

Cours n°4 : Création de processus

Victor Poupet

5 Mars 2018



Hiérarchie de processus

\$ ps ax

PID	PPID	S	COMMAND
1	0	S	/sbin/init
338	1	S	/sbin/udevd --daemon
481	338	S	/sbin/udevd --daemon
1200	880	S	lightdm --session-child
1301	1200	S	gnome-session
1528	1	S	/usr/lib/gvfs/gvfsd
1532	1301	S	nm-applet
1556	1301	S	/usr/lib/gnome-settings
1590	1	S	/usr/lib/gvfs/gvfs-gdu
1627	1	S	/usr/lib/bamf/bamfd daemon
1636	1635	S	/usr/bin/gtk-window
1639	1	S	/usr/lib/unity/unity
1845	1	S	/usr/lib/gnome-online
1853	1	R	gnome-terminal
2414	1	S	/usr/lib/dconf/dconf
2431	1853	S	bash
2597	2431	R	ps ax

Au démarrage de l'ordinateur, le processus **init** est lancé

- Par la suite, tous les autres processus sont démarrés par un processus existant (parent)
- Chaque processus est identifié par un numéro unique (PID) et connaît le numéro de son parent (PPID)
- La commande **ps** permet d'obtenir le PID et le PPID d'un processus

Hierarchie (pstree)

```
$ pstree -p
```

```
init(1)─NetworkManager(793)─dhclient(911)
      │                      └─dnsmasq(1012)
      │
      ├─accounts-daemon(1214)─{accounts-daemon}(1215)
      │
      ├─cron(855)
      │
      ├─cupsd(729)─dbus(3916)
      │
      ├─dconf-service(1997)─{dconf-service}(1998)
      │                     └─{dconf-service}(2000)
      │
      ├─gnome-keyring-d(1407)─{gnome-keyring-d}(1408)
      │                     └─{gnome-keyring-d}(1970)
      │
      ├─gnome-terminal(1737)─bash(1746)─pstree(20545)
      │                     │
      │                     ├─gnome-pty-helpe(1745)
      │                     └─{gnome-terminal}(1747)
      │
      ├─goa-daemon(1895)─{goa-daemon}(1911)
      │
      ├─gvfs-afc-volume(1612)─{gvfs-afc-volume}(1614)
      │
      ├─mission-control(1890)─{mission-control}(1893)
      │                     └─{mission-control}(1909)
      │
      ├─sh(884)─initctl(886)
      │
      └─udisks-daemon(1604)─udisks-daemon(1609)
```

Création

```
int main(int argc, char **argv) {  
    pid_t pid = fork();  
    if (pid == -1) {  
        perror("fork");  
    } else if (pid == 0) {  
        for (int i=0; i<3; i++) {  
            printf("F%d\n", i);  
        }  
    } else {  
        for (int i=0; i<3; i++) {  
            printf("P%d ", i);  
        }  
    }  
}
```

```
$ ./a.out
```

```
P0 P1 P2 F0 F1 F2
```

Pour créer un nouveau processus, on clone un processus existant à l'aide de l'appel `fork()`

■ Le système copie :

- la mémoire (pile, tas, code)
- les descripteurs de fichiers ouverts (cependant les pointeurs dans les fichiers sont partagés)
- l'état d'exécution (pointeur d'exécution, registres du processeur)

■ Différences entre les processus :

- le processus fils a un PID et un PPID différent du père (son PPID est le PID du père)
- la fonction `fork` renvoie 0 dans le processus fils, et le PID du fils dans le processus père

Création

```
// fork bomb !  
int main(int argc, char **argv) {  
    while (1) fork();  
}
```



Pour créer un nouveau processus, on clone un processus existant à l'aide de l'appel `fork()`

- Le système copie :
 - la mémoire (pile, tas, code)
 - les descripteurs de fichiers ouverts (cependant les pointeurs dans les fichiers sont partagés)
 - l'état d'exécution (pointeur d'exécution, registres du processeur)
- Différences entre les processus :
 - le processus fils a un PID et un PPID différent du père (son PPID est le PID du père)
 - la fonction `fork` renvoie 0 dans le processus fils, et le PID du fils dans le processus père

Attente de complétion

SYNOPSIS

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

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

```
pid_t wait(int *status);
```

```
pid_t waitpid(pid_t pid, int  
*status, int options);
```

Les commandes `wait` et `waitpid` demandent à un processus d'attendre que ses processus fils changent d'état

