

# Programmation Système

Alexis Wilhelm

alexis.wilhelm@gmail.com

hiver 2015–2016



version du  
5 décembre 2015

1

Attention !

Ce document n'est pas un cours, mais un *support* de cours : il est conçu pour illustrer mon propos, mais pas pour être lu seul (comme tout diaporama).

De plus, il n'est pas encore terminé et sera complété à mesure que le cours progressera.

Les notes que je laisse dans cette marge servent surtout à compenser les défaillances de ma mémoire, mais avec de la chance elles vous serviront aussi (et les retirer me demanderait plus de travail).

## Manipulation simple : jobs, bg, fg, kill

```
~$ albert # bloquant
^Z # [Ctrl][Z]
[1]+ Stopped albert
~$ jobs
[1]+ Stopped albert
~$ bg
[1]+ albert &
~$ jobs
[1]+ Running albert &
~$ fg # bloquant
albert
^Z # [Ctrl][Z]
[1]+ Stopped albert
~$ rené &
[2] 12558
~$ marcel &
[3] 12559
~$ jobs
[1]+ Stopped albert
[2] Running rené &
[3]- Running marcel &
~$ bg
[1]+ albert &
~$ jobs
[1] Running albert &
[2]- Running rené &
[3]+ Running marcel &
~$ kill %1
[1] Terminated albert
~$ jobs
[2]- Running rené &
[3]+ Running marcel &
~$ kill %+
[3]+ Terminated marcel
~$ jobs
[2]+ Running rené &
```

4

Qu'est-ce qu'un processus ? Pour répondre à cette question, on va chercher des indices dans les outils fournis par le système.

Tout d'abord, un processus peut être démarré en exécutant un programme. On peut alors attendre qu'il ait fini, l'interrompre, le passer à l'arrière-plan ou le fermer.

5

Sérieusement, au moins une fois.

 man sh

## Observation : ls /proc

acpi/	iomem	pagetypeinfo	vmstat	20/
asound/	ioports	partitions	zoneinfo	21/
buddyinfo	irq/	sched_debug	1/	22/
bus/	kallsyms	self@	2/	23/
cgroups	kcore	slabinfo	3/	24/
cmdline	keys	softirqs	5/	26/
consoles	key-users	stat	7/	27/
cpuinfo	kmsg	swaps	8/	28/
crypto	kpagecount	sys/	9/	29/
devices	kpageflags	sysrq-trigger	10/	31/
diskstats	loadavg	sysvipc/	11/	32/
dma	locks	thread-self@	12/	38/
driver/	meminfo	timer_list	13/	39/
execdomains	misc	timer_stats	15/	41/
fb	modules	tty/	16/	77/
filesystems	mounts@	uptime	17/	78/
fs/	mtrr	version	18/	79/
interrupts	net@	vmallocinfo	19/	80/

## Observation : ls /proc/1

attr/	environ	mem	personality	statm
autogroup	exe@	mountinfo	projid_map	status
auxv	fd/	mounts	root@	syscall
cgroup	fdinfo/	mountstats	sched	task/
clear_refs	gid_map	net/	schedstat	timers
cmdline	io	ns/	sessionid	uid_map
comm	limits	oom_adj	setgroups	wchan
coredump_filter	loginuid	oom_score	smaps	
cpuset	map_files/	oom_score_adj	stack	
cwd@	maps	pagemap	stat	

📖 man 5 proc

## Observation : ps -ef

UID	PID	PPID	C	STIME	TTY	TIME	CMD
alexis	11020	11019	0	13:33	?	00:00:00	/bin/sh ./ps.sh
alexis	11022	11020	0	13:33	?	00:00:00	ps -ef
alexis	11023	11020	0	13:33	?	00:00:00	grep \b11020\b \bPID\b

📖 man 1 ps

6 Le pseudo système de fichiers proc, monté dans /proc, donne des informations sur chaque processus ainsi que sur le noyau, notamment :

[cpuinfo](#) Informations sur les CPU

[meminfo](#) Informations sur la mémoire

[modules](#) Modules chargés

[mounts](#) Points de montage

Les suffixes sont ajoutés par ls -F :

/ pour les dossiers ;

@ pour les liens symboliques.

