TP2

Gilles Menez - UNS - UFR Sciences - Dépt. Informatique

21 septembre 2018

Table des matières

1 Manipulation des processus Unix

Comme UNIX est un système multi-tâches, multi-utilisateurs, il y a toujours un grand nombre de processus qui vivent à un instant donné sur la machine.

Les différents processus sont stockés dans une table et repérés par leur **numéro d'ordre** : Process ID (PID).

On vous rappelle qu'il existe un ensemble de commandes Unix qui permettent de voir et manipuler les processus : `ps` , `pstree` , `top` , `kill` ...

Il est important de pouvoir savoir les activités qui chargent votre machine.

1.1 Commande `ps`

Par défaut (i.e. sans paramètres), elle donne les processus contrôlés par le terminal :

```
/Users/menez/EnseignementsCurrent/Cours_Sockets/TP_L3Miage/TP0

menez@vtr /Users/menez/EnseignementsCurrent/Cours_Sockets/TP_L3Miage/TP0
$ tty
/dev/pts/4

menez@vtr /Users/menez/EnseignementsCurrent/Cours_Sockets/TP_L3Miage/TP0
$ ps
  PID TTY          TIME CMD
13279 pts/4    00:00:00 bash
14171 pts/4    00:00:03 evince
14212 pts/4    00:00:09 emacs
14475 pts/4    00:00:00 ps
```

Ce qui montre que le terminal "/pts/4" est lié à un processus shell (bash), ainsi qu'à trois autres processus : evince, emacs, et bien sûr ps.

✔ Ce lien "terminal-processus" permet, notamment si le processus s'exécute en avant plan, de communiquer avec lui depuis le clavier au travers de la console.

La commande `ps` permet aussi de visualiser les autres processus du systèmes ainsi que beaucoup d'informations les concernant :

```
menez@vtr /Users/menez/EnseignementsCurrent/Cours_Sockets/TP_L3Miage/TP0
$ ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 Jan24 ?        00:00:02 /sbin/init
root         2     0  0 Jan24 ?        00:00:00 [kthreadd]
root         3     2  0 Jan24 ?        00:00:04 [ksoftirqd/0]
root         5     2  0 Jan24 ?        00:00:00 [kworker/u:0]
root         6     2  0 Jan24 ?        00:00:00 [migration/0]
root         7     2  0 Jan24 ?        00:00:00 [watchdog/0]
root         8     2  0 Jan24 ?        00:00:00 [migration/1]
rtkit     1734     1  0 Jan24 ?        00:00:01 /usr/lib/rtkit/rtkit-daemon
root      1824     2  0 Jan24 ?        00:00:03 [flush-8:0]
menez     1870     1  0 Jan24 ?        00:00:00 /usr/bin/gnome-keyring-daemon --daemonize --login
menez     1881  1657  0 Jan24 ?        00:00:01 gnome-session --session=ubuntu
menez     1917  1881  0 Jan24 ?        00:00:00 /usr/bin/ssh-agent /usr/bin/dbus-launch --exit-with-session gnome-session
menez     1920     1  0 Jan24 ?        00:00:00 /usr/bin/dbus-launch --exit-with-session gnome-session --session=ubuntu
menez     1921     1  0 Jan24 ?        00:06:53 //bin/dbus-daemon --fork --print-pid 5 --print-address 7 --session
menez     1932  1881  0 Jan24 ?        00:00:34 /usr/lib/gnome-settings-daemon/gnome-settings-daemon
menez     1940     1  0 Jan24 ?        00:00:00 /usr/lib/gvfs/gvfsd
menez     1942     1  0 Jan24 ?        00:00:00 /usr/lib/gvfs//gvfs-fuse-daemon -f /Users/menez/.gvfs
menez     1949  1881  0 Jan24 ?        00:42:22 compiz
menez     1952     1  0 Jan24 ?        00:00:00 /usr/lib/x86_64-linux-gnu/gconf/gconfd-2
menez     1961     1  0 Jan24 ?        00:00:13 /usr/bin/pulseaudio --start --log-target=syslog
menez     1963     1  0 Jan24 ?        00:00:01 /usr/lib/gvfs/gvfsd-metadata
menez     1964  1881  0 Jan24 ?        00:00:00 /usr/lib/policykit-1-gnome/polkit-gnome-authentication-agent-1
menez     1965  1881  0 Jan24 ?        00:01:33 nautilus -n
```

```
menez    11378  2620  0 09:21 pts/1    00:03:33 emacs intro_socket.tex
menez    11389 11378  0 09:21 ?        00:00:01 /usr/bin/aspell -a -m -B --encoding=utf-8
menez    11394  2620  0 09:22 pts/1    00:00:36 evince master.pdf
menez    11399     1  0 09:22 ?        00:00:00 /usr/lib/evince/evinced
menez    12840     1  0 12:49 ?        00:00:02 unison-2.40.65-gtk
menez    12848 12840  0 12:50 pts/3    00:00:00 ssh seneque.i3s.unice.fr -e none unison -server
menez    13279  2611  0 14:48 pts/4    00:00:00 bash
menez    14171 13279  0 16:46 pts/4    00:00:04 evince master.pdf
menez    14212 13279  0 16:47 pts/4    00:00:18 emacs master.tex
menez    14223 14212  0 16:47 ?        00:00:00 /usr/bin/aspell -a -m -B --encoding=utf-8
root     14465     2  0 17:01 ?        00:00:00 [kworker/3:0]
root     14504     2  0 17:04 ?        00:00:00 [kworker/2:1]
```