- Lorsqu'un fils se termine, la commande `wait` permet au parent de recevoir le code de retour du fils
- Le fils est supprimé de la table de processus quand son père reçoit le code de retour

Zombies

```
int main(int argc, char **argv) {  
    pid_t pid;  
    pid = fork();  
    if (pid == 0) { // fils  
        sleep(10);  
        printf("Fin fils\n");  
    } else { // père  
        sleep(20);  
        printf("Fin père\n");  
    }  
    exit(1);  
}
```



Lorsqu'un processus se termine, il n'est pas immédiatement supprimé de la table

- Le parent doit explicitement le supprimer à l'aide de l'appel `wait`
- Si le processus reste, il devient un *zombie*
- Les processus zombies n'occupent pas de mémoire, mais ils bloquent un PID (et sont souvent le signe d'un bug dans le processus parent)
- Si le parent est terminé avant le fils, le fils devient orphelin et est récupéré par `init` (qui appelle `wait` régulièrement)

Zombies

```
$ ps
```

PID	TTY	STAT	COMMAND
1859	pts/0	Ss	bash
2570	pts/0	S	./prog
2571	pts/0	Z	[prog] <defunct>



Lorsqu'un processus se termine, il n'est pas immédiatement supprimé de la table

- Le parent doit explicitement le supprimer à l'aide de l'appel `wait`
- Si le processus reste, il devient un *zombie*
- Les processus zombies n'occupent pas de mémoire, mais ils bloquent un PID (et sont souvent le signe d'un bug dans le processus parent)
- Si le parent est terminé avant le fils, le fils devient orphelin et est récupéré par `init` (qui appelle `wait` régulièrement)

Éviter les zombies

```
int main() {  
    int pid1;  
    pid1 = fork();  
    if (pid1) {  
        // père  
        /* Tâche principale */  
    } else {  
        // fils  
        /* Tâche secondaire */  
    }  
}
```

Parfois, la tâche exécutée par le fils est longue et on ne veut pas bloquer le père en attente de complétion

- On peut mettre en place une gestion de signaux (SIGCHLD)
- On peut faire en sorte que le processus fils soit immédiatement adopté par `init`

Éviter les zombies

```
static void handler(int signo) {  
    wait(NULL);  
}  
  
int main() {  
    signal(SIGCHLD, handler);  
    int pid1;  
    pid1 = fork();  
    if (pid1) {  
        // père  
        /* Tâche principale */  
    } else {  
        // fils  
        /* Tâche secondaire */  
    }  
}
```

Parfois, la tâche exécutée par le fils est longue et on ne veut pas bloquer le père en attente de complétion

- On peut mettre en place une gestion de signaux (SIGCHLD)
- On peut faire en sorte que le processus fils soit immédiatement adopté par `init`

Éviter les zombies

```
int main() {  
    int pid1, pid2;  
    pid1 = fork();  
    if (pid1) {  
        // père  
        waitpid(pid1, NULL, 0);  
        /* Tâche principale */  
    } else {  
        // fils  
        pid2 = fork();  
        if (pid2) {  
            // fils  
            exit(0);  
        } else {  
            // petit- fils  
            /* Tâche secondaire */  
        }  
    }  
}
```

Parfois, la tâche exécutée par le fils est longue et on ne veut pas bloquer le père en attente de complétion

- On peut mettre en place une gestion de signaux (SIGCHLD)
- On peut faire en sorte que le processus fils soit immédiatement adopté par `init`

fork

La fonction **fork** est appelée par un processus mais renvoie deux résultats, dans deux processus distincts

- Le père reçoit le PID du fils
- Le fils peut obtenir le PID du père : **getppid()**
- Si le fils et le père doivent avoir des comportements différents, il faut que les deux aient l'ensemble du code

fork

La fonction **fork** est appelée par un processus mais renvoie deux résultats, dans deux processus distincts

- Le père reçoit le PID du fils
- Le fils peut obtenir le PID du père : **getppid()**
- Si le fils et le père doivent avoir des comportements différents, il faut que les deux aient l'ensemble du code... sauf s'il existe un moyen de changer le code d'un processus

Exécution

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

```
extern char **environ;
```

```
int execl(const char *path, const  
char *arg, ...);
```

```
int execlp(const char *file, const  
char *arg, ...);
```

```
int execlp(const char *path, const  
char *arg, ..., char * const  
envp[]);
```

```
int execv(const char *path, char  
*const argv[]);
```

```
int execvp(const char *file, char  
*const argv[]);
```

