Programmation Système

Gestion des Processus

Université François Rabelais de Tours
Faculté des Sciences et Techniques
Antenne Universitaire de Blois

Licence Sciences et Technologies

Mention : Informatique

2ème Année

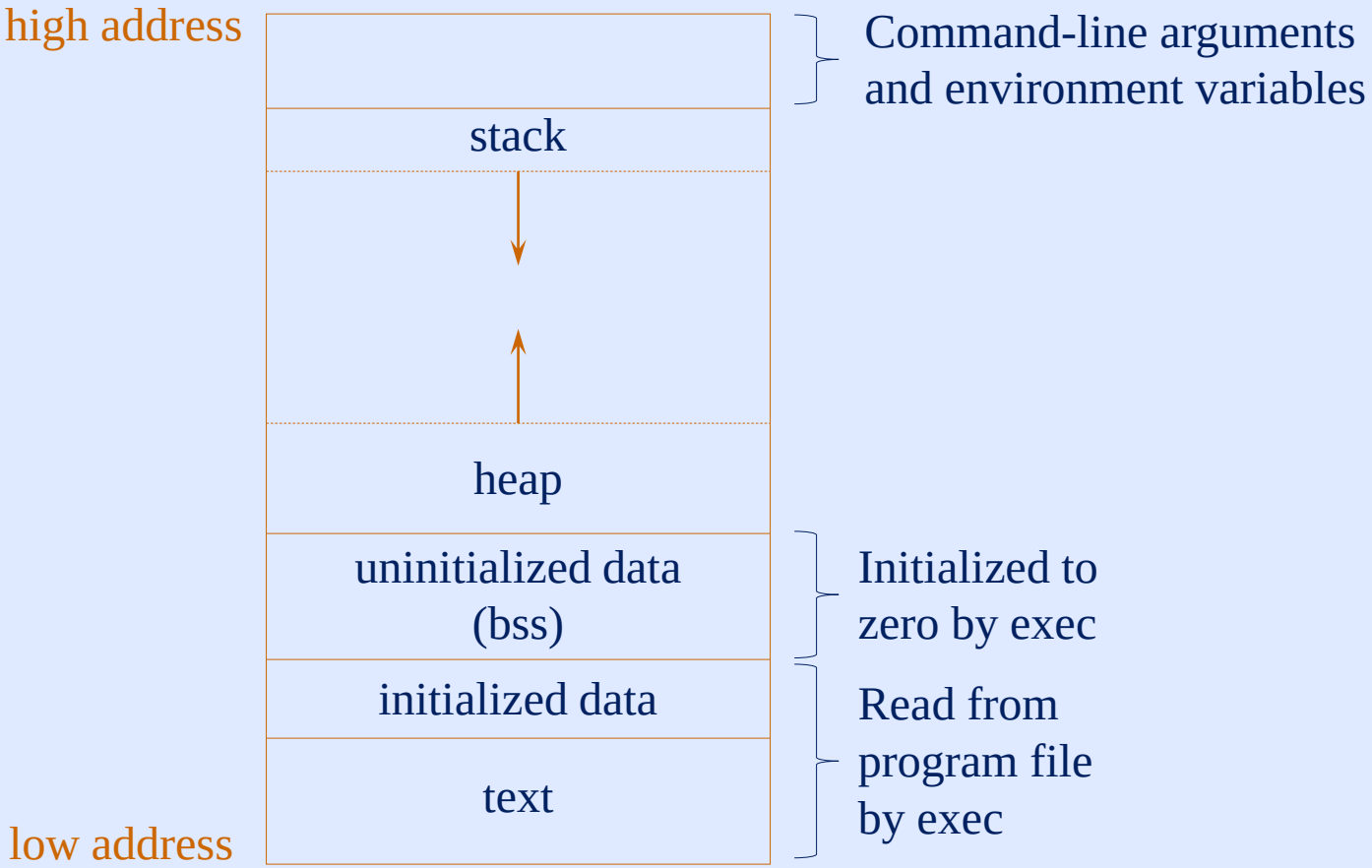
Mohamed TAGHELIT
taghelit@univ-tours.fr

La gestion des Processus

- Contexte, Types et Modes d'Exécution
- Création/Suspension/Terminaison de Processus
- Attributs des Processus
- Statistiques de Consommation CPU des Processus
- Chargement/Recouvrement de Programme
- Synchronisation des Processus Parent/Fils
- Groupes et Sessions
- Mini-Shell

Disposition Mémoire d'un Programme C

- Disposition en mémoire d'un programme C



Typical memory arrangement

Paramètres d'un Programme C

- Lorsqu'on demande l'exécution d'un programme, on peut passer des paramètres à ce dernier

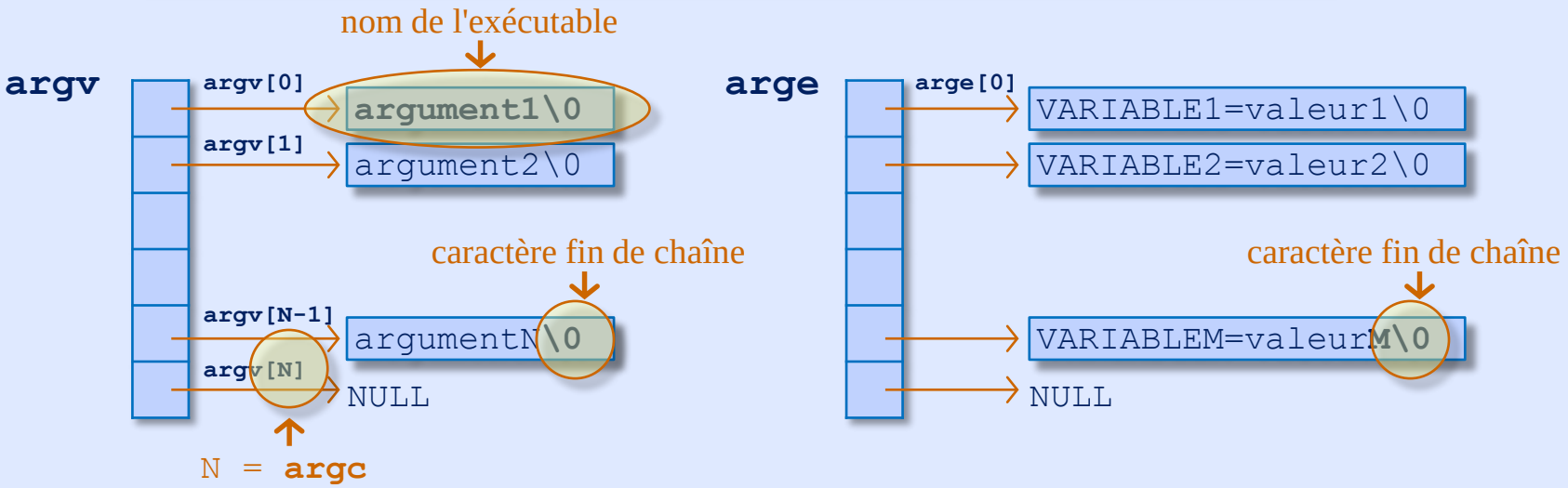
nombre d'arguments de la ligne de commande, exécutable inclus

tableau de pointeurs sur la liste des variables d'environnement

```
main (int argc, char *argv[], char *arge[]) {

        /* corps du programme */
}
```

tableau de pointeurs sur la liste des arguments, exécutable compris



Exemple : Paramètres d'un Programme C

Programme **affiche_arg.c**

```c
#include <stdio.h>

int main(int argc, char *argv[], char *arge[]) {
    int i;
    for (i = 0; i < argc; i++) printf("argv[%d] = %s\n", i, argv[i]);
    i = 0;
    while (arge[i] != NULL) {
        printf("\targe[%d] : %s\n", i, arge[i]); i++;
    }
    return 0;
}
```

Exécution de **affiche_arg.c**

```
[student]$ gcc affiche_arg.c -o affiche_arg
[student]$
[student]$
[student]$ ./affiche_arg option parametre fichier
```

exécutable — 1er paramètre — 3ème paramètre — ligne de commande — 2ème paramètre

```
argv[0] = ./affiche_arg
argv[1] = option
argv[2] = parametre
argv[3] = fichier
    arge[0] : ORBIT_SOCKETDIR=/tmp/orbit-student
    arge[1] : HOSTNAME=linux
    arge[2] : TERM=xterm
    arge[3] : SHELL=/bin/bash
    …
    …
    arge[45] : OLDPWD=/users/student/Documents/unix
[student]
```

Types et Modes d'Exécution

- Types de processus
  - Processus système

    Attachés à aucun terminal, ils sont créés par :
    - le noyau : `scheduler`, `pagedaemon`, …
    - `init` (fichiers de configuration `/etc/init`) : démons `lpd`, `ftpd`, ...
  - Processus utilisateurs

    Lancés par un utilisateur depuis un terminal.
- Modes d'exécution
  - Mode utilisateur

    Le processus exécute ses instructions et utilise ses propres données.
  - Mode noyau (système)

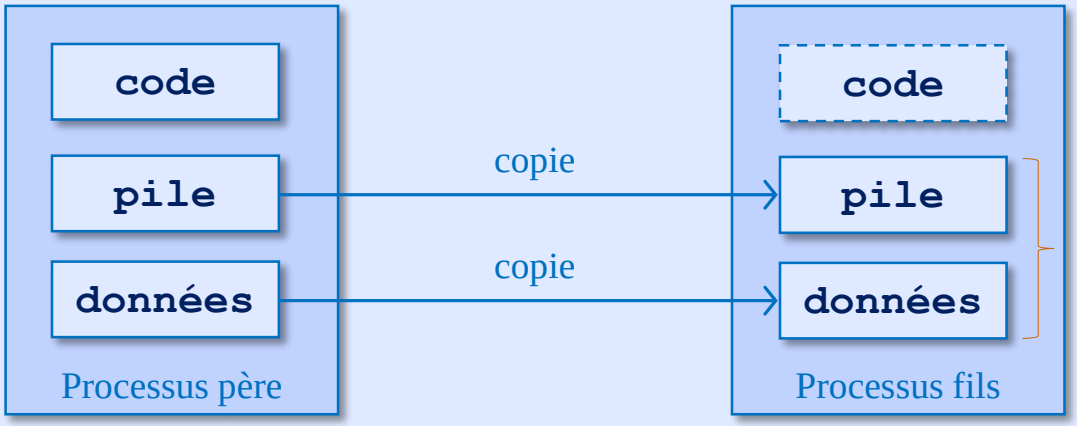
    Le processus exécute les instructions du noyau.