La signification des différentes colonnes est la suivante :

- UID nom de l'utilisateur qui a lancé le process
- PID correspond au numéro du process
- PPID correspond au numéro du process parent
- C au facteur de priorité : plus la valeur est grande, plus le processus est prioritaire
- STIME correspond à l'heure de lancement du processus
- TTY correspond au nom du terminal.

  L'occurence d'un ? indique que le processus n'est affecté à aucun terminal (tty). Cela définit en partie un processus démon.
- TIME correspond à la durée de traitement du processus
- COMMAND correspond au nom du processus.

La colonne "S" (si `ps -elf`), indique le mode d'exécution (ou STATUS) du processus. Cet état indique la phase dans laquelle il se trouve :

D	uninterruptible sleep (usually IO)
R	running or runnable (on run queue)
	ce processus dispose du microprocesseur ...
S	interruptible sleep (waiting for an event to complete)
	le status d'attente passive caractéristique des processus démons
T	stopped, either by a job control signal or because it is being traced.
W	paging (not valid since the 2.6.xx kernel)
X	dead (should never be seen)
Z	defunct ("zombie") process, terminated but not reaped by its parent.

Si la présence des crochets vous rend curieux ?

Cliquez !

Essayer la commande presque identique : `ps -e f`

Sur la figure suivante (`ps ax`), le processus correspondant à la commande `ps` dont le PID est 4307 est un processus en cours d'exécution : R+

```
2315 ?        Sl    78:45 /usr/lib/firefox/firefox
2341 ?        Sl     0:00 update-notifier
2348 ?        Sl     2:07 /usr/lib/thunderbird/thunderbird
2472 ?        Sl     0:00 /usr/lib/x86_64-linux-gnu/deja-dup/deja-dup-monitor
2484 ?        S      0:00 /usr/lib/cups/notifier/dbus dbus://
2580 ?        Rl     0:03 gnome-terminal
2589 ?        S      0:00 gnome-pty-helper
2590 pts/1    Ss     0:00 bash
2612 pts/1    Sl     1:03 emacs master.tex
2633 ?        Ss     0:00 /usr/bin/aspell -a -m -B --encoding=utf-8
2637 ?        S      0:03 [kworker/1:2]
2667 ?        Ss     0:00 kdeinit4: kdeinit4 Runnin e
2670 ?        S      0:00 kdeinit4: klauncher [kdei e
2672 ?        Sl     0:00 kdeinit4: kded4 [kdeinit]
2684 ?        Sl     0:00 /usr/bin/kactivitymanagerd
3203 ?        Sl     0:09 evince /Users/menez/EnseignementsCurrent/Cours_Unix/Slides/master.pdf
3210 ?        Sl     0:00 /usr/lib/evince/evinced
3287 ?        Sl     0:09 evince /Users/menez/EnseignementsCurrent/Cours_Sockets/TP_L3Miage/TP0/tp0.p
3423 ?        Sl     0:20 evince /Users/menez/EnseignementsCurrent/Cours_Sockets/Slides/master.pdf
3938 ?        S      0:00 [kworker/2:2]
3939 ?        S      0:02 [kworker/2:3]
3956 ?        S      0:00 [kworker/u8:1]
3981 ?        S      0:01 [kworker/3:0]
4014 ?        S      0:00 [kworker/0:2]
4040 ?        S      0:00 [kworker/u8:2]
4224 ?        S      0:00 [kworker/1:0]
4234 ?        S      0:00 [kworker/u8:0]
4307 pts/1    R+     0:00 ps ax
```

