Chapitre 2
Communication Interprocessus

Les processus

Abstraction d'une exécution

- C'est l'unité d'exécution visible par le S.E.
- C'est l'unité ordonnancée par le S.E. (et rien d'autre).
- Il contient le contexte d'exécution d'un programme
  - Espace d'adressage mémoire,
  - Pointeur d'instruction,
  - Pointeur de pile
  - Autre ressources système : les fichiers ouverts, les connections réseaux ouvertes

Appelé Job, Tâche, processus

Les processus

- 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: propriété du super-utilisateur (démons)
  - 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)
- Interruption d'un processus à la réception d'un signal
  - A partir d'un programme (SIGINT, SIGQUIT)
  - A partir d'un shell avec la commande:

```
kill num_signal pid_processus
```

Différence entre programme et processus

Programme

- Entité statique décrivant un traitement ;
- Code situé sur disque (en langage source, en langage machine)
- Un programme peut donner lieu à plusieurs processus possibles par exemple : un même programme exécuté avec des données différentes

Processus

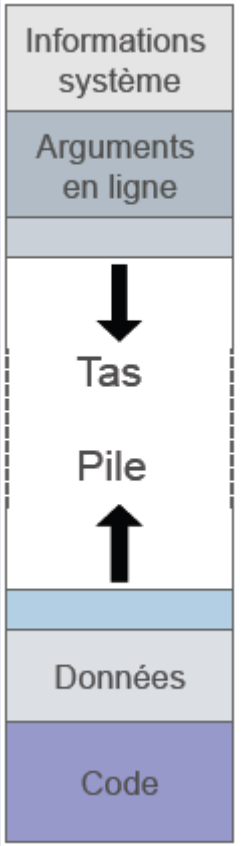
- Entité dynamique réalisant un traitement ;
- Code situé en mémoire centrale (en langage machine) ;
- Un processus peut mettre en jeu plusieurs programmes par exemple : un programme se terminant avec le lancement d'un autre programme (recouvrement).

Image

Ensemble des composants d'une image

Un programme en cours d'exécution manipule (met en jeu)

- Code
- Données
  - Statiques
  - Tas
  - Pile
- Contexte d'exécution
  - Pointeur d'instruction
  - Registres mémoire
  - Fichiers ouverts
  - Répertoire courant
  - Priorité



Image

•La mémoire d'un programme est divisée en les parties suivantes :
- Segment de données (données + BSS + tas binaire (heap en anglais)) ;
- Pile d'exécution, souvent abrégée en la pile (stack en anglais) ;
- Segment de code.

Le segment de données contient les variables globales et statiques utilisées par le programme et qui sont initialisées.

Le segment BSS aussi connu comme zone de données non initialisées commence à la fin du segment de données et contient toutes les variables globales et toutes les variables statiques qui sont initialisées à zéro ou n'ont pas d'initialisation explicite dans le code source. Par exemple, une variable déclarée static int i; sera « contenue » dans le segment BSS. le segment *bss se résume alors aux* variables locales de la fonction main().

tas : allocation dynamique (malloc, calloc, realloc)

Formellement

Définition

Image : Un ensemble d'objets qui peuvent donner lieu à une exécution (un code exécutable).

Définition

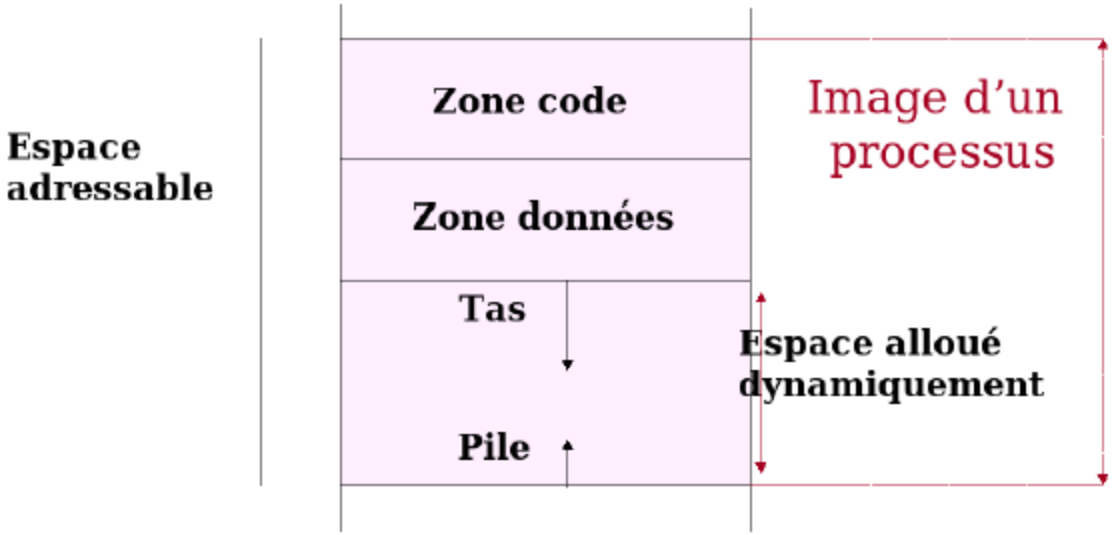
Processus : Exécution d'une image.

