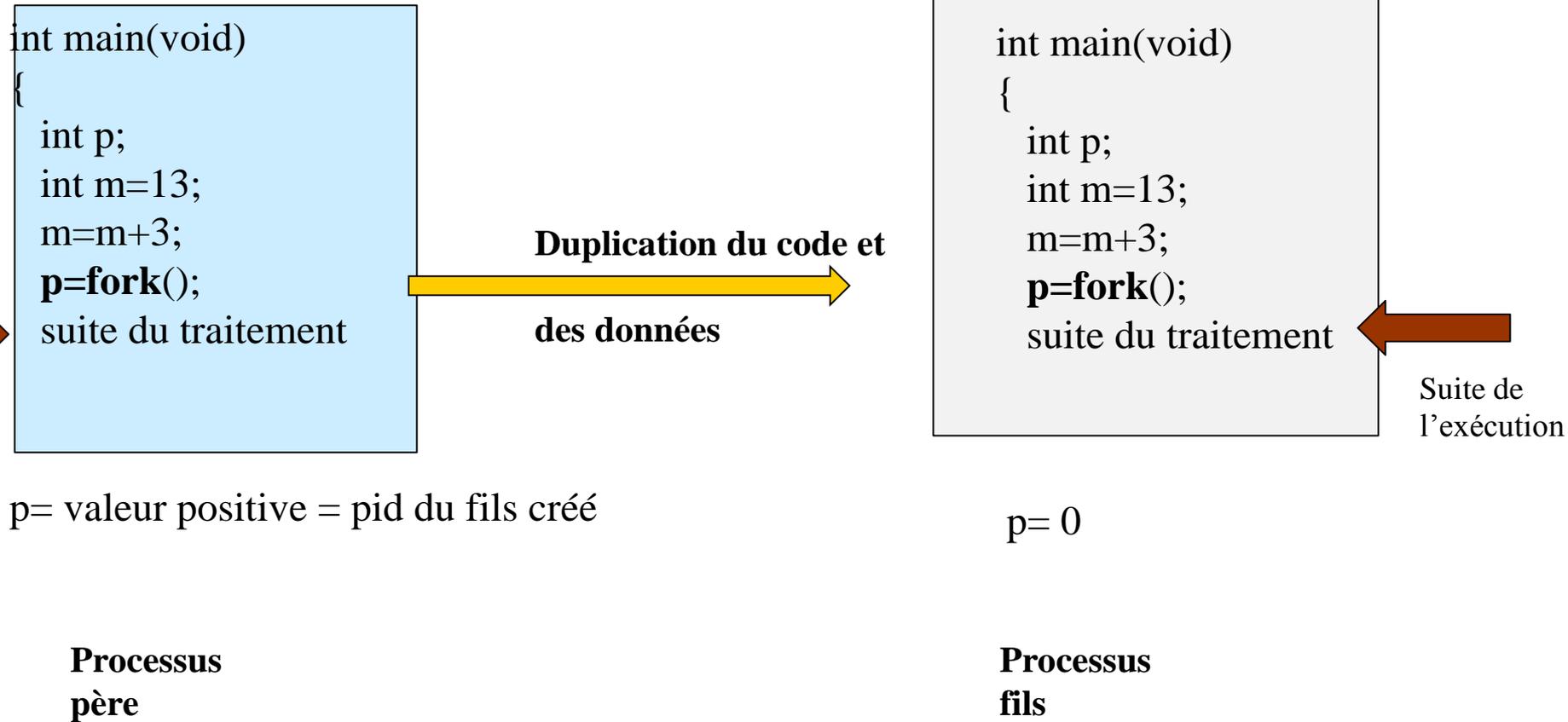
Création de processus (1)