Création de Processus

- La seule façon de créer un nouveau processus est de faire un appel `fork()`

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

- L'appel **`fork()`** crée un nouveau processus en copiant le processus appelant – le nouveau processus est donc une copie exacte du processus appelant. Le nouveau processus est appelé “processus fils” et le processus appelant est appelé “processus père” ou “processus parent”.
- Valeur de retour :
  - **`0`**, dans le processus fils,
  - **`PID`** du processus fils dans le processus père,
  - **`-1`** en cas d'échec (et **`errno`** est modifiée en conséquence).



copy-on-write
Duplication des pages que lorsqu'un processus en modifie une instance.

Héritage dans le cas d'un *fork()*

- Attributs du processus parent hérités par le processus fils
  - descripteurs des fichiers ouverts,
  - **UID** réel, **GID** réel, **UID** effectif, **GID** effectif,
  - répertoire de travail courant,
  - les flags **set-user-ID** et **set-group-ID**,
  - masque de création (**umask**),
  - masques des signaux,
  - le flag **close-on-exec** pour tout descripteur de fichier ouvert,
  - environnement,
  - segments de mémoire attachés, …

- Attributs non hérités
  - la valeur de retour du **fork()**,
  - identifiant du processus parent,
  - les verrous posés par le processus parent,
  - l'ensemble des signaux pendants du processus fils est initialisé à vide,
  - les compteurs **tms_utime**, **tms_stime**, **tms_cutime** et **tms_ustime** du processus fils sont mis à **0**, ...

Schématisation d'un *fork()*

- Programme **fork.c**

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

```
main() {

    int i, fd, pid;

    fd = open("fichier_partage", O_WRONLY | O_CREAT, 0660);
    i = 10;

    if ( ( pid = fork() ) == 0 ){

        printf("\tJe suis le fils !\n");
        printf("\ti = %d chez le fils !\n", i);
        write(fd, "Ecriture du fils\n", strlen("Ecriture du fils\n"));

    }

    else{

        printf("Je suis le pere ! \n");
        write(fd, "Ecriture du pere\n", strlen("Ecriture du pere\n"));

    }
}
```

partie exécutée par le père (le fils n'existe pas encore)

création du processus fils par le père

partie exécutée par le fils uniquement

partie exécutée par le père uniquement

Schématisation d'un *fork()*

Chronologie d'exécution

```
main() {

  int i, fd, pid;


  fd = open("fichier_partage", …);
  i = 10;

  if ( ( pid = fork() ) == 0 ){

    printf("\tJe suis le fils …);
    printf("\ti = %d chez le fils …);
    write(fd, "Ecriture du fils …);
  }

  else{

    printf("Je suis le pere …);
    write(fd, "Ecriture du pere …);

  }
}
```

Exécution du père "héritée" par le fils

Seul le père s'exécute

Exécuté par le père. Le fils est créé et commence son exécution au retour du `fork()`.

Le père et le fils s'exécutent en "parallèle" à partir du retour au `fork()`

- Exécution de **fork.c**

```
[student]$ ./fork
Je suis le pere !
     Je suis le fils !
     i = 10 chez le fils !
[student]$ cat fichier_partage
Ecriture du pere
Ecriture du fils
[student]$
```

Héritage du descripteur et partage du pointeur de position

Parties exécutées par le père

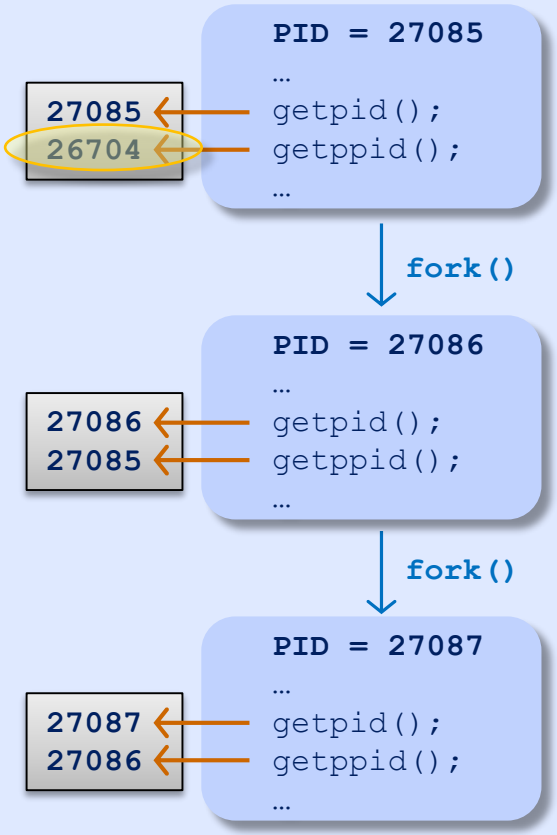
Parties exécutées par le fils

Attributs et Primitives Associées

- Tout processus peut prendre connaissance, à tout moment, de son identifiant (**PID** : **P**rocess **ID**entifier) ainsi que celui du processus parent (processus père)

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

- Valeur de retour :
  - **getpid** retourne le **PID** du processus courant (**PID** du processus appelant),
  - **getppid** retourne le **PID** du processus parent (**PID** du processus père),
  - Ces deux primitives n'ont pas de retour d'erreur. Elles réussissent toujours.



Attributs et Primitives Associées

- Tout processus peut prendre connaissance, à tout moment, de son identifiant d'utilisateur (**UID : User IDentifier**) réel et/ou effectif.

```
#include <unistd.h>
#include <sys/types.h>
        uid_t getuid(void);
        uid_t geteuid(void);
```

← retourne l'identifiant du propriétaire **réel** du processus courant

← retourne l'identifiant du propriétaire **effectif** du processus courant

- Valeur de retour :
  - **getuid** retourne l'**UID** réel du processus appelant
  - **geteuid** retourne l'**UID** effectif du processus appelant
  - Ces deux primitives n'ont pas de retour d'erreur. Elles réussissent toujours.

- Tout processus peut prendre connaissance, à tout moment, de son identifiant de groupe (**GID : Group IDentifier**) réel et/ou effectif.

```
#include <unistd.h>
#include <sys/types.h>
        gid_t getgid(void);
        gid_t getegid(void);
```

← retourne l'identifiant du groupe **réel** du processus courant

← retourne l'identifiant du groupe **effectif** du processus courant

- Valeur de retour :
  - **getgid** retourne le **GID** réel du processus appelant
  - **getegid** retourne le **GID** effectif du processus appelant
  - Ces deux primitives n'ont pas de retour d'erreur. Elles réussissent toujours.

Suspension d'un Processus

- Un processus peut suspendre son exécution durant un laps de temps qu'il définit.

```
#include <unistd.h>
      unsigned int sleep(unsigned int nb_sec);
```

- La primitive **`sleep()`** suspend le processus jusqu'à :
  - la totalité du temps spécifié par **`nb_sec`** se soit écoulé, ou
  - un signal non ignoré est intercepté par le processus

- Valeur de retour :
  - `0` si le temps prévu s'est écoulé,
  - le nombre de secondes restantes si l'appel a été interrompu par un gestionnaire de signal.

- Dans certaines implémentations, **`sleep()`** est implémentée avec la primitive **`alarm()`** (**`SVR4`**) , et il y a donc risque d'interférences.

Exemple

Programme `sleep_fork.c`

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <time.h>
#include <string.h>

main() {
   int fd; time_t now;

   fd = open("f_creation", O_WRONLY | O_CREAT, 0660);
   write(fd, "Père créé le : ", strlen("Père créé le : "));
   now=time(&now); write(fd, ctime(&now), strlen(ctime(&now)));
   sleep(5);

   if ( fork() == 0 ) {
       write(fd, "\tFils 1 créé le : ", strlen("\tFils 1 créé le : "));
       now=time(&now); write(fd, ctime(&now), strlen(ctime(&now)));
       sleep(10);

       if ( fork() == 0 ) {
           write(fd, "\t\tFils 1.1 créé le : ", strlen("\tFils 1.1 créé le : "));
           now=time(&now); write(fd, ctime(&now), strlen(ctime(&now)));
       }
   }
}
```

partie exécutée par le père

partie exécutée par le fils

partie exécutée par le petit-fils

Exécution de `sleep_fork.c`

```
[student]$ gcc sleep_fork.c -o sleep_fork
[student]$ ./sleep_fork
[student]$ cat f_creation
Père créé le : Sat Sep 20 16:16:30 2014
    Fils 1 créé le : Sat Sep 20 16:16:35 2014
        Fils 1.1 créé le :Sat Sep 20 16:16:45 2014
[student]
```

le descripteur **`fd`** est hérité du père par le fils, et du fils par le petit-fils

Temps CPU Consommé par un Processus

- Un processus peut prendre connaissance des durées **CPU** consommées par lui-même et par ses enfants terminés et attendus

```
#include <sys/times.h>
```

structure de sauvegarde des durées de consommation **CPU**

```
clock_t times(struct tms *buf);
```

- La primitive **times()** fournit les durées **CPU** consommées dans les deux modes (utilisateur et noyau) par le processus courant et ses fils (et leurs descendants) terminés et attendus. Ces durées sont stockées dans la structure **tms** pointée par **buf**.
- Valeur de retour :
  - le nombre de tops d'horloge écoulés depuis un instant arbitraire dans le passé (2^32/HZ – 300 ≅ 429 millions, depuis Linux 2.6, secondes avant le démarrage du système),
  - -1 en cas d'erreur (et **errno** est modifiée en conséquence).