7 [status](#) Informations générales

[exe](#) Fichier exécutable

[cmdline](#) Ligne de commande

[cwd](#) Répertoire de travail

[environ](#) Variables d'environnement

[fd](#) Descripteurs de fichiers

[maps](#) Carte des adresses mémoire

8 Affiche les mêmes informations que proc, mais présentées dans un tableau.

[UID](#) Propriétaire

[PID](#) Identifiant numérique

[PPID](#) Processus parent

[C](#) Utilisation du CPU moyenne en %

[STIME](#) Heure de début

[TTY](#) Terminal

[TIME](#) Utilisation du CPU totale en hh:mm:ss

[CMD](#) Ligne de commande

On trouve aussi la syntaxe BSD « ps ax ».

## Priorité : nice, renice

```
~$ nice ./gros-calcul &
[1] 15267

~$ renice -n 50 15267
15267 (process ID) old priority 10, new priority 19

~$ renice -n -50 15267
renice: failed to set priority for 15267 (process ID): Permission denied

~$ sudo !!
sudo renice -n -50 15267
15267 (process ID) old priority 19, new priority -20

📖 man 1 nice
📖 man 1 renice
```

9 On peut démarrer un processus avec une priorité moindre (il est plus gentil avec le reste du système).

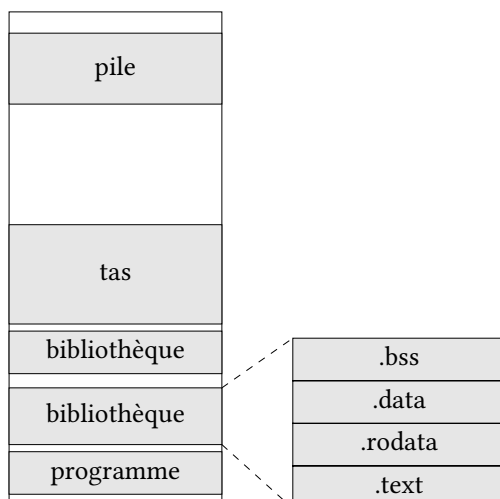
On peut aussi modifier sa priorité pendant qu'il tourne. Mais la gentillesse ne peut qu'augmenter et n'a pas le droit de décroître !

## Définition d'un processus

- ▶ Identifiant numérique
- ▶ Parent
- ▶ Propriétaire, permissions
- ▶ Répertoire, variables d'environnement, descripteurs
- ▶ Code
- ▶ Mémoire (tas, pile, registres)
- ▶ État (en cours, en attente, arrêté...)

10 Tout ceci est rangé dans une structure du noyau appelée « bloc de contrôle de processus » ou « PCB ».

## Organisation de la mémoire d'un processus



11 Le système se charge de faire correspondre les adresses virtuelles du processus avec les adresses physiques de la machine.

**.text** Code du programme

**.rodata** Constantes

**.data** Variables initialisées

**.bss** Variables non-initialisées. Signifiait historiquement « block started by symbol »

Le dessin n'est pas du tout à l'échelle.

## Duplication : fork()

13

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

int main()
{
    int child = fork(); // duplication
    assert(child != -1);
    if(child) printf("%d: Je suis le père de %d.\n", getpid(), child);
    else printf("%d: Je suis le fils de %d.\n", getpid(), getppid());
    sleep(1);
}
```

```
29696: Je suis le père de 29697.
29697: Je suis le fils de 29696.
```

fork() duplique le processus et retourne une valeur différente à chaque partie : le PID du fils au père et 0 au fils.

Le fils hérite donc de tout l'environnement et la mémoire de son père.

Par contre, après cet héritage ils évoluent indépendamment.

## Duplication : fork()

14

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

int main()
{
    fork(); // duplication
    fork(); // duplication
    printf("Je suis %d.\n", getpid()); // qui est là ?
}
```

```
Je suis 20129.
Je suis 20131.
Je suis 20130.
Je suis 20132.
```

Attention à bien contrôler qui fait quoi après un fork(), ou ça peut vite devenir le bazar.