Le suffixe + indique que c'est un processus en avant plan :

<	high-priority (not nice to other users)
N	low-priority (nice to other users)
L	has pages locked into memory (for real-time and custom IO)
s	is a session leader
l	is multi-threaded (using CLONE_THREAD, like NPTL pthreads do)
+	is in the foreground process group

1.2 Commande pstree

Les processus sont créés par filiation :

- ➢ Au lancement du système, le processus 0 est lancé, il crée le processus 1 qui prend la main.
- ➢ Celui-ci crée d'autres processus fils qui héritent de certaines propriétés de leur père.
  Par exemple, liste des descripteurs de fichiers incluant les fichiers ouverts, les sockets ...
- ➢ Certains processus sont créés, exécutent leur tâche et meurent, le numéro qui leur était attribué disparaît.
  Ceci explique pourquoi la numérotation obtenue par la commande `ps` n'est pas nécessairement continue.

La commande `pstree` illustre la descendance du processus 1 et celles de ses fils :

```
~> pstree -A -c -p

systemd(1)-+-ModemManager(466)-+-{gdbus}(489)
           |                   `-{gmain}(487)
           |-NetworkManager(457)-+-{gdbus}(493)
           |                     `-{gmain}(491)
           |-accounts-daemon(459)-+-{gdbus}(480)
           |                      `-{gmain}(474)
           |-apache2(547)-+-apache2(4309)-+-{apache2}(4313)
           |              |               |-{apache2}(4314)
           |              |               |-{apache2}(4315)
           |              |               |-{apache2}(4316)

           |              |-gnome-terminal-(3477)-+-bash(3483)-+-emacs(3548)-+-aspell(3588)
           |              |                       |            |             |-{dconf worker}(3555)
           |              |                       |            |             |-{gdbus}(3551)
           |              |                       |            |             `-{gmain}(3550)
           |              |                       |            |-okular(3629)---{QProcessManager}(3638)
           |              |                       |            `-pstree(1669)
```

2 Création des processus Unix

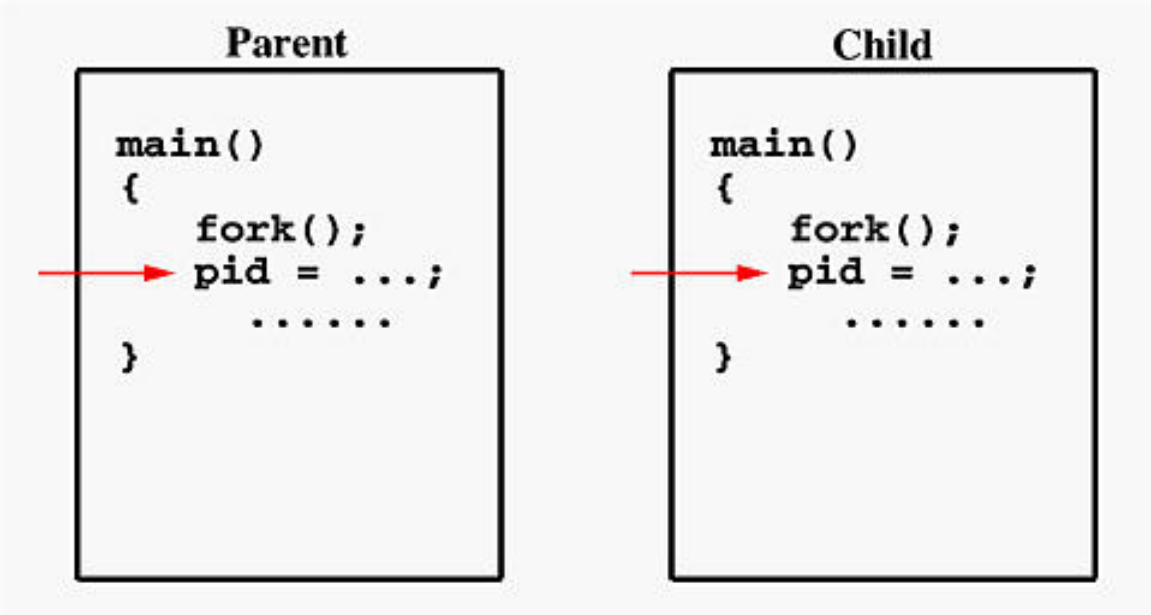
Tout processus UNIX, excepté le premier, est crée par un mécanisme unique au travers de cette primitive `fork` .