```
struct tms {
    clock_t    tms_utime;     /* durée CPU en mode utilisateur du processus courant */
    clock_t    tms_stime;     /* durée CPU en mode système du processus courant */
    clock_t    tms_cutime;    /* durée CPU utilisateur des fils terminés et attendus */
    clock_t    tms_cstime;    /* durée CPU système des fils terminés et attendus */
};
```

- Toutes les durées sont exprimées en tops d'horloge. Pour les convertir en durées exprimées en secondes, les diviser par **sysconf(_SC_CLK_TCK)**.

Exemple : Calcul Temps CPU Consommé

Programme **times.c**

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

int main() {
    int fd; unsigned long i; float j = 0.0; struct tms duree;
    for ( i = 1; i < 10000000; i++) {
        fd = open("times.c", O_RDONLY); j += 0.1; close(fd);
    }
    times(&duree);
    printf("Duree user : %d tops horloge, soit %3.2f seconde(s).\n",
        duree.tms_utime, (float) duree.tms_utime / sysconf(_SC_CLK_TCK));
    printf("Duree system : %d tops horloge, soit %3.2f seconde(s).\n",
        duree.tms_stime, (float) duree.tms_stime / sysconf(_SC_CLK_TCK));
    printf("\tDuree user fils : %d tops horloge, soit %3.2f seconde(s).\n",
        duree.tms_cutime, (float) duree.tms_cutime / sysconf(_SC_CLK_TCK));
    printf("\tDuree system fils : %d tops horloge, soit %3.2f seconde(s).\n",
        duree.tms_cstime, (float) duree.tms_cstime / sysconf(_SC_CLK_TCK));
    return 0;;
}
```

récupération des durées **CPU** et stockage dans **duree**

affichage des durées **CPU** consommées par le processus courant

affichage des durées **CPU** consommées par les descendants du processus courant

Exécution de **times.c**

```
[student]$ gcc Times.c -o Times
[student]$ ./Times
Duree user : 650 tops horloge, soit 6.50 seconde(s).
Duree system : 210 tops horloge, soit 2.10 seconde(s).
   Duree user fils : 0 tops horloge, soit 0.00 seconde(s).
   Duree system fils : 0 tops horloge, soit 0.00 seconde(s).
[student]$
```

le processus n'a aucun descendant

La priorité des Processus

- La priorité d'ordonnancement d'un processus, d'un groupe de processus ou d'un utilisateur, peut être lue ou fixée.

```
#include <sys/time.h>
#include <sys/resource.h>
```

```
int getpriority(int which, int who);
```
← lit la priorité

```
int setpriority(int which, int who, int prio);
```
← fixe la priorité

which	who
PRIO_PROCESS	→ ID de processus
PRIO_PGRP	→ ID de groupe de processus
PRIO_USER	→ ID d'utilisateur

- Usage
  - Si **who** = **0** ⇒ processus (groupe du processus ou **UID** réel du processus) appelant.
  - Les valeurs de **prio ∈ [-20, +19]**. La priorité par défaut est **0**.
  - Seul le super utilisateur peut diminuer la valeur numérique de la priorité.
- Valeur de retour
  - **getpriority()** retourne la plus haute priorité (la plus basse valeur numérique) dont bénéficie l'un des processus indiqués ou -1 en cas d'erreur.

    Remarque : **-1** peut être une priorité légitime (effacer **errno** avant appel).
  - **setpriority()** retourne **0** en cas de succès ou **-1** en cas d'erreur.

Exemple : Lecture/Écriture de Priorité

Programme `getsetprio.c`

```c
#include <stdio.h>
#include <sys/time.h>
#include <sys/resource.h>

int main(int argc, char **argv) {

    printf("\tMa priorité actuelle : %d\n", getpriority(PRIO_PROCESS, 0));

    if (setpriority(PRIO_PROCESS, 0, atoi(argv[1])) == 0)
        printf("\tMa nouvelle priorité  : %d\n", getpriority(PRIO_PROCESS, 0));
    else
        perror("Erreur modification priorité ");
    pause();
}
```

retourne la priorité actuelle du processus

fixe la priorité du processus à la valeur définie par le 3ème paramètre (`argv[1]`).

Exécution de `getsetprio.c`

```
[student]$ gcc getsetprio.c -o getsetprio
[student]$ ./getsetprio 10
  Ma priorité actuelle : 0
  Ma nouvelle priorité  : 10
^C
[student]$ ./getsetprio -10
  Ma priorité actuelle : 0
  Erreur modification priorité : Permission denied
^C
[student]$ sudo ./getsetprio -10
  Ma priorité actuelle : 0
  Ma nouvelle priorité  : -10
^C
[student]$
```

```
[student]$ ps -l -t pts/1
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1011  1708  1678  0  80   0 - 27092 wait   pts/1    00:00:00 bash
0 S  1011  3068  1708  0  90  10 -   981 pause  pts/1    00:00:00 getsetprio
[student]
```

visualisation à partir d'un autre terminal

```
[student]$ ps -l -t pts/1
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1011  1708  1678  0  80   0 - 27092 wait   pts/1    00:00:00 bash
4 S     0  3072  3071  0  70 -10 -   981 pause  pts/1    00:00:00 getsetprio
[student]
```

Priorité de Courtoisie d'un Processus

- La priorité de courtoisie (politesse) d'un processus peut être augmentée ou diminuée.

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

valeur ajoutée à la priorité de courtoisie du processus appelant

- Usage
  - La primitive **`nice()`** ajoute **`inc`** à la valeur de courtoisie du processus appelant. Une grande valeur de courtoisie implique une faible priorité.
  - Seul le super utilisateur peut donner une valeur négative à **`inc`** ⟹ augmenter la priorité du processus appelant.
  - Les valeurs de **`inc ∈ [-20, +19]`**. La priorité par défaut est **`0`**.
  - La valeur de courtoisie d'un processus peut être obtenue grâce à **`getpriority()`**.

- Valeur de retour
  - **`nice()`** retourne la nouvelle valeur de courtoisie si elle réussit,

    Remarque : **`-1`** peut être une priorité légitime (effacer **`errno`** avant appel).
  - **`-1`** en cas d'erreur (et **`errno`** est modifiée en conséquence),

Exemple : Priorité de Courtoisie

Programme `nice.c`

```
#include <unistd.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <stdio.h>
#include <errno.h>
```

```
int main(int argc, char **argv) {
    int vr;
    printf("\tMa courtoisie actuelle : %d\n",
        getpriority(PRIO_PROCESS, 0));
    sleep(5); errno = 0;
    vr = nice(atoi(argv[1]));
    if ( errno == 0)
        printf("\tMa nouvelle courtoisie : %d\n", vr);
    else
        perror("\tErreur modification priorité ");
    pause();
}
```

Exécution de `getsetprio.c`

```
[student]$ gcc nice.c -o nice
[student]$ ./nice 10
  Ma courtoisie actuelle : 0
  Ma nouvelle courtoisie : 10
^C
[student]$ ./nice -10
  Ma courtoisie actuelle : 0
  Erreur modification priorité : Operation not permitted
^C
[student]$ sudo ./nice -10
  Ma courtoisie actuelle : 0
  Ma nouvelle courtoisie : -10
^C
[student]$
```