- La primitive *fork()* permet de créer un nouveau processus (fils) par duplication
- Afin de distinguer le père du fils, *fork()* retourne:
 - -1 : en cas d'erreur de création
 - 0 : indique le fils
 - > 0 : le **pid** du processus fils au processus père
- Les primitives *getpid()* et *getppid()* permettent d'identifier un processus et son père.

```
#include <unistd.h>
pid_t fork(void) // crée un nouveau processus
pid_t getpid(void) // donne le pid du processus
pid_t getppid(void) // donne le pid du père du processus
```

Création de processus (2)

En Mémoire Centrale



Exemple 1 - Création d'un processus

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main(void)
{
    int p;

    p=fork();

    if (p==0) printf("je suis le processus fils\n");
    if (p>0) printf("je suis le processus père\n");

    return 0;
}
```



```
je suis le processus père
je suis le processus fils
```

Exemple 2 - Création d'un processus (1)

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main(void) {
    int p;

    p = fork();

    if (p > 0)
        printf("processus père: %d-%d-%d\n", p, getpid(), getppid());

    if (p == 0) {
        printf("processus fils: %d-%d-%d\n", p, getpid(), getppid());
    }

    if (p < 0)
        printf("Probleme de creation par fork()\n");

    system("ps -l");
    return 0;
}
```

Exemple 2 - Création d'un processus (2)

F	S	UID	PID	PPID	C	PRI	NI	ADDR	SZ	WCHAN	TTY	TIME	CMD
000	S	501	2402	2401	0	71	0	-	748	wait4	pts/1	00:00:00	bash
000	S	501	13681	2402	0	71	0	-	343	wait4	pts/1	00:00:00	exo2
040	S	501	13682	13681	0	71	0	-	343	wait4	pts/1	00:00:00	exo2
000	R	501	13683	13681	0	74	0	-	786	-	pts/1	00:00:00	ps
000	R	501	13684	13682	0	74	0	-	789	-	pts/1	00:00:00	ps

processus fils: 0-**13682**-**13681**

F	S	UID	PID	PPID	C	PRI	NI	ADDR	SZ	WCHAN	TTY	TIME	CMD
000	S	501	2402	2401	0	71	0	-	748	wait4	pts/1	00:00:00	bash
000	S	501	13681	2402	0	71	0	-	343	wait4	pts/1	00:00:00	exo2
044	Z	501	13682	13681	0	65	0	-	0	do_exi	pts/1	00:00:00	exo2 <def>
000	R	501	13683	13681	0	75	0	-	788	-	pts/1	00:00:00	ps

processus père: **13682**-**13681**-2402

L'héritage du fils (1)