2.1 Primitive fork()

Cette primitive est donc utilisée pour créer un nouveau processus (le fils) en **dupliquant le processus actuel** (le père) : l'espace d'adressage (code, données, pile) et une grosse partie du bloc de contrôle.

Dès que le fork() est exécuté le processus père et le processus fils exécute l'instruction suivante.

➢ Forcément c'est la même instruction puisque c'est le même code !



2.1.1 Exemple 0 :

```c
/* ------------------------------------------------------------------
* file : fork0.c
* This program illustrates the use of fork() and
* getpid() system calls.

* Note that write() is used instead of * printf() since the latter
* is buffered while the former is not.
*
------------------------------------------------------------------ */
#include  <stdio.h>
#include  <string.h>
#include <unistd.h>
#include  <sys/types.h>

#define   MAX_COUNT  100
#define   BUF_SIZE   50

int main(void){
```

```
  pid_t  pid;
  int    i;
  char   buf[BUF_SIZE];

  fork();
  pid = getpid();

  for (i = 1; i <= MAX_COUNT; i++) {
    sprintf(buf, "This line is from pid %d, value = %d\n", pid, i);
    write(1, buf, strlen(buf)); /* write pour eviter la bufferisation */
  }
}
```

① Comprendre et Expérimenter ce code.

```
$ gcc fork0.c

menez@vtr /Users/menez/EnseignementsCurrent
$ ./a.out
This line is from pid 7543, value = 1
This line is from pid 7543, value = 2
This line is from pid 7543, value = 3
This line is from pid 7543, value = 4
This line is from pid 7543, value = 5
This line is from pid 7543, value = 6
This line is from pid 7543, value = 7
This line is from pid 7543, value = 8
This line is from pid 7543, value = 9
This line is from pid 7543, value = 10
This line is from pid 7544, value = 1
This line is from pid 7544, value = 2
This line is from pid 7544, value = 3
This line is from pid 7544, value = 4
This line is from pid 7544, value = 5
This line is from pid 7544, value = 6
This line is from pid 7544, value = 7
This line is from pid 7544, value = 8
This line is from pid 7544, value = 9
This line is from pid 7544, value = 10
```

Vous remarquez que les deux processus écrivent sur le même terminal. C'est la preuve qu'ils partagent des caractéristiques communes.

② Augmenter d'un facteur 10 le nombre d'itérations (MAX_COUNT).
Qu'est-ce qui se passe ?

③ Vous obtenez un résultat/affichage ? Pouvez vous garantir que tout lancement à venir produira le même résultat ?