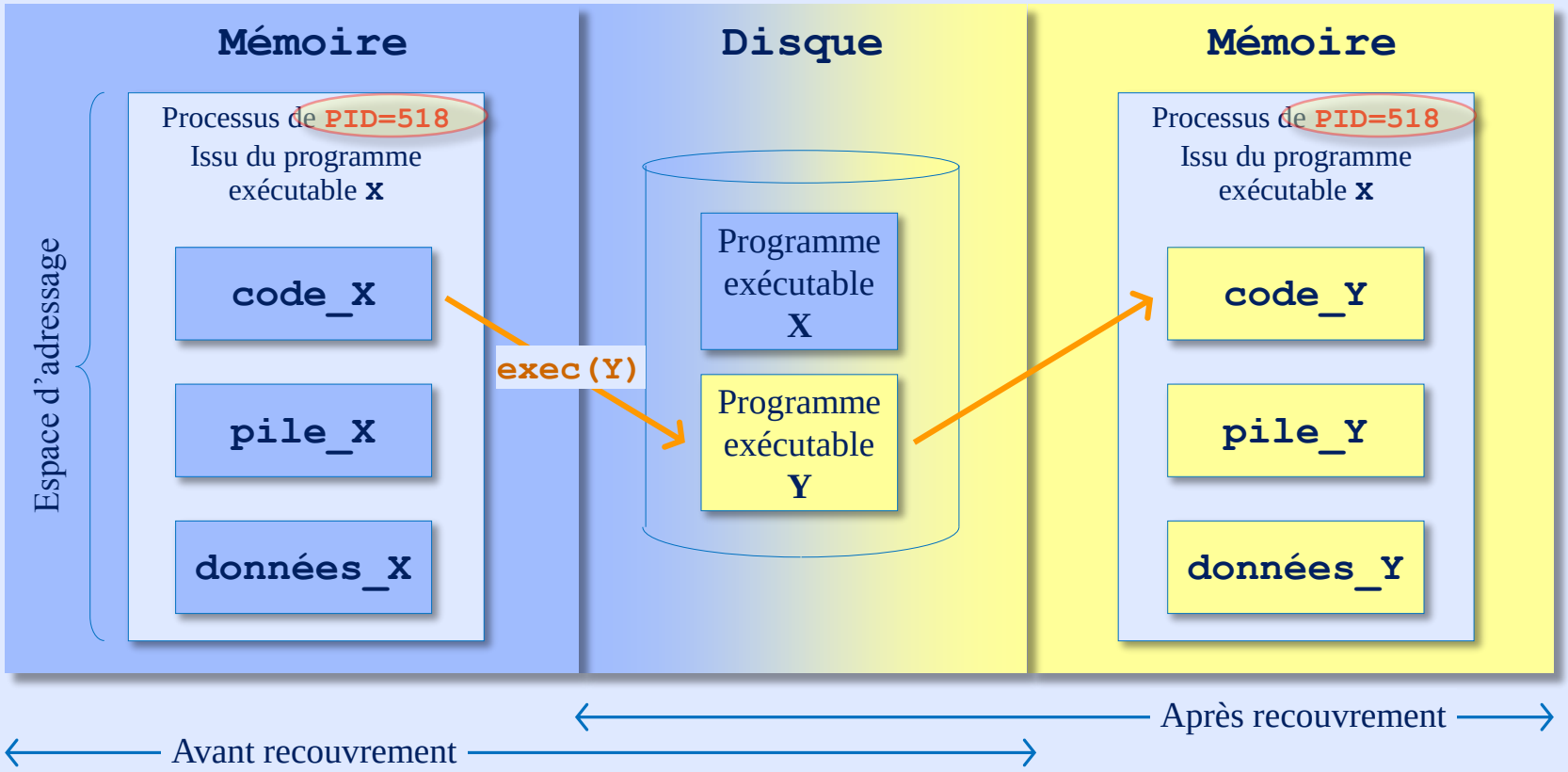
```
[student]$ ps -l -t pts/1
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1011  1708  1678  0  80   0 - 27092 wait   pts/1    00:00:00 bash
0 S  1011  3098  1708  0  90  10 -   981 pause  pts/1    00:00:00 nice
[student]
```

visualisation à partir d'un autre terminal

```
[student]$ ps -l -t pts/1
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1011  1708  1678  0  80   0 - 27092 wait   pts/1    00:00:00 bash
4 S     0  3106  3105  0  70 -10 -   981 pause  pts/1    00:00:00 nice
[student]
```

Le Recouvrement de Programme

- Il est possible à un processus de remplacer son image mémoire par une nouvelle image mémoire (obtenue de l'exécution d'un autre programme).



- Le processus courant "disparait" au profit du nouveau processus qui "hérite" de son **PID** (pour le système, il n'y a pas de création de nouveau processus).

Les Primitives de Recouvrement

- Les primitives de la famille **`exec()`** remplacent l'image mémoire du processus courant par une nouvelle image mémoire.

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

chemin (nom) de l'exécutable

liste des arguments que recevra le processus et qu'il accèdera via son paramètre **`argv`** du **`main`**

Frontaux à **`execve()`**

```
int execl  (const char *path, const char *arg0, *arg1, …,(char *) NULL);
int execlp (const char *file, const char *arg0, *arg1, …,(char *) NULL);
int execle (const char *path, const char *arg0, *arg1, …,(char *) NULL,
                                                    char * const envp[]);
int execv  (const char *path, char * const argv[]);
int execvp (const char *file, char * const argv[]);
int execve (const char *path, char * const argv[], char * const envp[]);
```

liste des arguments qui sera disponible au programme à exécuter via son paramètre **`argv`** du **`main`**

indique l'environnement qui sera disponible au programme à exécuter via son paramètre **`arge`** du **`main`**

- Les primitives se différentient par :
  - la transmission des arguments (suffixes **`l`** et **`v`**),
  - la recherche du fichier (suffixe **`p`**), et
  - la conservation ou pas de l'environnement (suffixe **`e`**).
- Valeur de retour
  - Ces primitives ne retournent pas en cas de succès et renvoient **`-1`** an cas d'échec.
- Par convention, le premier argument doit pointer sur le nom du fichier associé au programme à exécuter.

Attributs Conservés par le Nouveau Programme

- Le nouveau programme hérite, lors de son exécution, des objets suivants :
  - identifiant de processus (**`PID`**) et identifiant du processus parent (**`PPID`**),
  - identifiant utilisateur réel (**`UID`**) et identifiant de groupe réel (**`GID`**),
  - identifiants de groupes supplémentaires,
  - identifiant de session,
  - terminal de contrôle,
  - répertoire de travail courant,
  - masque de création de fichier,
  - verrous sur les fichiers,
  - masque des signaux,
  - signaux pendants,

  - Les signaux non ignorés (captés) retrouvent leur comportement par défaut.
  - L'héritage des descripteurs de fichiers ouverts dépend de la valeur de leur flag **`FD_CLOEXEC`** respectif.

Exemple : Recouvrement

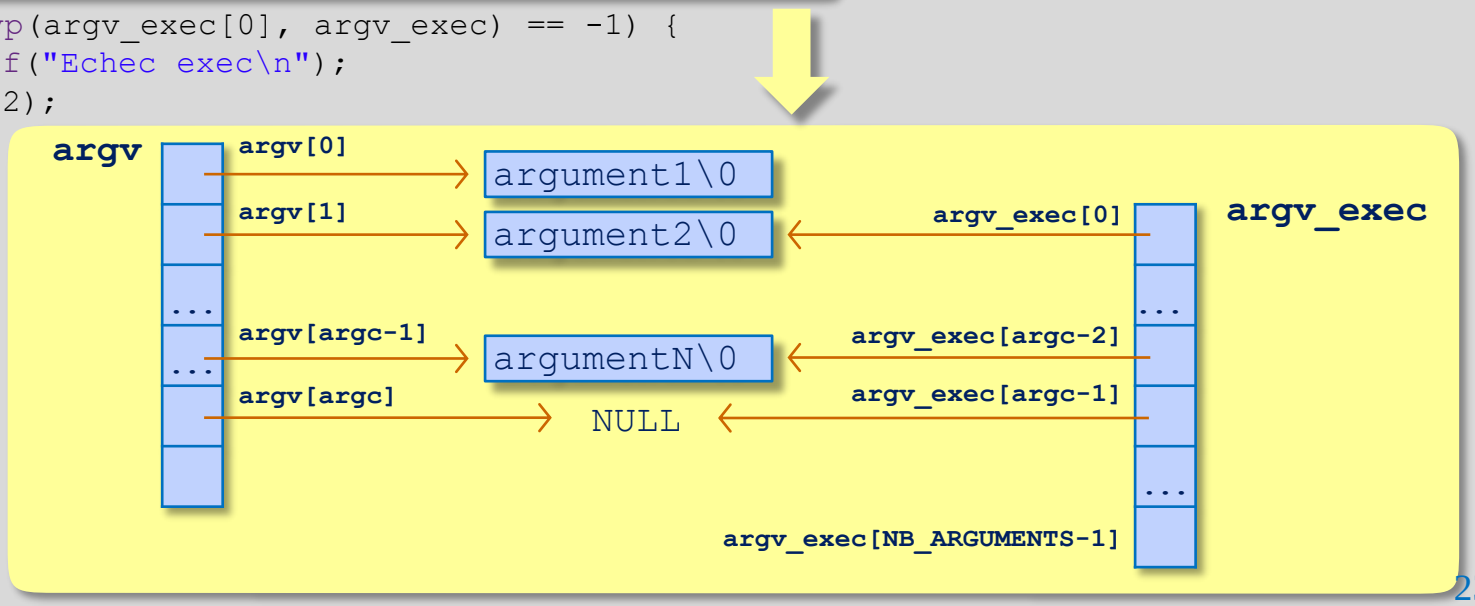
- Programme **execv.c**

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

```
#define NB_ARGUMENTS 16

int main(int argc, char **argv, char **arge) {
    char *argv_exec[NB_ARGUMENTS];
    int indice;

    if (argc < 2) {
        printf("Usage : programme commande liste_arguments\n");
        exit(1);
    }
        for (indice = 0; indice < argc; indice++)
                argv_exec[indice] = argv[indice + 1];
    if (execvp(argv_exec[0], argv_exec) == -1) {
        printf("Echec exec\n");
        exit(2);
    }
}
```