Illustration

Un programme (code) qui ajoute +1 à des entiers (données) sauvegardées dans un fichier (contexte). L'instance en train de s'exécuter (suivi des instructions dans l'environnement) est le processus.

Formellement

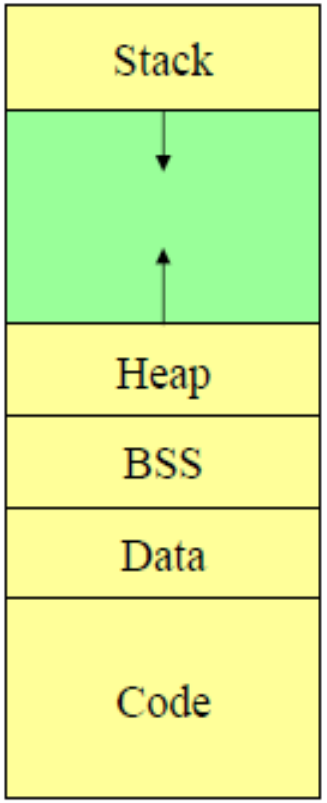
zone utilisateur



Processus en mémoire

- 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

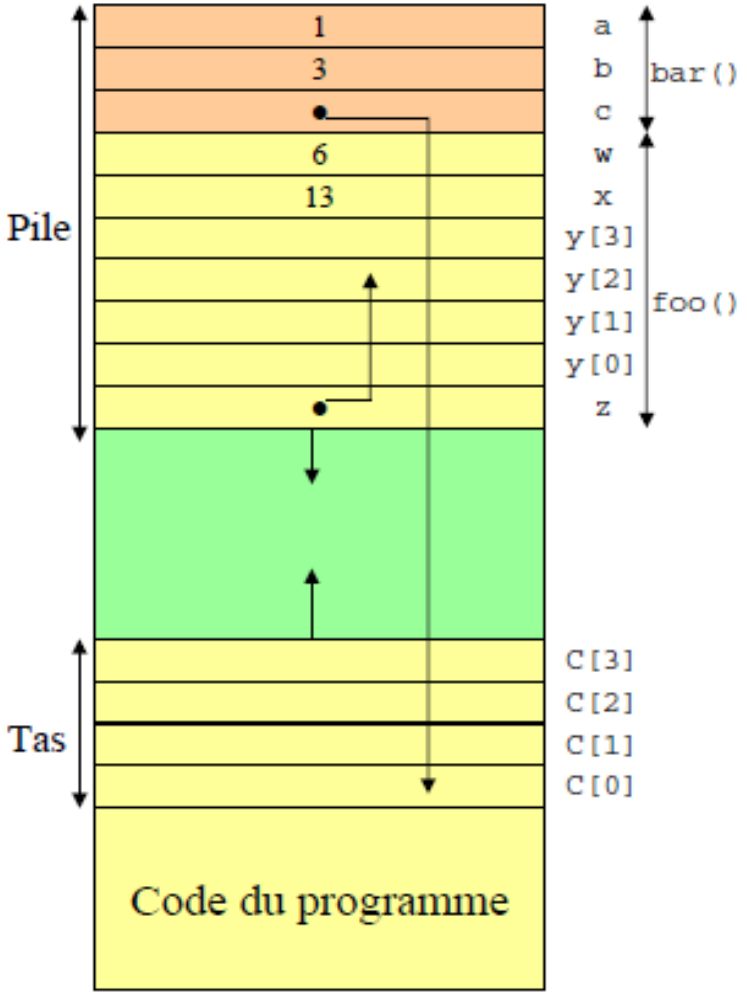


Processus en mémoire

```
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 *) calloc(sizeof(int)*4);
    foo(b+3);
}

main() {
    bar(1);
}
```



Les processus

Identification : le PID

- Identification de processus : le PID (process id)
- `pid_t getpid(void);` Appel système retournant le PID du processus
- visibles par les commandes top, ps
- Nom de l'entrée dans le dossier /proc

Affiliation

- tout processus à un père : processus qui l'a lancé
- `pid_t getppid(void);` Appel système retournant le PID du processus père
- L'ancêtre $pid = 1$, le processus `init`, lancé au boot
- les orphelins (processus dont le père est mort) sont le plus souvent récupérés par init

Propriétaire d'un processus

Tout processus à un propriétaire (l'utilisateur qui a lancé le processus)

- le processus possède les même droit sur les fichiers que le propriétaire.
- Seul le propriétaire d'un processus peut le tuer.