- 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

```
#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

Processus père

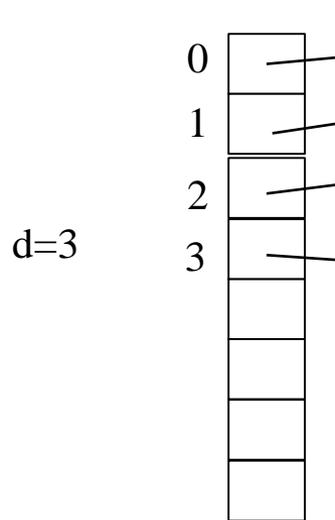


Table des descripteurs

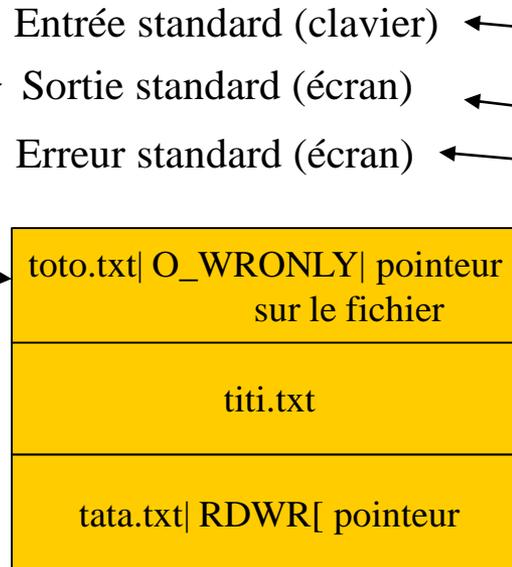


Table des fichiers ouverts

Processus fils

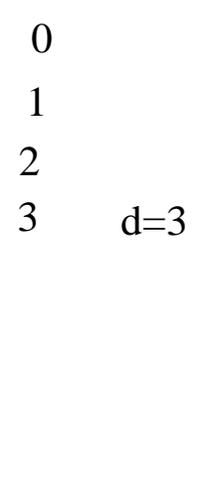


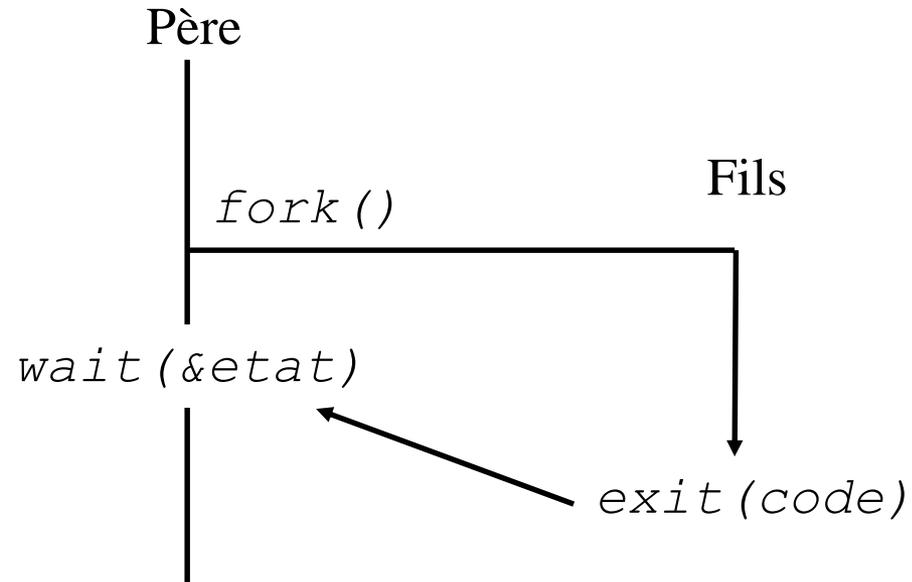
Table des descripteurs