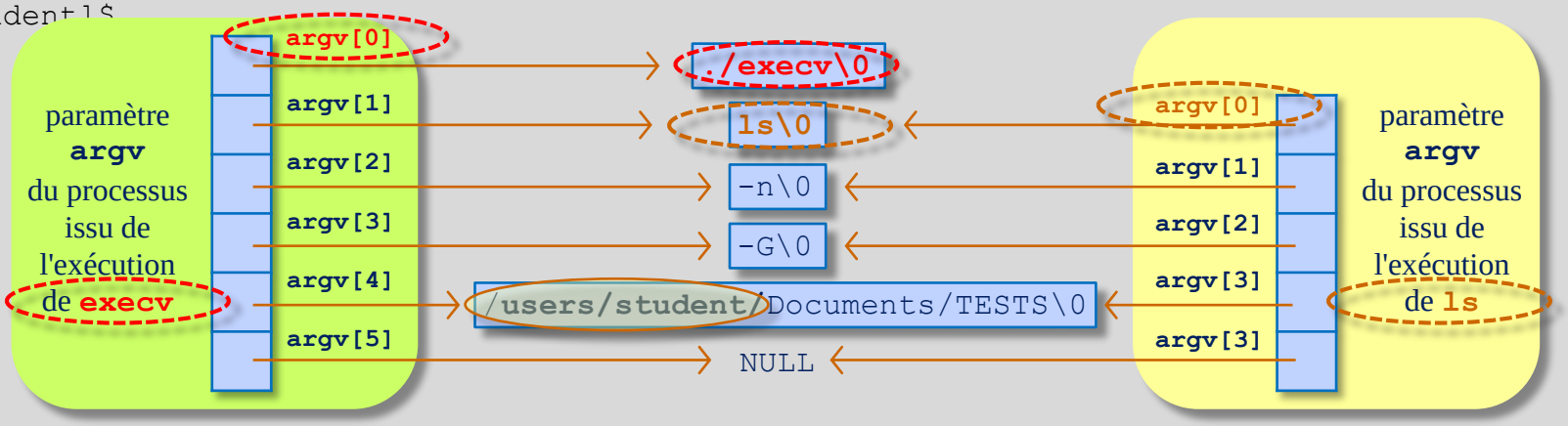


Exemple : Recouvrement

- Exécution de **execv.c**

remplacé par le **shell** par la valeur de **HOME**

```
[student]$ ls -n -G ~/Documents/TESTS
total 8
drwxrwxr-- 2    0 4096  2 sept. 15:10 DONNEES
-rw-rw-r-- 1 1011   71  2 août  15:06 fichier
lrwxrwxrwx 1 1011    4  2 août  15:04 file -> pipe
crw-rw---- 1    0 6, 0  2 août  15:00 lp0
prw-rw-r-- 1 1011    0  2 août  14:56 pipe
brw-rw---- 1    0 8, 0  2 août  15:05 sda
[student]$
[student]$ gcc execv.c -o execv
[student]$ ./execv ls -n -G ~/Documents/TESTS
total 8
drwxrwxr-- 2    0 4096  2 sept. 15:10 DONNEES
-rw-rw-r-- 1 1011   71  2 août  15:06 fichier
lrwxrwxrwx 1 1011    4  2 août  15:04 file -> pipe
crw-rw---- 1    0 6, 0  2 août  15:00 lp0
prw-rw-r-- 1 1011    0  2 août  14:56 pipe
brw-rw---- 1    0 8, 0  2 août  15:05 sda
[student]$
```



Exemple : Recouvrement

Programme `execl.c`

```
int main(int argc, char **argv, char **arge){
    if ( execl("ls", "ls", "-n", "-G", "/users/student/Documents/TESTS", NULL) == -1 ) {
        perror("\tEchec execl"); exit(1);
    }
}
```

programme à exécuter

`arg0` : doit pointer sur le nom du fichier associé au programme à exécuter

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

Programme `execlp.c`

```
int main(int argc, char **argv, char **arge){
    if ( execlp("ls", "ls", "-n", "-G", "/users/student/Documents/TESTS", NULL) == -1 ) {
        perror("\tEchec execlp"); exit(1);
    }
}
```

Exécution de `execl.c` et de `execlp.c`

```
[student]$ ./execl
    Echec execl: No such file or directory
[student]$ ./execlp
total 8
drwxrwxr-- 2    0 4096  2 sept. 15:10 DONNEES
-rw-rw-r-- 1 1011   71  2 août  15:06 fichier
lrwxrwxrwx 1 1011    4  2 août  15:04 file -> pipe
crw-rw---- 1    0 6, 0  2 août  15:00 lp0
prw-rw-r-- 1 1011    0  2 août  14:56 pipe
brw-rw---- 1    0 8, 0  2 août  15:05 sda
[student]$
```

l'exécutable **`ls`** n'est pas présent dans le répertoire courant

recherche l'exécutable **`ls`** dans tous les répertoires dont le chemin est défini dans **`PATH`**

Synchronisation Entre Processus

- Un processus peut attendre la terminaison d'un de ses fils.

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

si pas **NULL**, **wait()** y stocke l'état du fils terminé

Macros évaluant **status** (après retour de **wait()**)

WIFEXITED(status) : vrai si le fils s'est terminé normalement (**exit**, **_exit** ou **return**)
WEXITSTATUS(status) : donne le code de retour du fils (**WIFEXITED ≠ 0**)
WIFSIGNALED(status) : vrai si le fils s'est terminé à cause d'un signal non intercepté
WTERMSIG(status) : donne le numéro de signal qui a causé la fin du fils (**WIFSIGNALED ≠ 0**)
…

- L'appel **wait()** met en attente le processus appelant jusqu'à la terminaison de l'un quelconque de ses fils, puis stocke l'état de ce dernier dans **status**.

- Valeur de retour
  - **PID** du processus fils terminé en cas de réussite
  - **-1** en cas d'échec

Exemple : Synchronisation

- Programme **wait.c**

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

```
main() {
int pid; int status;

if ( ( pid = fork() ) == 0 ) {
   printf("\tFils 1 de pid : %d\n", getpid());
   pause();
}
else    if ( ( pid = fork() ) == 0 ) {
           printf("\tFils 2 de pid : %d\n", getpid());
           pause();
        }
        else    if ( ( pid = fork() ) == 0 ) {
                   printf("\tFils 3 de pid : %d\n", getpid());
                   exit( 3 );
                }
                else    while(1) {
                            if ( (pid = wait(&status)) != -1) {
                                printf("Fils de pid %d terminé !\n", pid);
                                if ( WIFEXITED(status) )
                                    printf("\tTerminaison normale avec code de retour
                                                %d\n", WEXITSTATUS(status));
                                else    if ( WIFSIGNALED(status) )
                                            printf("\tTerminaison sur réception du signal
                                                %d\n", WTERMSIG(status));
                            }
                            else {
                                perror("Erreur wait");
                                printf("Fin du père !\n");
                                return -1;
                            }
                        }
}
```

exécuté par le fils 1

exécuté par le fils 2

exécuté par le fils 3

exécuté par le père

Exemple : Synchronisation

Exécution de `wait.c`

```
[student]$ gcc wait.c -o wait
[student]$ ./wait &
[1] 2688
  Fils 2 de pid : 2690
  Fils 1 de pid : 2689
  Fils 3 de pid : 2691
```

← chaque processus fils affiche son **PID**

```
Fils de pid 2691 terminé !
    Terminaison normale avec code de retour 3
```

← après retour du **wait()**, le père affiche le **PID** du fils terminé et la cause de la terminaison

```
[student]$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1011  1741  1683  0  80   0 - 27127 wait   pts/2    00:00:00 bash
0 S  1011  2688  1741  0  80   0 -   981 wait   pts/2    00:00:00 wait
1 S  1011  2689  2688  0  80   0 -   981 pause  pts/2    00:00:00 wait
1 S  1011  2690  2688  0  80   0 -   981 pause  pts/2    00:00:00 wait
0 R  1011  2692  1741  1  80   0 - 27034 -      pts/2    00:00:00 ps
[student]$ kill -TERM 2690
```

← émission du signal **TERM** au processus fils de **PID 2690**

```
Fils de pid 2690 terminé !
    Terminaison sur réception du signal 15
```

← après retour du **wait()**, le père affiche le **PID** du fils terminé et la cause de la terminaison

```
[student]$ kill -USR1 2689
```

← émission du signal **USR1** au processus fils de **PID 2689**

```
Fils de pid 2689 terminé !
    Terminaison sur réception du signal 10
```

← après retour du **wait()**, le père affiche le **PID** du fils terminé et la cause de la terminaison

```
Erreur wait: No child processes
Fin du père !
```

← échec du **wait()**, affichage de la raison de l'échec puis le père se termine

```
[student]$
```

Synchronisation Entre Processus

- Un processus peut attendre qu'un de ses fils changent d'état.
  - un processus est considéré comme changeant d'état s'il se termine, s'il est stoppé ou relancé par un signal.

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

si pas **NULL**, stocke l'état du fils terminé

tout processus fils dans le groupe **|pid|** : **pid < -1**
tout processus fils : **pid = -1**
tout processus fils du même groupe que l'appelant : **pid = 0**
processus fils d'identité **pid** : **pid > 0**

OU binaire entre les constantes suivantes :

WNOHANG : ne pas bloquer si aucun fils ne s'est terminé
WUNTRACED : recevoir l'information concernant également les fils bloqués
WCONTINUED : recevoir l'information concernant également les fils stoppés quand ils sont relancés par le signal **SIGCONT**

- L'appel **waitpid()** met en attente le processus appelant jusqu'au changement d'état de l'un de ses fils spécifié par **pid**, puis stocke l'état de ce dernier dans **status**. Par défaut, **waitpid()** attend la terminaison d'un fils.
- Valeur de retour
  - **PID** du processus fils dont l'état a changé, en cas de réussite
  - **0** si **WNOHANG** utilisé et aucun fils n'a changé d'état
  - **-1** en cas d'échec