Réel-effectif

- propriétaire réel : celui qui exécute la commande (idem pour groupe), le plus souvent celui qui a lancé la commande.
  Donnés par UID et GID obtenus avec `getuid()`, `getgid()`.
- le propriétaire effectif (respectivement groupe effectif) celui à qui appartient l'exécutable.
  Donnés par EUID et EGID obtenus avec `geteuid()`, `getegid()`.

Exemple classique : l'exécutable `passwd` exécuté par n'importe qui mais possédé par root.

Etats d'un processus

Sortie d'une commande ps

D	sommeil ininterruptible
R	Actif ou prêt (dans la file)
S	Sommeil interruptible (attente d'un événement)
T	Stoppé (par un signal)
X	Mort
Z	Deficient ("zombie")
	processus, terminé mais données non recup par parent

Caractères additionnels

$<$	Processus ayant une très haute priorité sur les autres
N	Basse priorité
s	session leader
l	multi-threadé
$+$	En avant plan

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

Création de processus

Primitive `fork()`, clônage (Sous Unix)

- Primitive système : `fork()`
- Recopie totale (données, attributs) du processus père vers son processus fils (nouveau pid).
- Le fils continue son exécution à partir de cette primitive

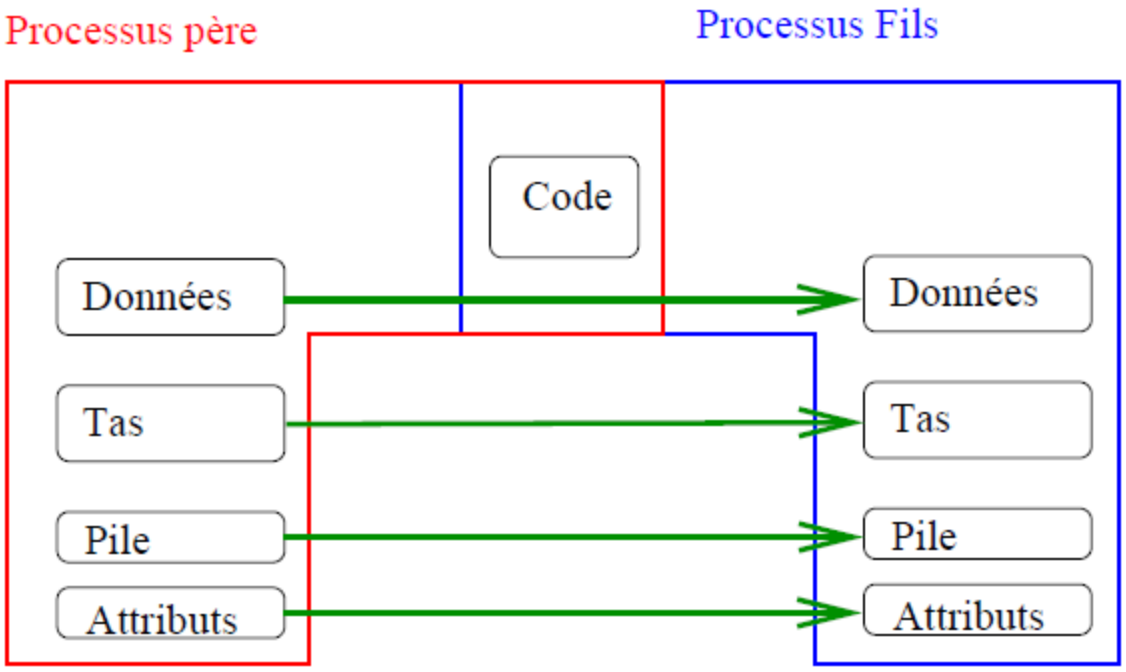
Code C : p1.c

```
#include<stdlib.h>
int main() {
 printf("Coucou %d\n",getpid());
 fork();
 printf("Hé %d\n",getpid());
}
```

Résultat

```
Machine:~/Moi> ./p1
Coucou 6874
Hé 6875
Hé 6874
Machine:~/Moi>
```

Création de processus
Clonage



Création de processus

Pour créer un processus, le système doit :

- Nommer le processus
- Créer un bloc de contrôle BCP
- Déterminer la priorité du processus
- Allouer des ressources au processus