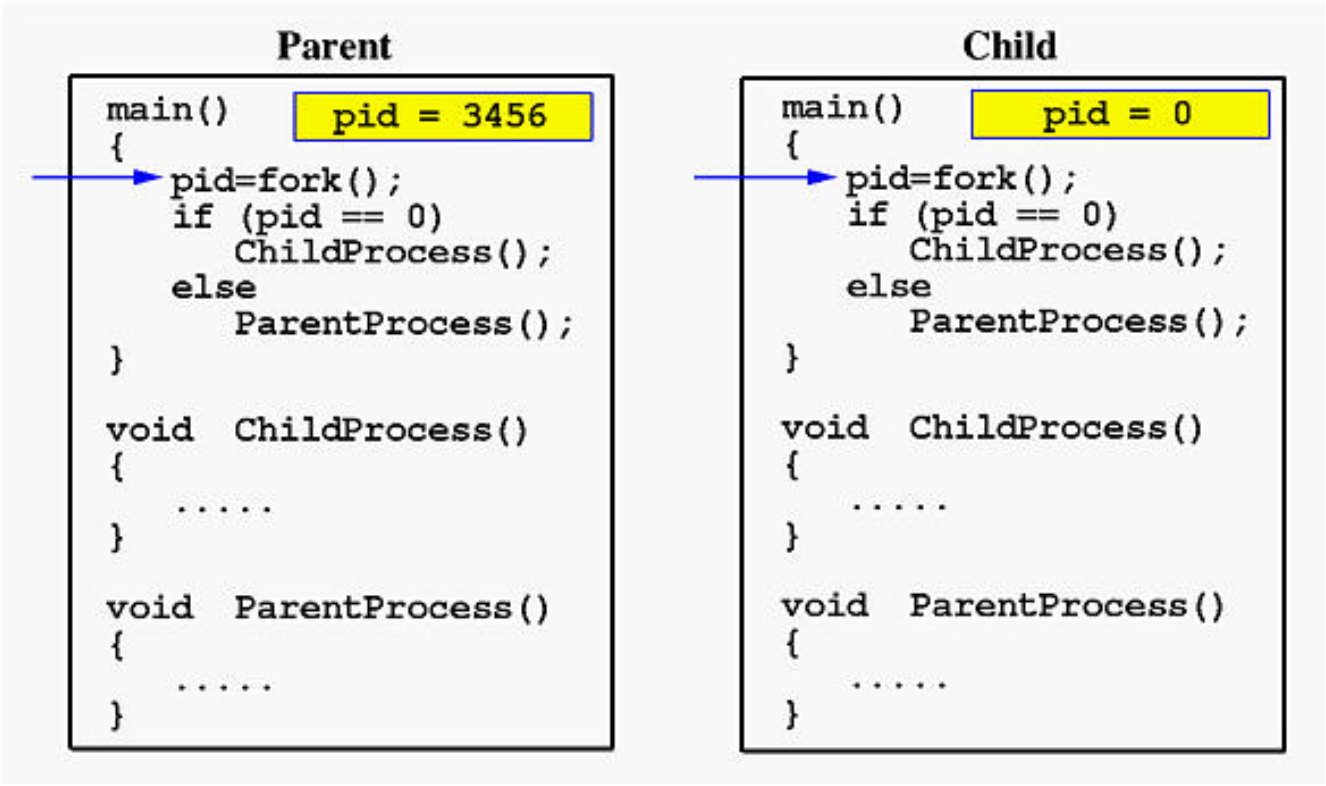
④ Une fois que vous avez compris que c'est l'ordonnanceur (scheduler) du système Linux qui décide de l'ordre d'exécution, est-ce que vous pouvez proposer un critère qui selon vous est utilisé par cet ordonnanceur pour prendre ses décisions ?
Si vous ne le savez pas, vous pouvez le deviner : quel critère vous paraîtrait "juste"/efficace/... ?

2.2 Codage différencié

Pour distinguer le père du fils, et pouvoir différencier leurs activités (à partir d'un même code), on utilise la valeur retournée par `fork()` .

➢ Sur la base de ce test, on pourra choisir entre des activités !

➢ On pourra aussi modifier l'environnement des processus.



Ainsi, si `fork()` :

- ✔ retourne une valeur négative, la création du fils a échoué.
- ✔ retourne la valeur 0 c'est qu'on est dans le processus fils
- ✔ retourne une valeur positive. C'est le PID du processus fils.