Exemple : Synchronisation

- Programme **waitpid.c**

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

```
main() {
int pid; int status;

if ( ( pid = fork() ) == 0 ) {
   printf("\tFils de pid : %d\n", getpid());
   pause();
}

else    while(1) {
            if ( (pid = waitpid(-1, &status, WUNTRACED | WCONTINUED)) != -1) {
                printf("Fils de pid %d a changé d'état !\n", pid);
                if ( WIFEXITED(status) )
                    printf("\tTerminaison normale avec code de retour %d\n",
                                                        WEXITSTATUS(status));
                else    if ( WIFSIGNALED(status) )
                            printf("\tTerminaison sur réception du signal %d\n",
                                                        WTERMSIG(status));
                        else    if ( WIFSTOPPED(status) )
                                    printf("\tFils stoppé\n");
                                else    if ( WIFCONTINUED(status) )
                                            printf("\tFils relancé\n");
            }
            else {
                perror("Erreur waitpid");
                printf("Fin du père !\n");
                return -1;
            }
        }
}
```

exécuté par le fils

attente des fils stoppés et/ou relancés (en plus, par défaut, des fils terminés)

exécuté par le père

Exemple : Synchronisation

Exécution de `waitpid.c`

```
 [student]$ ./waitpid &
[1] 2164
   Fils de pid : 2165
[student]$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1011  1736  1679  0  80   0 - 27092 wait   pts/2    00:00:00 bash
0 S  1011  2164  1736  0  80   0 -   980 wait   pts/2    00:00:00 waitpid
1 S  1011  2165  2164  0  80   0 -   981 pause  pts/2    00:00:00 waitpid
0 R  1011  2166  1736  0  80   0 - 27034 -      pts/2    00:00:00 ps
[student]$ kill -STOP 2165
Fils de pid 2165 a changé d'état !
    Fils stoppé
[student]$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1011  1736  1679  0  80   0 - 27092 wait   pts/2    00:00:00 bash
0 S  1011  2164  1736  0  80   0 -   981 wait   pts/2    00:00:00 waitpid
1 T  1011  2165  2164  0  80   0 -   981 signal pts/2    00:00:00 waitpid
0 R  1011  2167  1736  0  80   0 - 27034 -      pts/2    00:00:00 ps
[student]$ kill -CONT 2165
Fils de pid 2165 a changé d'état !
    Fils relancé
[student]$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1011  1736  1679  0  80   0 - 27092 wait   pts/2    00:00:00 bash
0 S  1011  2164  1736  0  80   0 -   981 wait   pts/2    00:00:00 waitpid
1 S  1011  2165  2164  0  80   0 -   981 pause  pts/2    00:00:00 waitpid
0 R  1011  2168  1736  1  80   0 - 27035 -      pts/2    00:00:00 ps
[student]$ kill -TERM 2165
Fils de pid 2165 a changé d'état !
    Terminaison sur réception du signal 15
Erreur waitpid: No child processes
Fin du père !
[1]+  Exit 255                ./waitpid
[student]$
```

le processus fils affiche son **PID**

émission du signal **STOP** au processus fils de **PID 2165**

après retour du **waitpid()**, le père affiche le **PID** du fils qui a changé d'état et indique ce dernier

émission du signal **CONT** au processus fils de **PID 2165**

après retour du **waitpid()**, le père affiche le **PID** du fils qui a changé d'état et indique ce dernier

émission du signal **TERM** au processus fils de **PID 2165**

après retour du **waitpid()**, le père affiche le **PID** du fils qui a changé d'état et indique ce dernier

échec du **waitpid()**, affichage de la raison de l'échec puis le père se termine

Terminaison de Processus

- Un processus peut, à tout moment, mettre fin normalement à son exécution.

valeur renvoyée au processus parent

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

- L'appel **`exit()`** (appartient à la bibliothèque standard) met fin normalement à l'exécution de l'appelant :
  - toutes les fonctions enregistrées par **`atexit()`** et **`on_exit()`** sont appelées dans l'ordre inverse de leur enregistrement,
  - tous les flux d'**`E/S`** standards ouverts sont vidés et fermés,
  - la valeur de **`status & 0377`** est envoyée au processus parent,
  - son processus parent reçoit un signal **`SIGCHLD`** (si l'implémentation supporte ce signal),
  - met fin normalement à l'exécution du processus appelant.

- Valeur de retour
  - La fonction **`exit()`** ne revient jamais.

Terminaison de Processus

- Un processus peut, à tout moment, mettre fin immédiatement à son exécution.

valeur renvoyée au processus parent

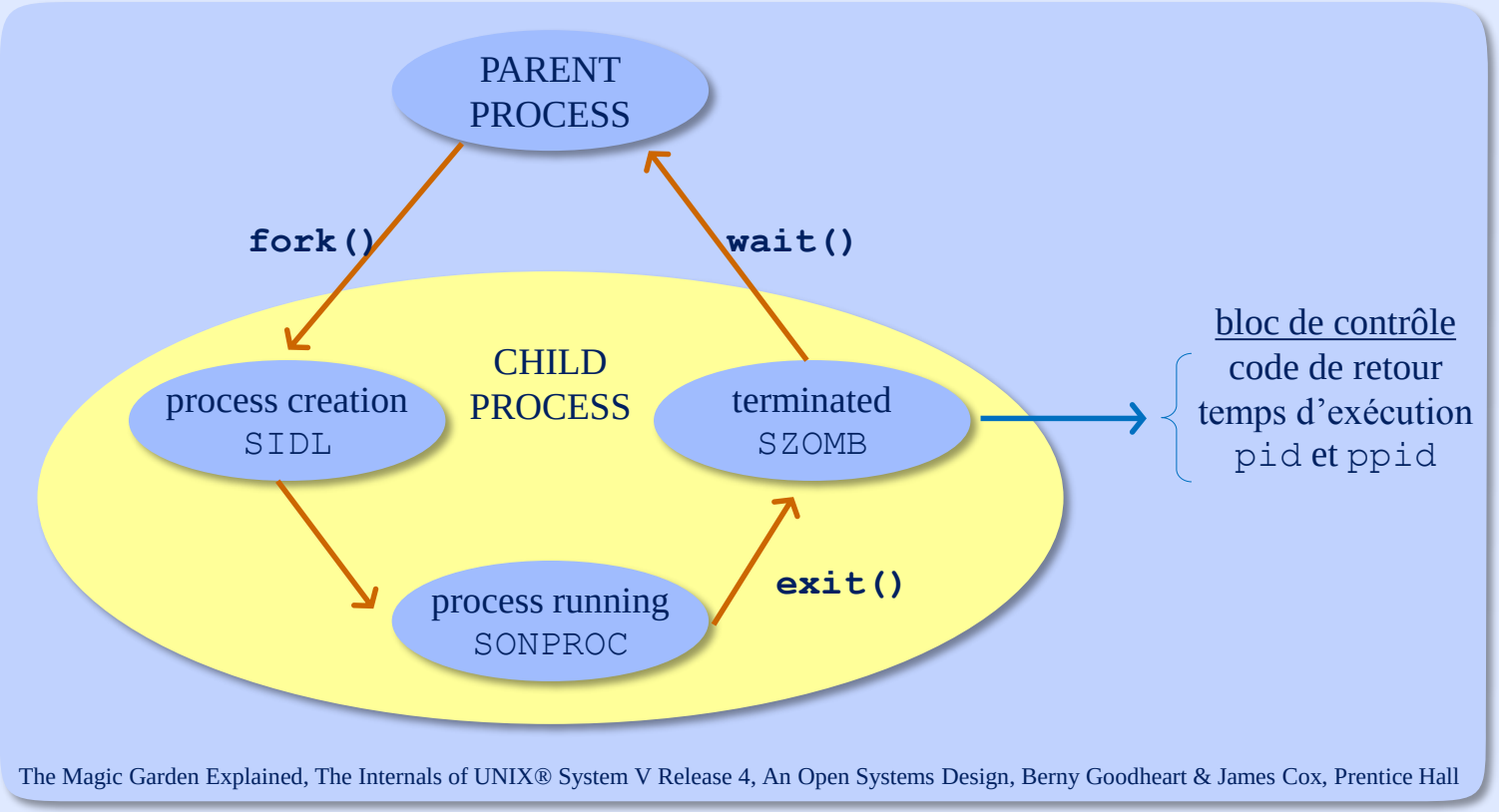
```
#include <unistd.h>
                void _exit(int code);
```

- L'appel **`_exit()`** (appel système) met fin immédiatement à l'exécution de l'appelant :
  - les descripteurs de fichier appartenant à l'appelant sont fermés,
  - tous ses éventuels processus fils sont hérités par le processus de **`PID 1`** (**`init`**),
  - son processus parent reçoit un signal **`SIGCHLD`**.

- Valeur de retour
  - La fonction **`_exit()`** ne revient jamais.

Phases de Terminaison d'un Processus

- Phases de création et de terminaison d'un processus.



The Magic Garden Explained, The Internals of UNIX® System V Release 4, An Open Systems Design, Berny Goodheart & James Cox, Prentice Hall

- Si le parent n'est pas en attente, et n'a pas indiqué qu'il désire ignorer le code de retour, le processus fils devient un processus "zombie".

Exemple : Terminaison

- Programme **stdio_exit.c**

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

main() {
     int pid;