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 zombies 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 (1)



Synchronisation père-fils (2)

etat

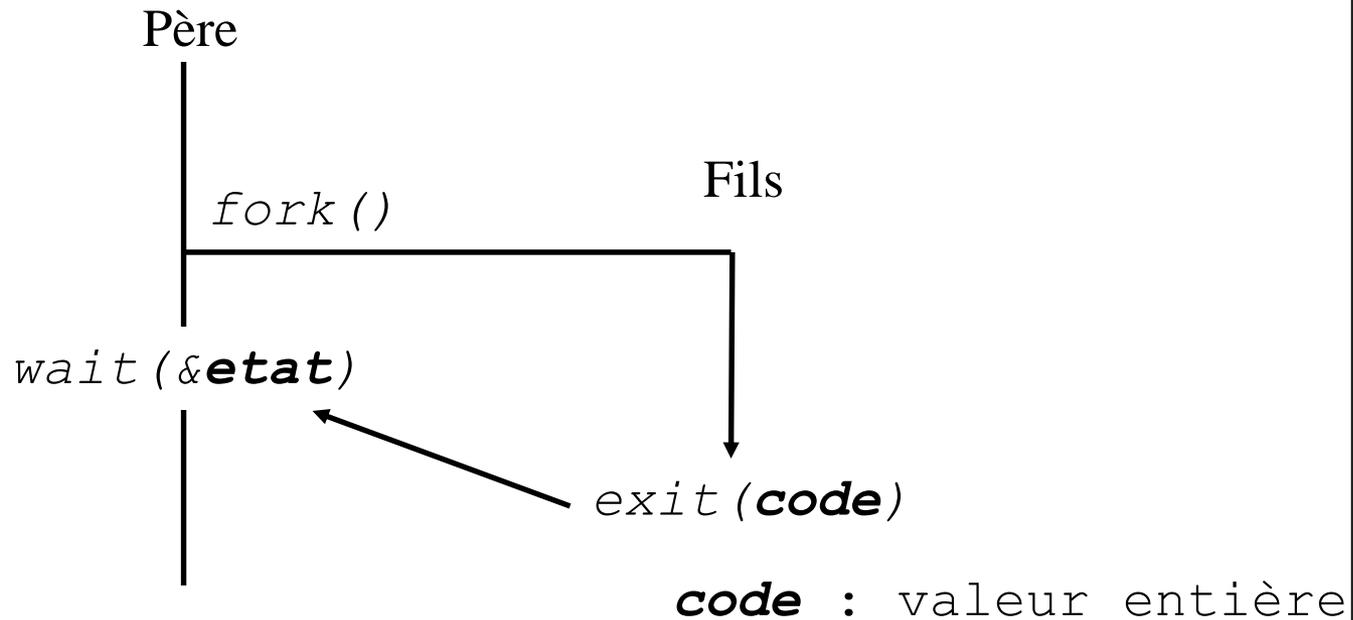


octet1

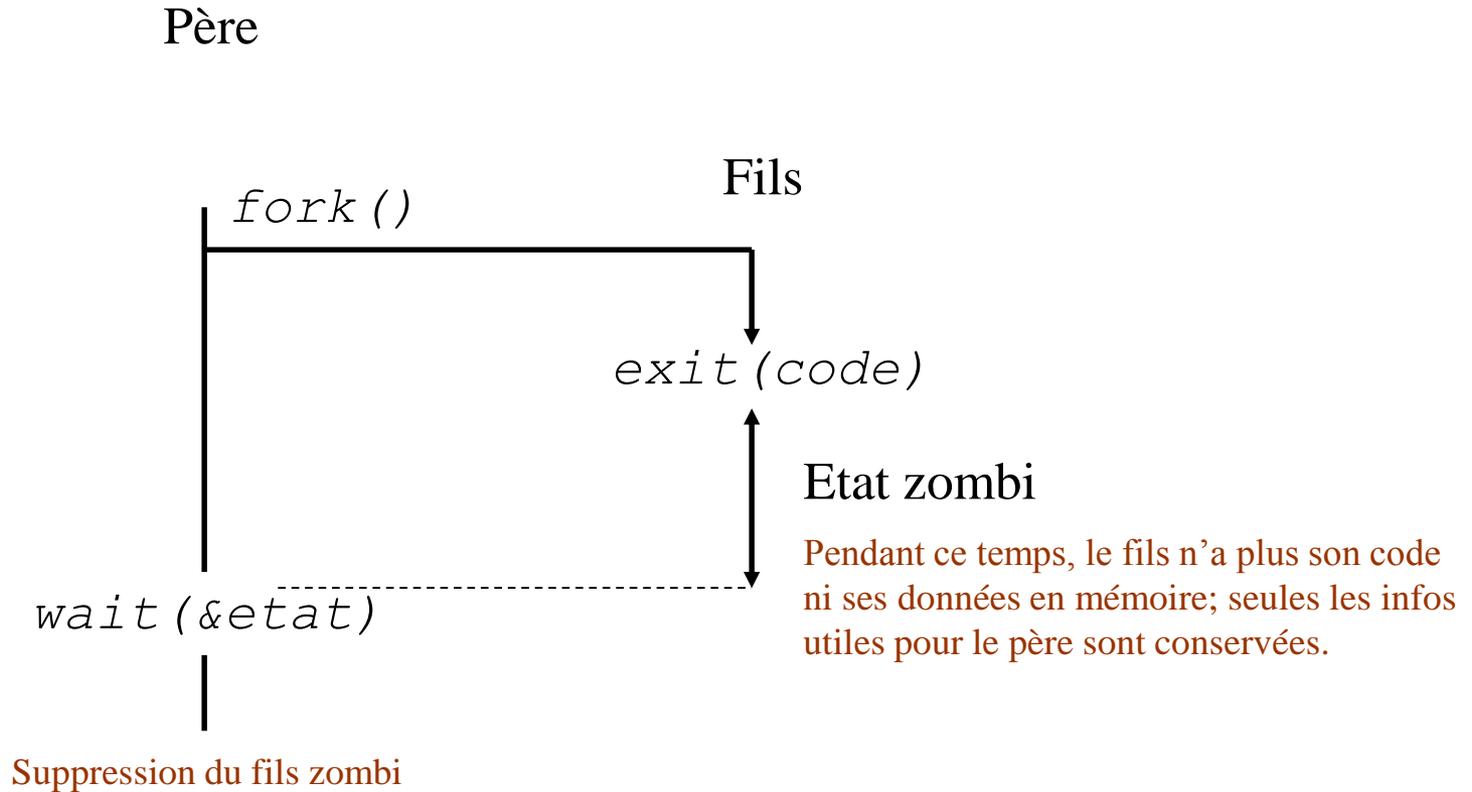
octet2 (Bits précisant si fin normale ou due à un signal)

Pour retrouver le code de retour à partir de *etat*, il suffit de décaler *etat* de 8 bits