La famille de fonctions exec permet de remplacer le code d'un processus par un autre

- le premier argument est un exécutable
- les arguments suivants permettent de passer des paramètres et des variables d'environnement
- à l'appel de la fonction, tout le code du processus est remplacé par celui indiqué et l'état du processus est réinitialisé

Exécution

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

```
extern char **environ;
```

```
int execl(const char *path, const  
char *arg, ...);
```

```
int execlp(const char *file, const  
char *arg, ...);
```

```
int execlx(const char *path, const  
char *arg, ..., char * const  
envp[]);
```

```
int execv(const char *path, char  
*const argv[]);
```

```
int execvp(const char *file, char  
*const argv[]);
```

Il existe plusieurs variantes :

- *l* (*list*) : les arguments sont passés un par un à la fonction, avec un pointeur nul (**NULL**) en dernier
- *v* (*vector*) : les arguments sont passés dans un unique tableau
- *p* (*path*) : l'exécutable est cherché dans les répertoires du chemin d'exécution
- *e* (*environment*) : permet de passer un tableau contenant des variables d'environnement pour l'exécution du nouveau programme

Exécution

```
int main(int argc, char **argv) {  
    pid_t pid;  
    int status;  
    pid = fork();  
    if (pid == 0) {  
        execlp("ls", "ls", "-l", NULL);  
    } else {  
        printf("Père \n");  
        wait(&status);  
    }  
}
```

Il existe plusieurs variantes :

- *l* (*list*) : les arguments sont passés un par un à la fonction, avec un pointeur nul (**NULL**) en dernier
- *v* (*vector*) : les arguments sont passés dans un unique tableau
- *p* (*path*) : l'exécutable est cherché dans les répertoires du chemin d'exécution
- *e* (*environment*) : permet de passer un tableau contenant des variables d'environnement pour l'exécution du nouveau programme

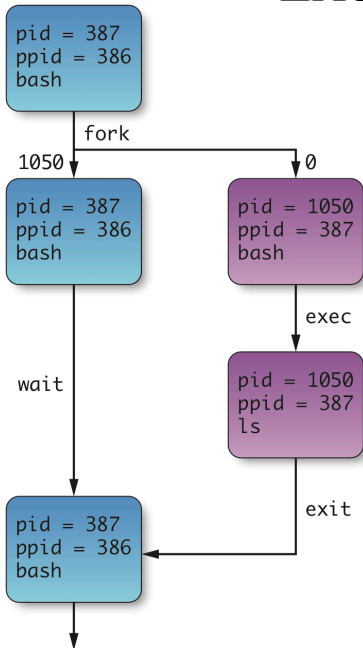
Exécution

```
int main(int argc, char **argv) {
    pid_t pid;
    int status;
    pid = fork();
    if (pid == 0) {
        char *args[3];
        args[0] = "ls";
        args[1] = "-l";
        args[2] = NULL;
        execvp("ls", args);
    } else {
        printf("Père \n");
        wait(&status);
    }
}
```

Il existe plusieurs variantes :

- *l* (*list*) : les arguments sont passés un par un à la fonction, avec un pointeur nul (**NULL**) en dernier
- *v* (*vector*) : les arguments sont passés dans un unique tableau
- *p* (*path*) : l'exécutable est cherché dans les répertoires du chemin d'exécution
- *e* (*environment*) : permet de passer un tableau contenant des variables d'environnement pour l'exécution du nouveau programme

Exemple : ls



Le shell veut exécuter la commande `ls`

- `fork()` pour créer le processus fils
- le père attend que le fils ait fini son exécution (si la commande est en arrière plan le père ne se bloque pas)
- le fils appelle `exec` pour remplacer son code par celui du programme `ls`
- quand le fils termine, le père reçoit le code de retour, supprime le processus fils et reprend son exécution

Copie sur écriture

L'utilisation de **fork/exec** est le moyen le plus classique (et parfois le seul disponible) pour créer des nouveaux processus

- Si le nouveau processus va être remplacé par un autre programme (**exec**) il est inutile de copier toute sa mémoire au moment du **fork**
- Pour alléger, on utilise souvent une technique appelée *copy on write* :
 - lorsque le nouveau processus est créé, on ne copie pas sa mémoire
 - il dispose de pointeurs vers la mémoire du père pour la lecture
 - la mémoire est copiée au moment de la première écriture par le fils
- Dans le cas où **exec** suit **fork**, la mémoire n'est pas copiée