     if ( fork() == 0 ) {
          if ( fork() == 0 ) {
               sleep(3);
               printf("\tPetit Fils de PID = %d", getpid());
               sleep(5);
               exit(0);
          }
          else {
               printf("\tFils de PID = %d", getpid());
               sleep(4);
               exit(0);
          }
     }
     else {
          printf("\tPère de PID = %d", getpid());
          sleep(10);
          _exit(0);
     }
}
```

création du fils par le père

création du petit fils par le fils

exécuté par le petit fils

exécuté par le fils

exécuté par le père

Exemple : Terminaison

- Exécution de **stdio_exit.c**

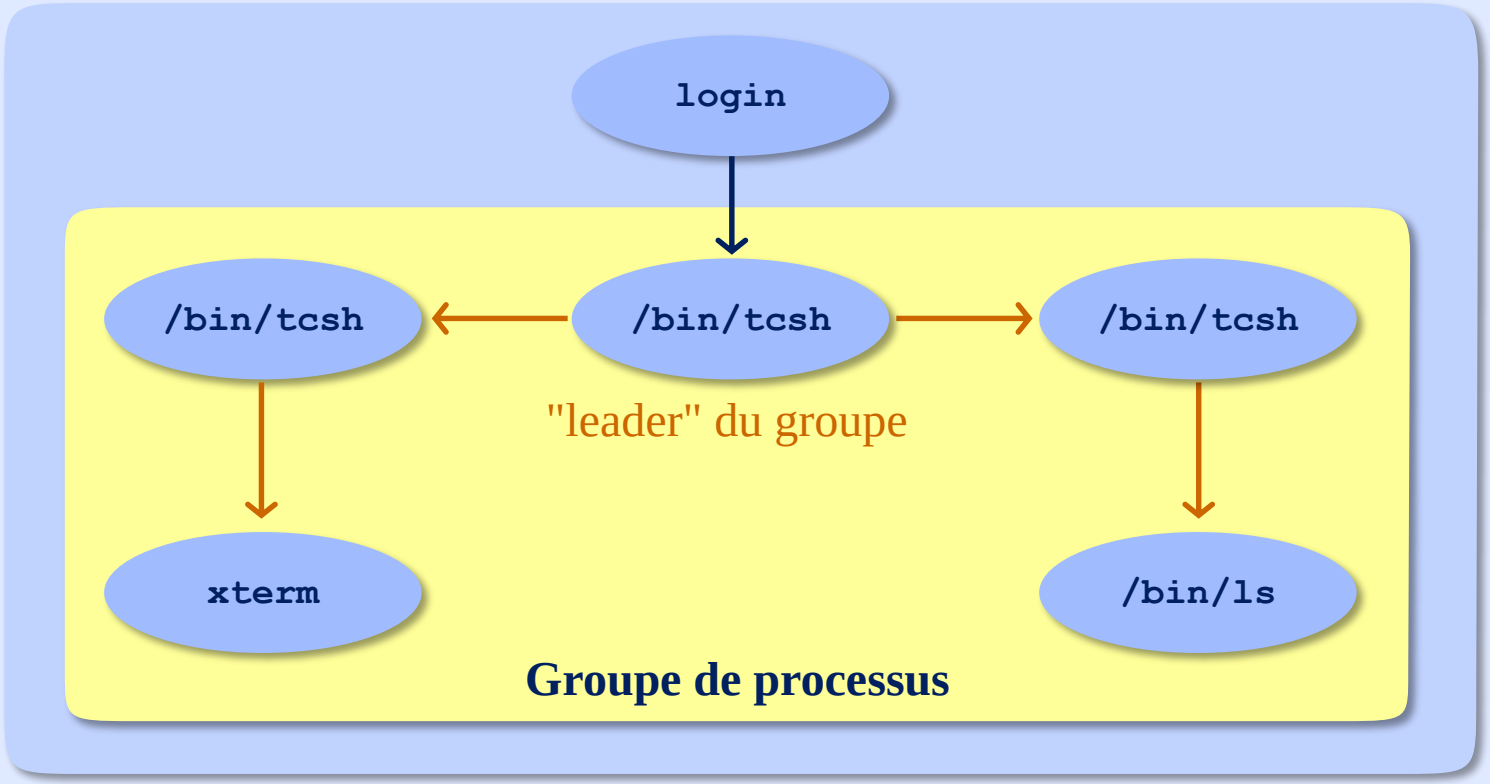
```
[student]$ gcc stdio_exit.c -o stdio_exit
[student]$ ./stdio_exit &
[1] 3637                    ← PID du père
[student]$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1011  1841  1789  0  80   0 - 27092 wait   pts/1    00:00:01 bash
0 S  1011  3637  1841  0  80   0 -   981 hrtime pts/1    00:00:00 stdio_exit   ← père
1 S  1011  3638  3637  0  80   0 -   981 hrtime pts/1    00:00:00 stdio_exit   ← fils
1 S  1011  3639  3638  0  80   0 -   980 hrtime pts/1    00:00:00 stdio_exit   ← petit fils
0 R  1011  3640  1841  0  80   0 - 27035 -      pts/1    00:00:00 ps
[student]$          Fils de PID = 3638   ← affichage du fils
[student]$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1011  1841  1789  0  80   0 - 27092 wait   pts/1    00:00:01 bash
0 S  1011  3637  1841  0  80   0 -   981 hrtime pts/1    00:00:00 stdio_exit
1 Z  1011  3638  3637  0  80   0 -     0 exit   pts/1    00:00:00 stdio_exit <defunct>
1 S  1011  3639     1  0  80   0 -   981 hrtime pts/1    00:00:00 stdio_exit
0 R  1011  3641  1841  0  80   0 - 27033 -      pts/1    00:00:00 ps
[student]$          Petit Fils de PID = 3639   ← affichage du petit fils
[student]$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1011  1841  1789  0  80   0 - 27092 wait   pts/1    00:00:01 bash
0 S  1011  3637  1841  0  80   0 -   981 hrtime pts/1    00:00:00 stdio_exit
1 Z  1011  3638  3637  0  80   0 -     0 exit   pts/1    00:00:00 stdio_exit <defunct>
0 R  1011  3642  1841  0  80   0 - 27035 -      pts/1    00:00:00 ps
[student]$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1011  1841  1789  0  80   0 - 27092 wait   pts/1    00:00:01 bash
0 R  1011  3643  1841  0  80   0 - 27034 -      pts/1    00:00:00 ps
[1]+  Done                    ./stdio_exit   ← annonce de la fin du père
[student]
```

le petit fils a disparu

absence d'affichage du père ?!

Groupe de Processus

- Tout processus appartient à un groupe de processus.



- Un groupe de processus est identifié par un identifiant de groupe **PGID**.
- Dans tout groupe de processus, un processus membre joue un rôle particulier et est appelé leader du groupe.
- Le **PID** du leader du groupe = l'identifiant du groupe **PGID**

Fonctions Relatives aux Groupes

- Tout processus peut connaitre à quel groupe de processus il appartient.

```
#include <unistd.h>
    pid_t getpgid(pid_t pid);
```

PID du processus dont on veut connaitre le groupe auquel il appartient

```
pid_t getpgrp(void);          POSIX
pid_t getpgrp(psid_t pid);    BSD
```

  - L'appel **getpgid()** retourne l'identifiant de groupe auquel appartient le processus d'identifiant **pid**. Si **pid = 0**, alors il s'agit du processus appelant.
  - Valeur de retour
    - l'identifiant de groupe auquel appartient le processus d'identifiant **pid**, en cas de succès,
    - **-1** en cas d'erreur.

- Un processus peut créer un nouveau groupe de processus ou en changer.

```
#include <unistd.h>
    pid_t setpgid(pid_t pid, pid_t pgid);
```

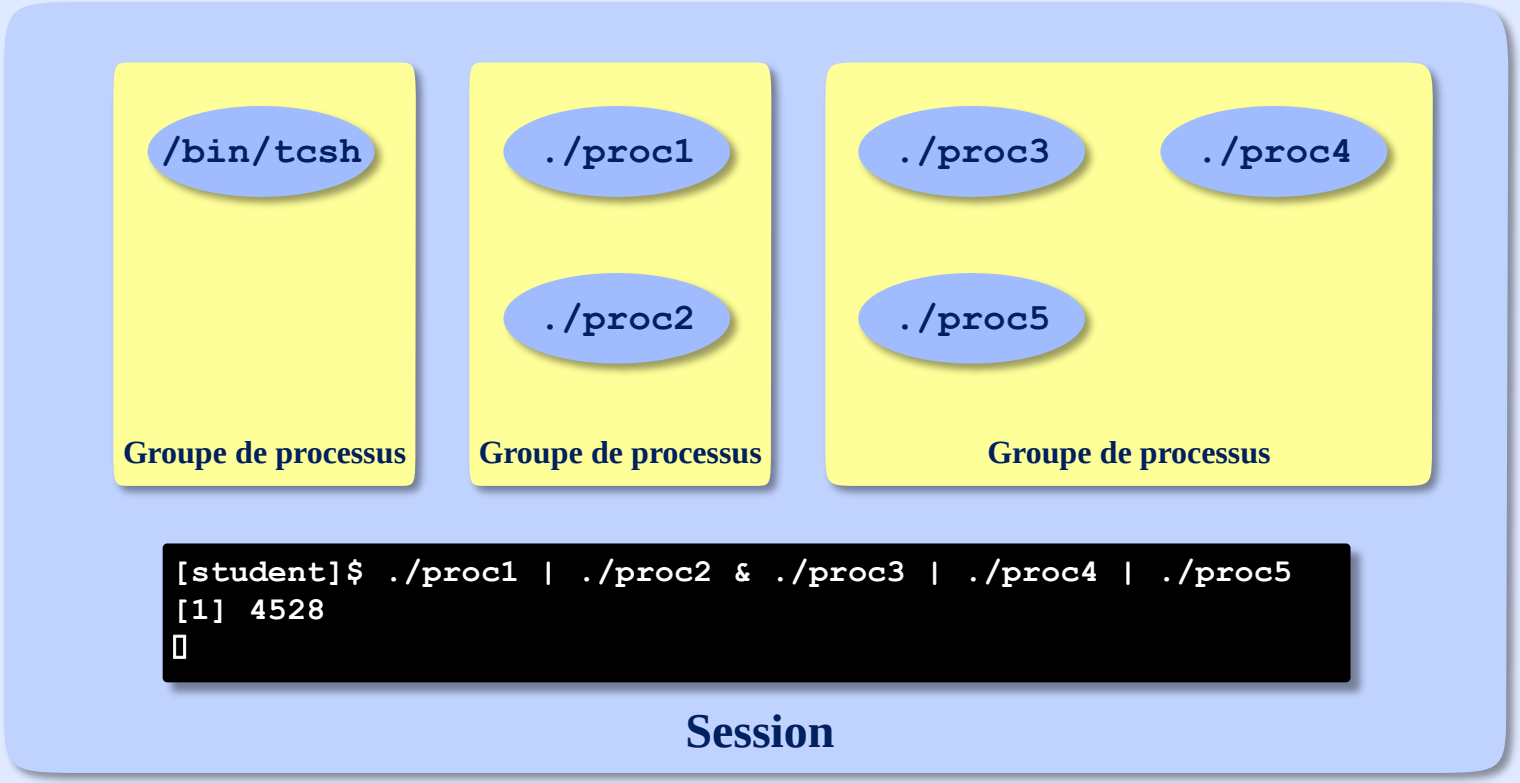
PGID du groupe à rejoindre ou à créer

```
int setpgrp(void);                    System V
int setpgrp(pid_t pid, pid_t pgid);   BSD
```

  - L'appel **setpgid()** fait rejoindre le processus d'identifiant **pid** au groupe d'identifiant **pgid**. Si **pgid = 0**, alors création d'un nouveau groupe ayant comme identifiant le **pid** du processus appelant.
  - Valeur de retour
    - 0 en cas de succès, et **-1** en cas d'erreur.

Sessions

- Tout processus appartient à une session.



- Dans une session, un seul groupe de processus peut être le groupe de processus au premier plan. Les autres groupes de processus sont en arrière-plan.
- Une session peut avoir un terminal de contrôle. Si un signal est généré à partir du terminal, ce signal est émis au groupe de processus au premier plan.

Fonctions Relatives aux Sessions

- Tout processus peut prendre connaissance de l'identifiant de la session à laquelle il appartient.

`PID` du processus dont on veut connaitre la session à laquelle il appartient

```
#include <unistd.h>
    pid_t getsid(pid_t pid);
```

  - Valeur de retour
    - l'identifiant d'une session en cas de succès, `-1` en cas d'échec.

- Tout processus peut créer une nouvelle session.