2.2.1 Exemple 1 :

```
/* ------------------------------------------------------------------
* PROGRAM fork1.c
*  This program runs two processes, a parent and a child.
* Note that printf() is used in this program for simplicity. */
#include<stdio.h>
#include<sys/types.h>
#define   MAX_COUNT  200
void  ChildProcess(void);          /* child process prototype  */
void  ParentProcess(void);        /* parent process prototype */
int  main(void){
     pid_t  pid;
     pid = fork();
     if (pid == 0)
          ChildProcess();
```

```
        else
                ParentProcess();
        return 0;
}
void  ParentProcess(void){
        int   i;
        for (i = 1; i <= MAX_COUNT; i++)
                printf("This line is from parent, value = %d\n", i);
        printf("*** Parent is done ***\n");
}
void  ChildProcess(void){
        int   i;
        for (i = 1; i <= MAX_COUNT; i++)
                printf("   This line is from child, value = %d\n", i);
        printf("   *** Child process is done ***\n");
}
```

① Comprendre et Expérimenter ce code.

```
This line is from parent, value = 41
This line is from parent, value = 42
This line is from parent, value = 43
This line is from parent, value = 44
This line is from parent, value = 45
This line is from parent, value = 46
This line is from parent, value = 47
This line is from parent, value = 48
This line is from parent, value = 49
This line is from parent, value = 50
This line is from parent, value = 51
This line is from parent, value = 52
   This line is from child, value = 1
This line is from parent, value = 53
   This line is from child, value = 2
This line is from parent, value = 54
   This line is from child, value = 3
This line is from parent, value = 55
   This line is from child, value = 4
This line is from parent, value = 56
   This line is from child, value = 5
This line is from parent, value = 57
   This line is from child, value = 6
This line is from parent, value = 58
   This line is from child, value = 7
```

3 Terminaison des processus

Vous avez appris à créer des processus par la programmation mais il est important de savoir les terminer correctement !

Cette opération vitale, si on veut éviter de surcharger la machine, nécessite une parfaite maîtrise des différentes problématiques qui lui sont associées.

➤ Même si ce n'est que pour optimiser quelques entrées dans une table, n'oubliez pas qu'un serveur tourne 24h/24 et 7j/7.

Si il laisse s'accumuler des ressources inutiles en mémoire, les performances de la machines vont, à force, en pâtir.

Ce n'est pas pour rien que l'on parle de fuites mémoires (peut être plus dans le cas de la programmation dans le tas/heap, mais le phénomène est similaire) : on fait des mers avec des gouttes !

3.1 Processus orphelin

Normalement, **un processus père doit attendre la terminaison de ses fils avant de se terminer lui-même** ;

➢ sinon, on crée des processus "orphelins".

Ces processus orphelins sont automatiquement **adoptés** par le processus "grand père" (déclaré reaper ou subreaper via `prctl(2)` : reaper = moissonneur) le plus proche.

`http://unix.stackexchange.com/questions/250153/what-is-a-subreaper-process`

En soit, cette adoption est plutôt une bonne chose (cela évite de générer un zombie) mais encore faut-il que le processus ainsi adopté est un rôle souhaité et qu'il ne s'agisse pas d'un oubli.

```c
/** Fichier orphelin.c :  Faire mourir le pere avant le fils */
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
int main (int argc, char *argv[]){
  int pid, ppid;
  switch(pid = fork()) {
    case -1:   /* Pb... */
      perror("Erreur du fork");
      exit(1);
    case 0:    /* Ce code s'execute chez le fils */
      ppid = getppid();
      printf("Je suis le fils (%d) et le PID de mon pere est (%d)\n",
                getpid(), ppid);
      printf("Vous avez 15 sec pour lancer un ""ps -e -f"" !\n");
      /* Il s'agit de montrer le pere et le fils en cours d'execution */

      printf("\nJe suis le fils (%d) et le PID de mon pere est (%d)\n",
                getpid(), getppid());
      sleep(40);
      if (getppid() != ppid)
        printf("Je suis donc devenu orphelin !\n"); /* le ppid a change */
      break;
  default:   /* Ce code s'execute chez le pere */
    printf("Je suis le pere (%d) \n",getpid());
    sleep(15); /* Pour laisser le temps au fils d'afficher le premier ppid */
    printf("Je suis le pere (%d) et je meurs AVANT mon fils (%d)\n",getpid(),pid);
   }
  printf("Fin \n");
```

```
  exit(0);
}
```

(a) Comprendre au niveau du code pourquoi il y a création d'un orphelin.