Ensuite

- Recopie des données
- Recopie des attributs sauf
  - pid, ppid
  - signaux pendants
  - priorité
- Partage du code (si réentrance) sinon recopie du code

Création de processus Déroulement de l'exécution

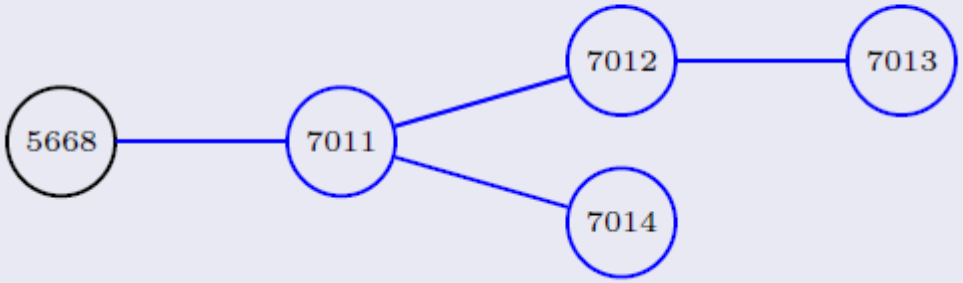
Code C : p2.c

```
#include<unistd.h>
int main() {
   printf("Coucou %d\n",getpid());
   fork();
   fork();
   printf("Hé %d:%d\n",getpid(),
                       getppid());
}
```

Résultat

```
Machine:~/Moi> ./p2
Coucou 7011
Hé 7013:7012
Hé 7012:7011
Hé 7014:7011
Hé 7011:5668
Machine:~/Moi>
```

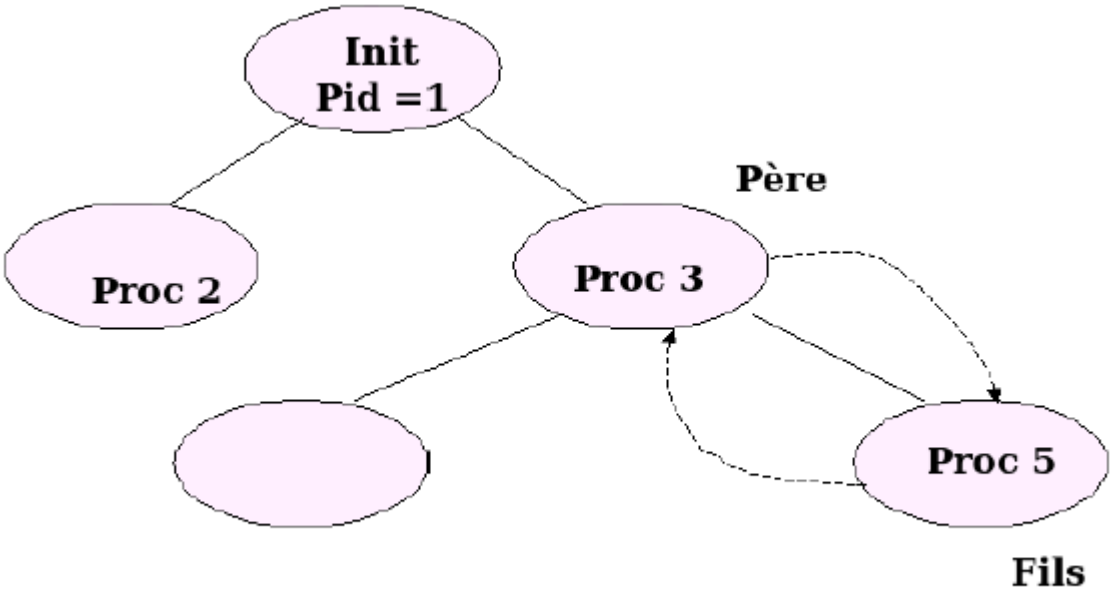
Hiérarchie



- 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.

Création de processus
Arborescence

L'itération de `fork()` conduit à une arborescence à partir du processus init (pid =1)



