

Processus

François Trahay



Contents

Caractéristiques d'un processus	1
Création de processus	2
fork	3
La classe de fonctions <code>exec</code>	4
Terminaison d'un processus	5

Caractéristiques d'un processus

- **PID** (*Process Identifier*) : identifiant unique du processus
 - `pid_t getpid()`;
 - retourne le PID du processus courant
- **PPID** (*Parent PID*) : identifiant du processus père
 - `pid_t getppid()`;
 - retourne le PPID du processus courant

Voici un exemple de programme affichant son PID et son PPID:

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

int main(int argc, char**argv) {
    printf("Current process ID: %d\n", getpid());
    printf("Current parent process ID: %d\n", getppid());
    return EXIT_SUCCESS;
}
```

Ce programme donne pour résultat:

```
$ ./print_id
Current process ID: 17174
```

Current parent process ID: 25275

Lorsque le processus parent (P1) d'un processus (P2) meurt, le processus fils est rattaché au processus au processus initial de PID 1. Le PPID de P2 devient donc 1.

Sur certains systèmes, le processus `systemd` se charge d'adopter les processus orphelins. Dans ce cas, le PPID de P2 devient le PID de `systemd`.

Création de processus

- `int system(const char* cmd);`
 - Crée un processus shell qui exécute `cmd`
 - Retourne le code de retour de la commande (0 si tout s'est bien passé)

Voici un exemple de programme utilisant la fonction `system`:

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

int main(int argc, char**argv) {
    int ret_val;
    char* cmd="ps f";
    printf("Running command '%s'\n", cmd);
    printf("-----\n");
    ret_val = system(cmd);
    printf("-----\n");
    printf("system returned %d\n", ret_val);
    return EXIT_SUCCESS;
}
```

Ce programme donne pour résultat:

```
$ ./system
Running command 'ps f'
-----
  PID TTY          STAT TIME COMMAND
16847 pts/1        Ss+  0:00 bash
17076 pts/1        S    0:01 _ okular chap.pdf
 8199 pts/0        Ss+  0:00 bash
25275 pts/7        Ss   0:00 bash
 8174 pts/7        Sl   0:14 _ emacs contenu.tex
17540 pts/7        S+   0:00 _ ./system
17541 pts/7        S+   0:00 _ sh -c ps f
17542 pts/7        R+   0:00 _ ps f
```

system returned 0

fork

- `pid_t fork();`
- duplique le processus courant
 - duplication de la mémoire, des fichiers ouverts, etc.
- le processus créé est le fils du processus courant
- le processus fils est une copie exacte du père sauf
 - le PID du fils est différent
 - le PPID du fils est le PID du père
- seule la valeur retournée par `fork` permet de différencier le père et le fils
 - le père reçoit le PID du fils
 - le fils reçoit 0

Voici un exemple d'utilisation de `fork`:

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

int main(int argc, char**argv) {
    printf("I am process %d. My PPID: %d\n", getpid(), getppid());

    pid_t ret_val = fork();
    if(ret_val == 0) {
        printf("I'm the child process. PID=%d, PPID=%d\n", getpid(), getppid());
        sleep(1);
    } else if (ret_val > 0) {
        printf("I'm the parent process. PID=%d, PPID=%d\n", getpid(), getppid());
    } else {
        printf("Fork failed\n");
    }
    return EXIT_SUCCESS;
}
```

Lors du `fork`, le processus est intégralement dupliqué. Le processus fils possède donc la même mémoire que le processus père: on retrouve donc les variables affectées dans le processus père avant le `fork`.

Après le `fork`, les espaces mémoire des deux processus deviennent dissociés: si le processus fils (resp. le père) modifie la variable `value`, il ne modifie que sa copie de la variable, et le processus père (resp. le fils) ne voit pas la modification. Par exemple, le programme suivant:

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

```

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

int main(int argc, char**argv) {
    int value = 1;
    printf("[%d] Before forking. value = %d\n", getpid(), value);
    pid_t ret_val = fork();
    if(ret_val == 0) {
        value++;
        printf("[%d] After forking. value = %d\n", getpid(), value);
    } else if (ret_val>0) {
        sleep(1);
        printf("[%d] After forking. value = %d\n", getpid(), value);
    } else {
        printf("Fork failed\n");
    }
    return EXIT_SUCCESS;
}

```

donnera, lors de son exécution, le résultat suivant:

```

$ ./fork_variable
[13229] Before forking. value = 1
[13230] After forking. value = 2
[13229] After forking. value = 1

```

La classe de fonctions exec

- ensemble de fonctions permettant d'exécuter une commande
 - `execlp`, `execvp`, `execve`, `execle`, `execlp`, etc.
- remplace le programme appelant par un nouveau
 - donc, on ne “sort” jamais de la fonction (sauf erreur)

Voici un exemple d'utilisation de `execlp`:

```

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

int main(int argc, char**argv) {
    printf("I am process %d. My PPID: %d\n", getpid(), getppid());

    pid_t ret_val = fork();
    if(ret_val == 0) {
        printf("I'm the child process. PID=%d, PPID=%d\n", getpid(), getppid());
        execlp("ps", "ps", "-l", NULL);
    }
}

```

```

    printf("This is printed only if execlp fails\n");
    abort();
} else if (ret_val>0) {
    printf("I'm the parent process. PID=%d, PPID=%d\n", getpid(), getppid());
    sleep(1);
}

return EXIT_SUCCESS;
}

```

`exec` est une famille de fonction permettant de remplacer l'image du processus courant: l'ensemble de l'espace mémoire est effacé et remplacé par l'image du programme exécuté.

Les paramètres peuvent être passés sous la forme d'une liste de paramètres (avec les fonctions `execl*`), ou sous la forme d'un tableau de chaînes de caractères (avec les fonctions `execv*`).

Terminaison d'un processus

- `pid_t wait(int *status);`
 - Attend la terminaison d'un processus fils
 - Le champs `status` permet de connaître la cause du décès.
- Variante: `pid_t waitpid(pid_t pid, int *wstatus, int options);`

Voici un exemple d'utilisation de la fonction `wait` :

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char**argv) {
    printf("I am process %d. My PPID: %d\n", getpid(), getppid());

    pid_t ret_val = fork();
    if(ret_val == 0) {
        printf("I'm the child process. PID=%d, PPID=%d\n", getpid(), getppid());
        sleep(1);
        return 17;
    } else if (ret_val>0) {
        printf("I'm the parent process. PID=%d, PPID=%d\n", getpid(), getppid());
        int status;
        pid_t pid = wait(&status);

        int exit_status = WEXITSTATUS(status);
    }
}

```

```

    printf("The child process %d ended with exit status %d\n", pid, exit_status);

} else {
    printf("Fork failed\n");
}
return EXIT_SUCCESS;
}

```

La fonction `wait` retourne le PID du processus fils qui s'est terminé et remplit la variable entière `status`. Cette variable peut ensuite être utilisée pour obtenir des informations sur la terminaison du processus fils.

Par exemple, la macro `WEXITSTATUS(status)` retourne le code de retour du processus fils. Ce programme donnera donc :

```

$ ./exemple_wait
I am process 21088. My PPID: 20960
I'm the parent process. PID=21088, PPID=20960
I'm the child process. PID=21089, PPID=21088
The child process 21089 ended with exit status 17

```

La fonction `waitpid` est une variante de `wait`. Elle permet d'attendre un processus fils précis, ou de tester (sans bloquer le processus appelant) la terminaison d'un processus.