(b) Tester le code et faire afficher au niveau de la console les résultats de la commande `ps` montrant les deux phases :

① Le père et le fils en cours d'exécution.

② Le fils devenu orphelin après la mort du père.

3.2 Processus zombie

Si maintenant un processus fils se termine et que son père ne consulte pas son statut (par un appel à wait(2) ou assimilé), le processus fils est bien détruit de la liste des processus en cours d'exécution (puisqu'il s'est terminé), **mais son PID apparaît encore dans la table des processus du système**.

➟ Ce processus est dit "zombie" : il ne peut pas être tué (n'existant plus), mais il continue d'occuper une entrée dans la table et son PID ne peut pas être réutilisé par un autre processus.

➟ On comprend que ce statut "encombre" le système et occupe inutilement de la ressource (cf cours).

➟ Lorsque le père meurt, ses fils zombies disparaissent aussi. Mais si le père est un serveur, il ne devrait pas mourir !

```c
/** Fichier zombie.c : Faire mourir le fils avant le pere.
    Et ce dernier ne s'en preoccupe pas ! */
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
int main (int argc, char *argv[]){
  int pid;
  switch(pid = fork()) {
  case -1:   /* Pb... */
    perror("Erreur du fork");
    exit(1);
  case 0:    /* Ce code s'execute chez le fils */
    printf("Je suis le fils : PID %d et le PID de mon pere est %d\n",
           getpid(), getppid());
    printf("Je suis le fils et je meurs : PID %d\n",
           getpid());
    break;
  default:   /* Ce code s'execute chez le pere */
    printf("\nJe suis le pere : PID %d\n", getpid());
    printf("Qu'est devenu mon fils %d ?\n", pid);
    printf("Vous avez 30 sec pour lancer un ""ps -e -f"" et constater qu'il est zombi !\n");
    sleep(30);
    printf("Je suis le pere, et je meurs : PID %d\n",getpid());
  }
  printf("\n");
  exit(0);
}
```

(a) Comprendre au niveau du code pourquoi il y a création d'un zombie.

(b) Tester le code et faire afficher au niveau de la console les résultats de la commande `ps` montrant les deux phases :

① Le père et le fils en cours d'exécution.

② Le fils devenu zombie après sa mort.

(c) Montrer aussi qu'il est impossible de tuer le zombie.

(d) Tuer le père et voir le zombie disparaître.

3.3 Primitive wait()

Pour éviter la création de zombies, **un processus père doit donc toujours récupérer le statut de ses fils** !

Un processus père peut attendre la terminaison de son fils par un appel à la primitive `wait(int *status)` (cf wait(2)) qui renverra le PID du fils qui s'est terminé, ou -1 en cas d'erreur.

- ➟ Le paramètre status contient le code de retour de ce processus fils (celui qu'il a renvoyé par exit).
- ➟ Le processus appelant wait est bloqué jusqu'à ce qu'un fils se termine.

Notes :

- ✔ La fonction POSIX `waitpid(2)` est plus générale et permet de passer des options. Pour l'instant, on utilise `wait()` .
- ✔ La macro `WEXITSTATUS` extrait le code de sortie du processus fils.
- ✔ La macro `WIFEXITED` pour déterminer si un processus s'est terminé correctement à partir de son code de statut (via la fonction exit ou la sortie de main) ou est mort à cause d'un signal non intercepté.
- ✔ Dans ce dernier cas, utilisez la macro `WTERMSIG` pour extraire le numéro du signal ayant causé la mort du processus à partir du code de statut.

```c
/** Fichier wait.c :   création/terminaison propre d'un processus :
    a) On crée un fils.
    b) Le père attend la fin de celui-ci en testant la condition de sa fin.

    On peut simuler une mauvaise fin en effectuant un 'kill' du
    processus fils.  */
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
int main(int argc, char *argv[]) {
  int exit_cond;
  pid_t pid;
  pid = fork();
  switch (pid) {
  case -1 : perror("Erreur de création du processus");
    exit(1);
  case 0 : /* Ce code s'exécute chez le fils */
    printf("Pid du fils = %d\n", getpid());
    sleep(20); /* Duree de vie du fils */
    break;
  default : /* Ce code s'exécute chez le père */
    printf("Pid du pere = %d\n", getpid());
    printf("Attente de la terminaison du fils...\n");
    pid = wait(&exit_cond);
    if (WIFEXITED(exit_cond))
      printf("Le fils %d s'est termine correctement : %d\n",
             pid, WEXITSTATUS (exit_cond));
    else
```

```
        printf("Le fils %d s'est mal termine : %d\n", pid, WTERMSIG (exit_cond));
  } /* switch */
  exit(0);  /* exécuté par le fils et le père */
}
```