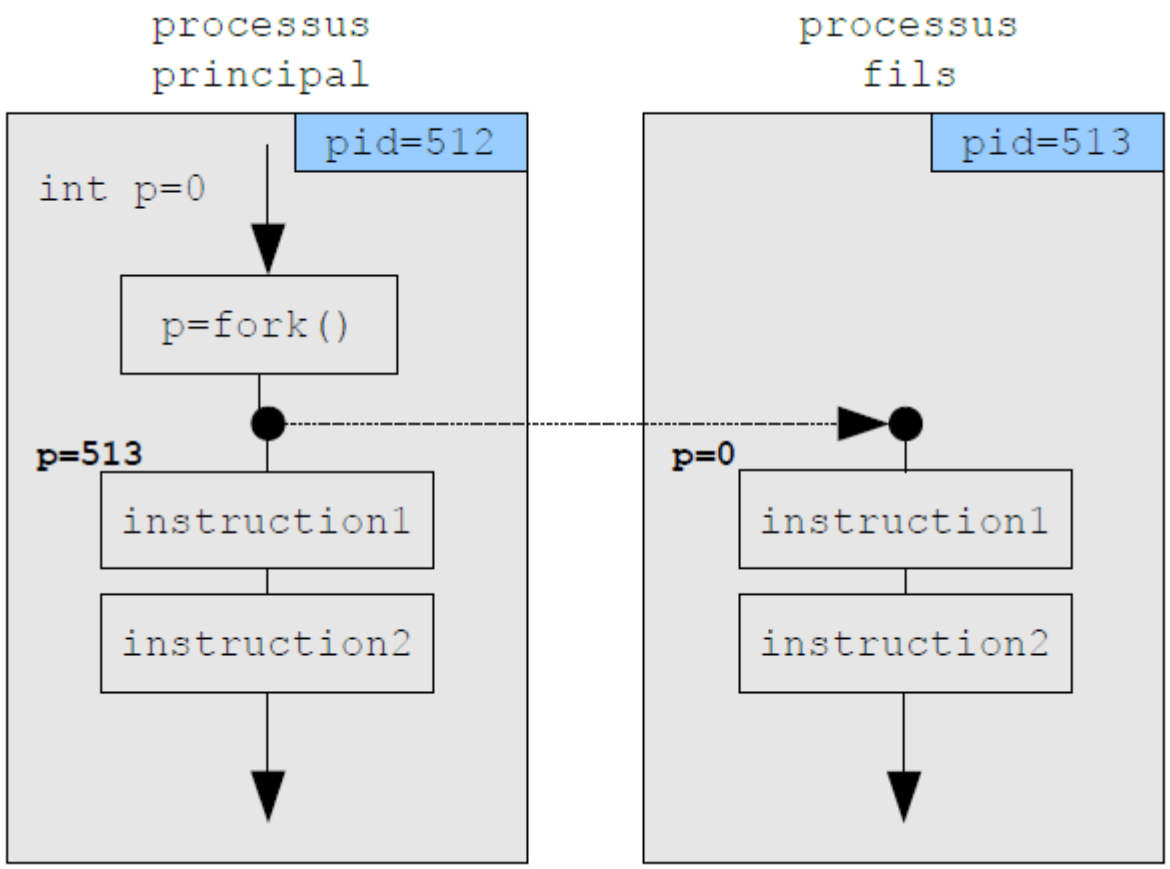
Création de processus
Différence fils-pére

Différencier le père du fils ?

- Code de retour du `fork()`
- Dans le père : le `fork` retourne le pid du processus fils
- Dans le fils : le `fork` retourne 0

Pourquoi ?

- Le fils peut connaître le pid de son père avec `getppid()`.
- Le code de retour du fork : seul moyen pour le père de connaître le pid du processus fils.


processus
principal
processus
fils
pid=512
pid=513
int p=0
p=fork()
p=513
p=0
instruction1
instruction2
instruction1
instruction2

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

int main(void){
   int pid;

   pid = fork();

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

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

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

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

Code C : p3.c

```
#include<unistd.h>
int main() { int r;
    printf("Coucou %d\n",getpid());
    if ( (r=fork())==0 )
        printf("Fils %d\n",getpid());
    else
        printf("Père %d\n",getpid());
    printf("Tous %d\n",getpid());
}
```

Résultat

```
Machine:~/Moi> ./p3
Coucou 7209
Père 7209
Tous 7209
Fils 7210
Tous 7210
Machine:~/Moi>
```

Portée du code : p4.c

```
int main(){
  int k ;
  printf(''Je suis seul au monde'\n');
  k=fork();
  if (k==-1) { printf("Erreur fork()"); exit(1);}
  if (k== 0) { /* code du fils */
      printf("Je suis le processus fils\n");
      exit(0);
  } else
      printf("Je suis le processus pere\n");
  printf("Et la qui suis-je %d\n",getpid());
}
```

Portée du code : Résultat

```
Machine:~/Moi> ./p4
Je suis seul au monde
Je suis le processus fils 6681
Je suis le processus pere 6680
Et la qui suis-je 6680
Machine:~/Moi>
```

Portée des variables : p5.c

```
int main(){
  int i,j,k ;
  i=5; j=2;
  if ((k=fork()) == -1) {printf("Erreur fork()");exit(1);}
  if (k== 0) { /* code du fils */
      printf("fils %d \n",getpid());
      j--;
  } else {
      printf("pere %d\n",getpid());
      i++;
  }
  printf("Pid:%d i:%d j:%d\n", getpid(),i,j);
}
```