```
int etat;
```



Synchronisation père-fils (3)



Synchronisation père-fils (4)

```
#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

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>")

Communication avec l'environnement

- `int main(int argc, char *argv[], char *envp[])`
- 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 ligne 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.

Ex. `$./prog 12 lapin`

Commande: `./prog / argc=3 / argv[0]="./prog" argv[1]="12", argv[2]="lapin"`

Communication avec l'environnement: exemple (1)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

char *chaine1;

void main(int argc, char *argv[], char *envp[]) {
    int k; int a, b, somme=0;
    if (argc<3) {printf("erreur d'argument - argc=%d\n");
        exit(0);
    }
    printf("\naffichage des variables d'environnement:\n");
    for (k=0; envp[k]!=NULL; k++) {
        printf("%d: %s\n", k, envp[k]);
    }
    printf("\naffichage d'une variable d'environnement interceptee:\n");
    chaine1 = getenv("SHELL");
    printf("SHELL = %s\n", chaine1);

    a=atoi(argv[1]);  b=atoi(argv[2]);
    somme=a+b;
    printf("somme est : %d\n", somme);

}
```

Communication avec l'environnement: exemple (2)

```
$ ./exo2 12 45
```

```
affichage des variables d'environnement:
```

```
0: USERDOMAIN_ROAMINGPROFILE=samia
```

```
1: HOMEPATH=\Users\bouzefrane
```

```
...
```

```
83: TERMNUM=2
```

```
84: COMPUTERNAME=SAMIA
```

```
85: _=./exo2
```

```
affichage d'une variable d'environnement interceptee:
```

```
SHELL = /bin/bash.exe
```

```
somme est : 57
```

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*): `arg0`, `arg1`, ..., `argN`, `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 (1)

```
#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 (2)

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int res;

int main(void) {
    int p;
    p=fork();
    if (p==0) {
        execl("./exo2", "exo2", "12", "34", NULL);
    }
    wait(NULL);
    return 0;
}
```

```
$ gcc exo3.c -o exo3
./exo3
...
82: SHELL=/bin/bash.exe
83: LOGNAME=samia
84: COMPUTERNAME=SAMIA
```

```
affichage d'une variable d'environnement interceptee:
SHELL = /bin/bash.exe
somme est : 46
```

Bibliographie