(a) Tester le code et montrer au niveau de la console deux cas :

① Le fils meurt "naturellement",

② Le fils meurt suite à la réception d'un signal "9"

(b) Faire évoluer ce code pour générer deux (ou plusieurs) processus fils à partir du même père.

Attention, il ne s'agit pas de générer un père, son fils et un petit fils !

(c) Ensuite l'idée, c'est de montrer que la primitive `wait()` attend la première terminaison et que le problème du zombie se pose pour le deuxième fils.

➢ A moins de boucler sur les "wait" autant de fois que l'on a bouclé sur les "fork" !

4 Gestion asynchrone de la terminaison

4.1 Signal SIGCHLD

Quand un processus fils se termine ou est stoppé, il envoie le signal SIGCHLD à son père.
Le père peut donc utiliser ce signal pour s'interrompre, récupérer le statut du fils ayant émis le signal et ainsi éviter qu'il soit zombie en faisant un wait() dessus.

➢ On peut ainsi gérer de façon asynchrone la terminaison des fils.

```c
/** Fichier : sigchld.c. Gestion asynchrone des terminaisons */
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

void eliminer_zombie(int sig){
  /* Handler du signal SIGCHLD */
  int exit_cond;
  pid_t pid;
  printf("Attente de la terminaison du fils...\n");
  pid = wait(&exit_cond);
#if 0
  /* Utilisation alternative d'une gestion non bloquante du wait */
  while(waitpid (-1, &sexit_cond, WNOHANG));
#endif
  if (WIFEXITED(exit_cond))
    printf("Le fils %d s'est termine correctement : %d\n", pid,
           WEXITSTATUS (exit_cond));
  else
    printf("Le fils %d s'est mal termine : %d\n", pid,
           WTERMSIG(exit_cond));
}

int main(int argc, char *argv[]) {
  struct sigaction action;
  int i;
  pid_t pid;

  for (i=0; i<2 ;i++){
    pid = fork();

    switch (pid) {
    case -1 : perror("Erreur de création du processus");
      exit(1);
    case 0 : /* on est chez le fils */
      printf("Pid du fils = %d\n", getpid());
      if (i==0)
        sleep(5); /* Le premier fils meurt au bout de 5 secondes */
      if (i==1)
        sleep(10); /* Le deuxieme fils meurt au bout de 20 secondes */
      exit(0);
    } /* switch */
  } /* for */

  /* on est forcement chez le pere */
  printf("\nPid du pere = %d\n", getpid());

  /* Attachement du handler du signal SIGCHLD */
  memset (&action, 0, sizeof (action));
```

```
  action.sa_handler=eliminer_zombie; /* Fonction handler */
  sigaction(SIGCHLD, &action, NULL);   /* Attachement de l'action au signal */

  /* Le pere peut desormais faire ce qu'il veut en concurrence de ses
     fils */

  /* Ce n'est pas tres propre mais je mets ici autant d'actions
     qu'il y a de fils a attendre car chaque fonction sleep est
     interrompue par l'execution asynchrone du handler */
  sleep(20);
  sleep(20);

  printf("Mort du pere !\n");
  exit(0);  /* execute par le pere */
}
```

(a) Comprendre l'intérêt de l'utilisation du signal `SIGCHLD` .
Vous notez la petite contrainte liée à l'effet de l'exécution asynchrone du handler sur la fonction en cours d'exécution : ici sleep() mais cela serait pareil avec un gets() ou tout autre fonction "bloquante".

(b) Tester le code.