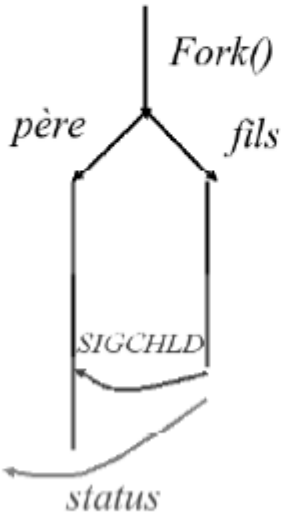
Portée des variables : Résultat de p6

```
Machine:~/Moi> ./p5
fils 6774
Pid:6774 i:5 j:1
pere 6773
Pid:6773 i:6 j:2
Machine:~/Moi>
```

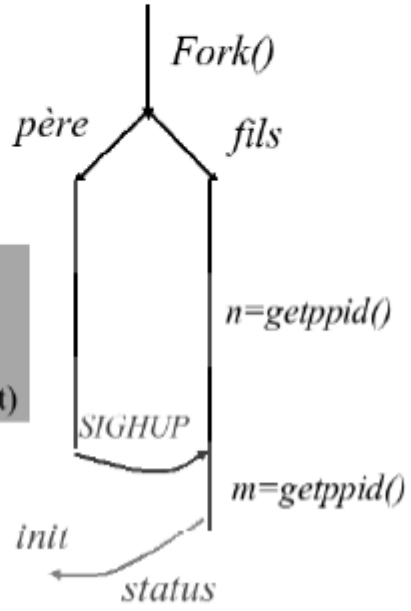
Indéterminisme du déroulement

Exécutions concurrentes

- Parallélisme d'exécution du père et du fils
- Concurrence pour l'accès aux ressources
- Ordre du déroulement (et des fins) est impossible à contrôler



Père se termine
avant le fils
n= pid du père
m=1 (pid de l 'init)



Synchronisation

- Père toujours prévenu de la fin d'un fils
- Fils toujours prévenu fin du père
- mais il faut que le père soit en attente
- Si la fin d'un fils n'est pas traitée par le père ce processus devient un processus zombie.

Synchronisation pour les fins d'exécution

- Possibilités d'échanges d'informations permettant une synchronisation sur les fins d'exécutions.
- Éviter les processus *zombies* (fin propre).
- Nécessite que le père soit en attente.
- Information de la fin du processus fils.

Appels système wait et exit

`exit` et code de retour

`exit (int)`

- valeur du `int` est "transmise" au père : code de retour
- Fin du processus fils après `exit`
- Par convention (défaut) une fin correcte donne un code de retour nul.

`wait`

`int wait(int *)`

- Entier retourné : pid du fils qui s'est terminé depuis l'invocation du wait
- Si aucun fils susceptible de se terminer alors renvoi de -1
- L'entier pointé enregistre l'état du fils lorsqu'il se finit (valeur en paramètre dans exit)

Appels système wait et exit

```
#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)

Création de processus et synchronisation

```c
#include <stdio.h>

int main(void)
{
        if (fork() == 0) {
                printf("Fin du processus fils de N° %d\n", getpid());
                exit(2);
        }
        sleep(30);
        wait(0);
}
```

Code C : p6.c

```
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>
int main() { int r,s,w;
  if ( (r=fork())==0 ) {
    printf("Fils %d\n",getpid());
    // Traitement long
    exit(14);
  } else {
    // Père doit attendre la
    // mort de son fils.
    w=wait(&s);
    printf("w:%d s:%d\n",w,s);}
}
```

Résultat

```
Machine:~/Moi> ./p4
Fils 7344
w:7344 s:2560
Machine:~/Moi>
```

Recouvrement d'un processus

Nouveau code à la place d'un autre

Un processus peut changer de code par un appel système à `exec`.

- code et données remplacés
- pointeur d'instruction réinitialisé

Appel Système : `exec()`

```
#include <unistd.h>
int execl(const char *path, const char *arg0, ...,
                const char *argn, char * /*NULL*/);
```

path = nom de l'exécutable recherché dans $PATH

arg0 = nom exécutable affiché par ps

argn = $n-2$ième argument de l'exécutable

NULL = argument de fin de la ligne de commande

Recouvrement d'un processus

Exemple : r1.c

```
#include<unistd.h>
int main() {
  printf("Je suis un programme qui va exécuter /bin/ls -l\n");
  execl("/bin/ls","ls","-l",NULL);
}
```

Recouvrement d'un processus : Résultats