## Attente : waitpid()

15

```
int main() {
    int child = fork();
    if(child) {
        printf("%d: Je suis le père de %d.\n", getpid(), child);
        int status;
        child = waitpid(child, &status, 0); // lecture
        printf("%d: Mon fils %d a fini avec un code %d.\n", getpid(), child,
            WEXITSTATUS(status));
    } else {
        printf("%d: Je suis le fils de %d.\n", getpid(), getppid());
        return 5; // écriture
    }
}
```

```
29680: Je suis le fils de 29679.
29679: Je suis le père de 29680.
29679: Mon fils 29680 a fini avec un code 5.
```

waitpid() attend qu'un fils ait terminé et récupère des informations retournées par ce fils.

On peut attendre (bloquant) ou juste vérifier le statut (non-bloquant) d'un fils en particulier ou de n'importe quel fils. On a aussi une forme simplifiée wait() qui attend n'importe quel fils.

## Attente : wait

16

Attention, en bash wait sans paramètre attend tous les enfants.

```
for i in 1 2 3
do sleep $i && echo $i &
done
echo début
wait
echo fin
```

début
1
2
3
fin

## Recouvrement : exec()

17

exec() remplace le processus par un autre. On conserve le PID, le PPID, le propriétaire, le répertoire, les variables d'environnement, les descripteurs... Mais on réinitialise toute la mémoire.

Ça permet de préparer le terrain avant d'exécuter un programme.

Attention, la ligne de commande commence par argv[0].

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

int main()
{
    execlp("date", "", "+%A %-d %B", NULL); // changement de programme
    puts("Cette ligne ne sera pas exécutée.");
}
```

mercredi 14 octobre
---------------------

## Recouvrement : exec()

18

Cette autre version prend une ligne de commande et des variables d'environnement rangées dans des tableaux. Du coup, les variables d'environnement ne sont pas héritées.

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

int main()
{
    char *args[] = {"", "-r", "execve.c", 0}; // nouvelle ligne de commande
    char *env[] = {"LC_ALL=C", 0}; // nouvelles variables d'environnement
    execve("/bin/date", args, env); // nouveau programme
    puts("Cette ligne ne sera pas exécutée.");
}
```

Mon Oct 12 16:57:20 CEST 2015
-------------------------------

## Environnement : getenv(), setenv(), unsetenv()

19

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

int main(int argc, char **argv)
{
    (void) argc;
    char const *var = getenv("var"); // lecture
    printf("var=%s\n", var);
    if(!var)
    {
        setenv("var", "OK", 1); // écriture
        fflush(0);
        execv(argv[0], argv);
    }
}
```

var=(null) var=OK
----------------------

getenv() (C) est utilisable depuis n'importe où dans le programme.

setenv() (POSIX) peut servir pour configurer une bibliothèque dans certains cas.

## Environnement : export

20

```
./setenv.exe

var=1 ./setenv.exe # local à cette commande

export var=2 # pour toutes les prochaines commandes
./setenv.exe
```

var=(null) var=OK var=1 var=2
--


Avec bash on peut passer des variables à un seul processus en les définissant devant la commande, ou à tous les prochains processus en utilisant export.

## Documentation


21


À lire le soir avant de dormir.

### POSIX

 The Open Group Base Specifications Issue 7  
<http://pubs.opengroup.org/onlinepubs/9699919799/>

### C

 C reference  
<http://en.cppreference.com/w/c>

 C gibberish ↔ English  
<http://www.cdecl.org/>