```
#include <unistd.h>
     pid_t setsid(void);
```

  - Si l'appelant n'est pas un leader de groupe de processus, alors création d'une nouvelle session :
    - le processus devient leader de cette nouvelle session, et il est le seul processus dans cette session,
    - le processus devient leader d'un nouveau groupe de processus, et il est le seul processus dans ce groupe,
    - le processus n'a pas de terminal de contrôle.
  - Valeur de retour
    - l'identifiant de la nouvelle session est retourné, en cas de succès, -1 en cas d'échec.

Exemple : Groupes-Sessions

Programme `proc.c`

```
#include <unistd.h>

int main() {
    pause();
    return 0;
}
```

Exécution de `proc.c`

```
[student]$ gcc proc.c -o proc1
…
[student]$ gcc proc.c -o proc5
[student]$ ./proc1 | ./proc2 & ./proc3 | ./proc4 | ./proc5
[1] 2078
^Z
[2]+  Stopped                 ./proc3 | ./proc4 | ./proc5
[student]$ jobs
[1]-  Running                 ./proc1 | ./proc2 &
[2]+  Stopped                 ./proc3 | ./proc4 | ./proc5
[student]$ bg
[2]+ ./proc3 | ./proc4 | ./proc5 &
[student]$ jobs
[1]-  Running                 ./proc1 | ./proc2 &
[2]+  Running                 ./proc3 | ./proc4 | ./proc5 &
[student]$ ps -lj
F S   UID   PID  PPID  PGID   SID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1011  1710  1680  1710  1710  0  80   0 - 27092 wait   pts/1    00:00:00 bash
0 S  1011  2077  1710  2077  1710  0  80   0 -   980 pause  pts/1    00:00:00 proc1
0 S  1011  2078  1710  2077  1710  0  80   0 -   980 pause  pts/1    00:00:00 proc2
0 S  1011  2079  1710  2079  1710  0  80   0 -   980 pause  pts/1    00:00:00 proc3
0 S  1011  2080  1710  2079  1710  0  80   0 -   980 pause  pts/1    00:00:00 proc4
0 S  1011  2081  1710  2079  1710  0  80   0 -   980 pause  pts/1    00:00:00 proc5
0 R  1011  2084  1710  2084  1710  0  80   0 - 27034 -      pts/1    00:00:00 ps
[student]$
```

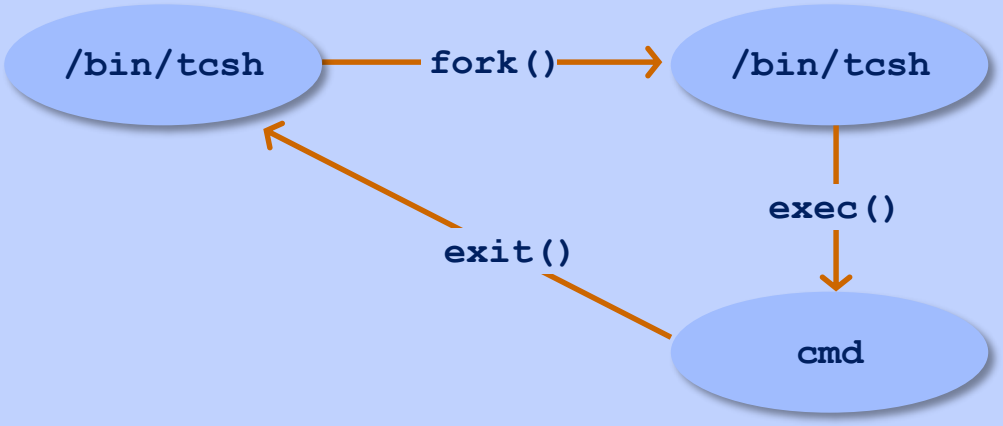
- `[1] 2078` : groupe en arrière-plan
- `^Z` : stoppe le groupe au premier plan
- lancement de 2 groupes de processus, l'un au premier plan et l'autre en arrière-plan
- `[2]+ Stopped ./proc3 | ./proc4 | ./proc5` : groupe (tâche) stoppé(e)
- `[1]- Running` / `[2]+ Stopped` : relancer en arrière-plan la tâche courante

Exemple : Groupes-Sessions

```
[student]$ fg 1                                  ← mettre au premier plan la tâche 1
./proc1 | ./proc2
^Z                                               ← stoppe le groupe au premier plan
[1]+  Stopped                 ./proc1 | ./proc2
[student]$ jobs
[1]+  Stopped                 ./proc1 | ./proc2
[2]-  Running                 ./proc3 | ./proc4 | ./proc5 &
[student]$ bg                                    ← relancer en arrière-plan la tâche courante
[1]+ ./proc1 | ./proc2 &
[student]$ jobs
[1]-  Running                 ./proc1 | ./proc2 &
[2]+  Running                 ./proc3 | ./proc4 | ./proc5 &
[student]$ ^C                                    ← émission du signal TERM depuis le clavier (frappe ^C)
[student]$ jobs
[1]-  Running                 ./proc1 | ./proc2 &          ← les 2 tâches sont "insensibles"
[2]+  Running                 ./proc3 | ./proc4 | ./proc5 &  au signal TERM
[student]$ fg
./proc3 | ./proc4 | ./proc5                      ← groupe s'exécutant au premier plan
^C
[student]$ jobs
[1]+  Running                 ./proc1 | ./proc2 &          ← la tâche s'exécutant au premier plan
[student]$ ^C                                              a été terminée par le signal TERM
[student]$ jobs
[1]+  Running                 ./proc1 | ./proc2 &          ← la tâche en arrière-plan reste
[student]$ fg                                              "insensible" au signal TERM
./proc1 | ./proc2
^C
[student]$ jobs
[student]$ ps -lj
F S   UID   PID  PPID  PGID   SID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1011  1710  1680  1710  1710  0  80   0 - 27092 wait   pts/1    00:00:00 bash
0 R  1011  2085  1710  2085  1710  0  80   0 - 27035 -      pts/1    00:00:00 ps
[student]$
```

Lancement d'un Shell Utilisateur

- Lorsqu'on soumet une commande au **shell**, ce dernier vérifie si :
  - la commande lui est propre (**builtin**), dans ce cas il l'exécute,
  - la commande ne lui est pas propre, dans ce cas il crée un **shell** fils qui demandera l'exécution de la commande au travers d'un **exec**.



Lancement d'une commande **cmd**

Exemple - Écriture d'un Mini-Shell

- Programme **mini_shell.c**

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
```

```c
int main() {
    char cmd[80];
    int etat_fils, pid_fils, i;

    printf("$> ");
    while (fgets(cmd, 79, stdin) != NULL) {
        if (strlen(cmd) >= 2) {
            if ((pid_fils = fork()) != 0) {
                for (i = 0; (cmd[i] != '\0') && (cmd[i] != '&'); i++)
                    ;
                if (cmd[i] != '&') {
                    wait(&etat_fils);
                    printf("Code de retour = %d\n", WEXITSTATUS(etat_fils));
                }
                else {
                    printf("[%d]\n", pid_fils);
                    signal(SIGCHLD, SIG_IGN);
                }
                printf("$> ");
            }
            else {
                for (i = 0; cmd[i] != '\n' && cmd[i] != '&'; i++)
                    ;
                cmd[i] = '\0';
                execlp(cmd, cmd, NULL);
                perror("Erreur a l'execl");
                exit(EXIT_FAILURE);
            }
        }
        else
            printf("$> ");
    }
    exit(EXIT_SUCCESS);
}
```

Exemple - Écriture d'un Mini-Shell

- Exécution de **mini_shell.c**

```
[student]$ gcc mini_shell.c -o mini_shell
[student]$ ./mini_shell
$> ps
  PID TTY          TIME CMD
 1710 pts/1    00:00:00 bash
 2682 pts/1    00:00:00 mini_shell
 2683 pts/1    00:00:00 ps

Code de retour = 0
$> ls
a.out  mini_shell  mini_shell.c  mini_shell.c~

Code de retour = 0
$> ps&
[2685]
$>   PID TTY          TIME CMD
 1710 pts/1    00:00:00 bash
 2682 pts/1    00:00:00 mini_shell
 2685 pts/1    00:00:00 ps

$> ls&
[2686]
$> a.out  mini_shell  mini_shell.c  mini_shell.c~

$> ../Exit/stdio_Exit
           Fils de PID = 2688    Petit Fils de PID = 2689
Code de retour = 0
$> ../Exit/stdio_Exit&
[2690]
$>         Fils de PID = 2691    Petit Fils de PID = 2692
$>
```