```
Machine:~/Moi/Test> ls
d1  f1  f2
Machine:~/Moi/Test> ../r1
Je suis un programme qui va exécuter /bin/ls -l
total 4
drwxr-xr-x 2 blec sys 4096 2006-09-29 14:00 d1
-rw-r--r-- 1 blec sys    0 2006-09-29 14:00 f1
-rw-r--r-- 1 blec sys    0 2006-09-29 14:00 f2
Machine:~/Moi/Test>
```

Recouvrement d'un processus

1. Le programme `r1.c` exécute le printf
2. lorsqu'il exécute le execl, il est totalement remplacé par l'exécution du "ls -l"
3. le processus tel qui l'était avec le programme, disparaît

Recouvrement d'un processus : remplacement du code

Exemple : r2.c

```
#include<unistd.h>
int main() {
   printf("Je suis un programme qui va exécuter /bin/ls -l\n");
   execl("/bin/ls","ls","-l",NULL);
   printf("Je suis un programme qui a exécuté /bin/ls -l\n");
   /// Code qui ne sera pas exécuté !!!!!
}
```

Résultats

```
Machine:~/Moi/Test> ls
d1  f1
Machine:~/Moi/Test> ../r2
Je suis un programme qui va exécuter /bin/ls -l
total 4
drwxr-xr-x 2 blec aoc 4096 2006-09-29 14:00 d1
-rw-r--r-- 1 blec aoc    0 2006-09-29 14:00 f1
Machine:~/Moi/Test>
```

Création puis Recouvrement et synchronisation

Exemple : pr1.c

```
#include<stdlib.h>
#include<unistd.h>
#include<sys/wait.h>
int main() { int r,status,w;
  r=fork();
  if ( r==0 ) {
    execl("/bin/ls","ls","-a",NULL);
  }
  else {
    printf("Attente du fils\n");
    w=wait(&status);
    printf("w:%d s:%d\n",w,status);
} }
```

Résultats

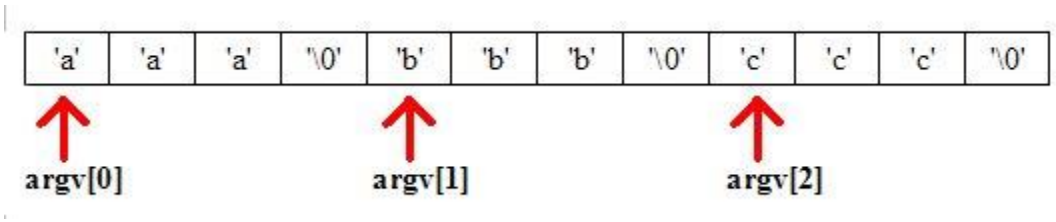
```
Machine:~/Moi/Test>../pr1
Attente du fils
..  d1 f1 f2 d2
w:7508 s:0
Machine:~/Moi/Test>
```

Lancement d'un binaire par le shell ??

Shell lance un processus fils puis recouvrement par la commande

Les paramètres : argc et argv

Le paramètre *argv* est un tableau de pointeurs.
Chacun de ces pointeurs pointent sur des chaînes de caractères :



Le paramètre *argc* quant à lui, indique simplement le nombre de chaînes de caractères sur lequel pointe *argv.*.

Par exemple, si *argc* vaut 2, cela veut dire que *argv* pointe sur deux chaînes de caractères, qu'on notera ainsi :

argv[0]
argv[1]

Exercice d'application

Ecrire un programme qui affiche les arguments qui lui ont été passés par la console.

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

int main(int argc, char *argv[])
{
   int i;

   for (i=0; i < argc; i++)
   {
      printf("Argument %ld : %s \n", i+1, argv[i]);
   }

   getchar();
   return EXIT_SUCCESS;
}
```

Bibliothèques

```
#include <unistd.h> /* necessaire pour les fonctions exec */

#include <sys/types.h> /* necessaire pour la fonction fork */

#include <unistd.h> /* necessaire pour la fonction fork */

#include <stdio.h> /* necessaire pour la fonction perror */
```

Conclusion

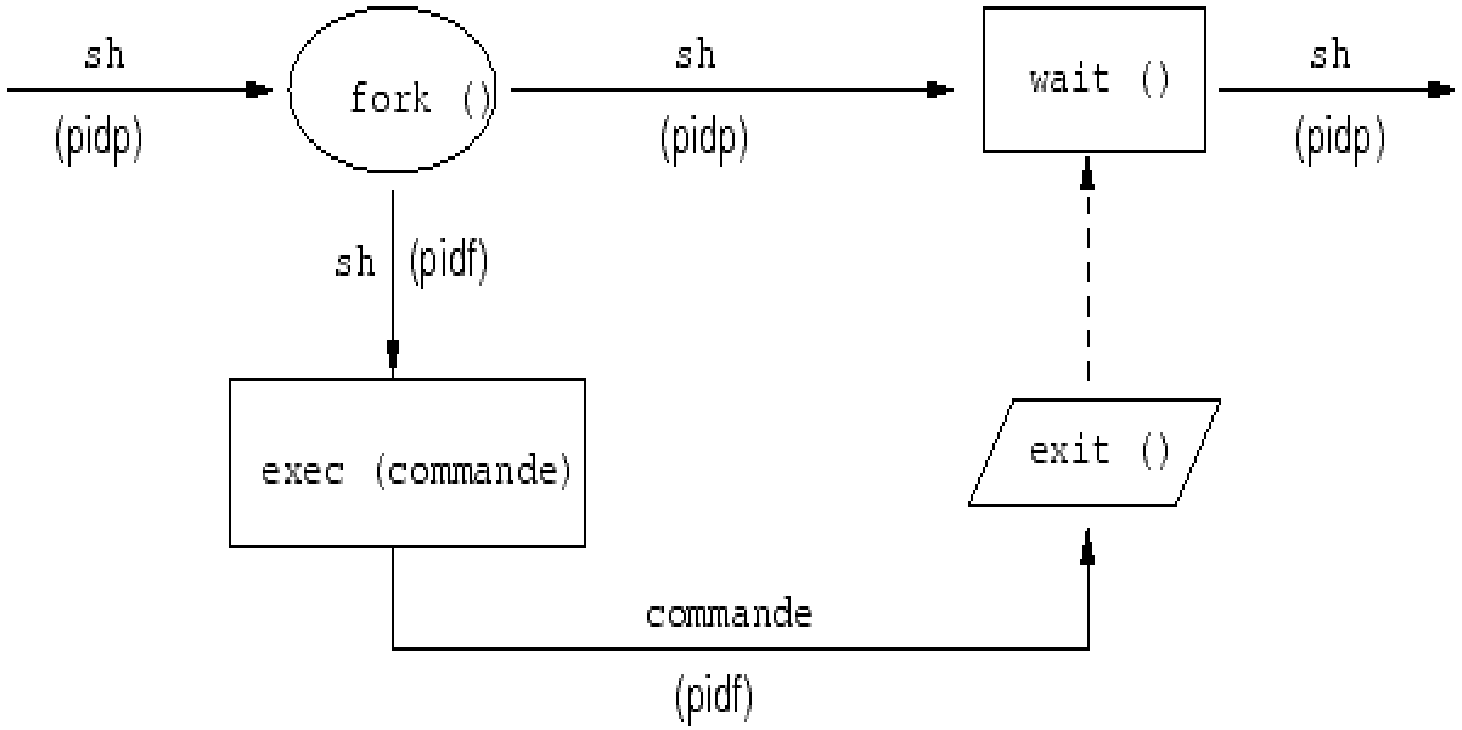


Figure 4 : Exécution d'une commande par un shell Unix

Conclusion

`pid_t fork()`

fonction qui crée un processus fils par duplication du processus qui y fait appel. En cas de succès, la fonction `fork` retourne la valeur du `pid` du fils au processus père et la valeur 0 au processus fils créé. En cas d'erreur lors de la création du processus fils, cette fonction renvoie -1 au processus père et le processus fils ne sera pas créé.

`pid_t getpid()`

renvoie le `pid` du processus qui y fait appel.

`pid_t getppid()`

renvoie le `pid` du père du processus qui y fait appel.

`void exit(int status)`

provoque la terminaison du processus en cours. Il prend en argument un code de retour. Ce code peut être récupéré par le processus père (*voir wait*). Cette fonction ne renvoie pas de valeur dans la fonction qui l'invoque.

`pid_t wait(int *status)`

attend la fin d'un processus fils. `wait` retourne le `pid` du fils en cas de succès et -1 en cas d'erreur. `status` récupère la valeur retournée par le fils, par exemple par la commande `exit(status)`.

`int exec()`

permet de lancer et d'exécuter un programme externe en remplaçant le processus en cours. Différentes variantes pour la spécification du fichier et/ou des arguments existent : `execl`, `execle`, `execlp`, `execv`, `execv`, `execvp`, ...

`execl (char *path, char *arg0, char *arg1, …, char *argn, NULL)`

- cette variante exécute un programme ;
- `path` est le chemin du fichier programme ;
- `arg0` : premier argument (correspondant au nom du programme exécutable);
- `argn` est le n ième argument.

Exemple : `execl ("/bin/ls", "ls", "-la", NULL) ;`

- exécute la commande `ls`
- située dans le répertoire `/bin/`
- en utilisant le paramètre `-la`