Systèmes d'exploitation: processus

Exécution d'une commande

msyska@mascopt:~

```
top - 00:07:21 up 1 day,  6:39,  2 users,  load average: 0.00, 0.01, 0.
Tasks: 175 total,   1 running, 174 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.0 us,  0.1 sy,  0.0 ni, 99.9 id,  0.0 wa,  0.0 hi,  0.0 si,
KiB Mem :  7993484 total,  6478260 free,   350088 used,  1165136 buff/c
KiB Swap:  8175612 total,  8175612 free,        0 used.  7336144 avail
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+
1464	mysql	20	0	907020	94408	7704	S	0.3	1.2	1:26.87
1	root	20	0	59604	7024	3984	S	0.0	0.1	0:03.13
2	root	20	0	0	0	0	S	0.0	0.0	0:00.02
3	root	20	0	0	0	0	S	0.0	0.0	0:00.24
5	root	0	-20	0	0	0	S	0.0	0.0	0:00.00
7	root	rt	0	0	0	0	S	0.0	0.0	0:00.12
8	root	20	0	0	0	0	S	0.0	0.0	0:00.00
9	root	20	0	0	0	0	S	0.0	0.0	0:00.00
10	root	20	0	0	0	0	S	0.0	0.0	0:00.00
11	root	20	0	0	0	0	S	0.0	0.0	0:00.00
12	root	20	0	0	0	0	S	0.0	0.0	0:00.00

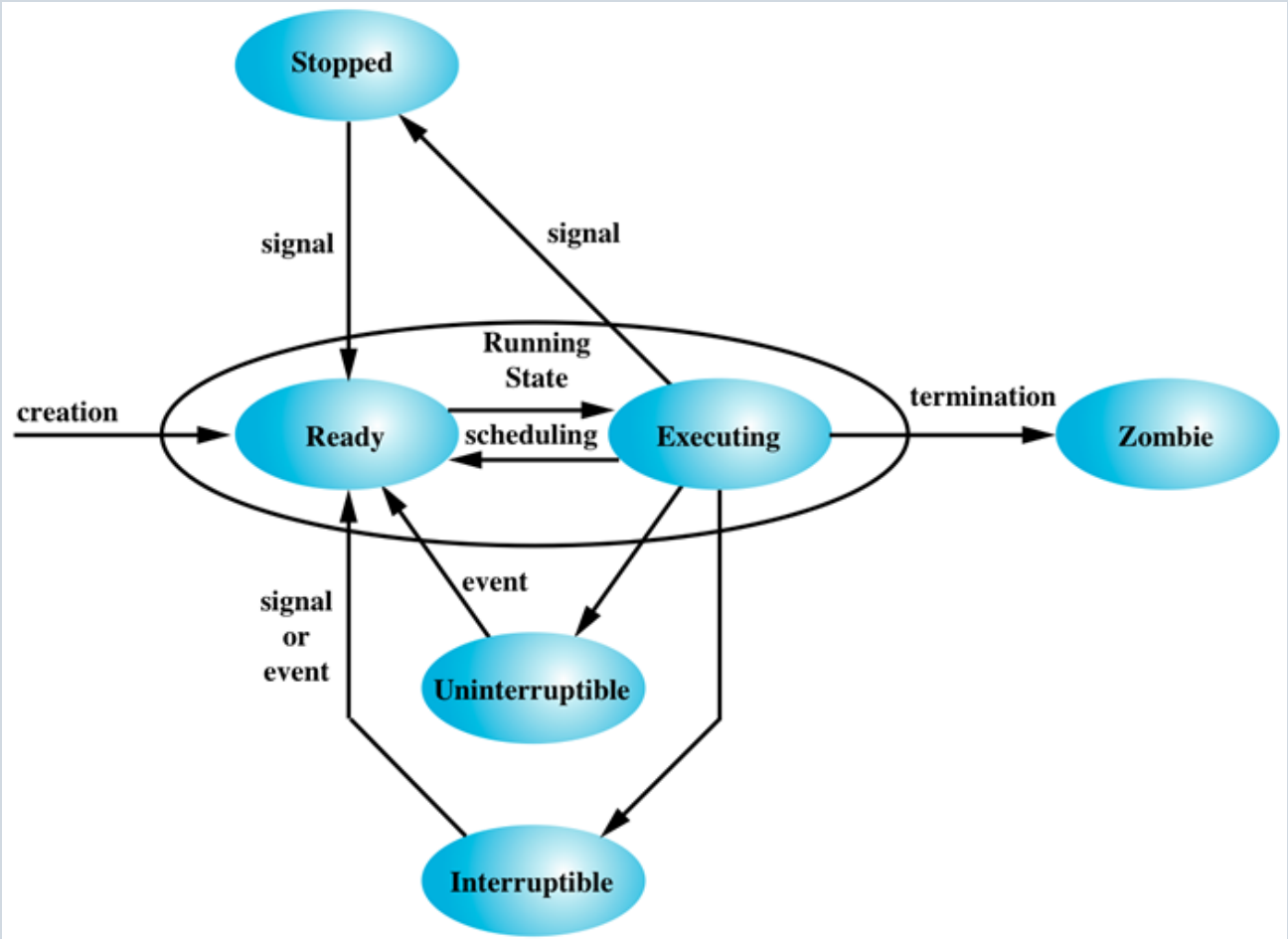


Figure 4.18 Linux Process/Thread Model

Organisation mémoire d'un programme C:

text segment : les instructions exécutées par la CPU

data segment : variables initialisées,

```
int maxcount = 99;
```

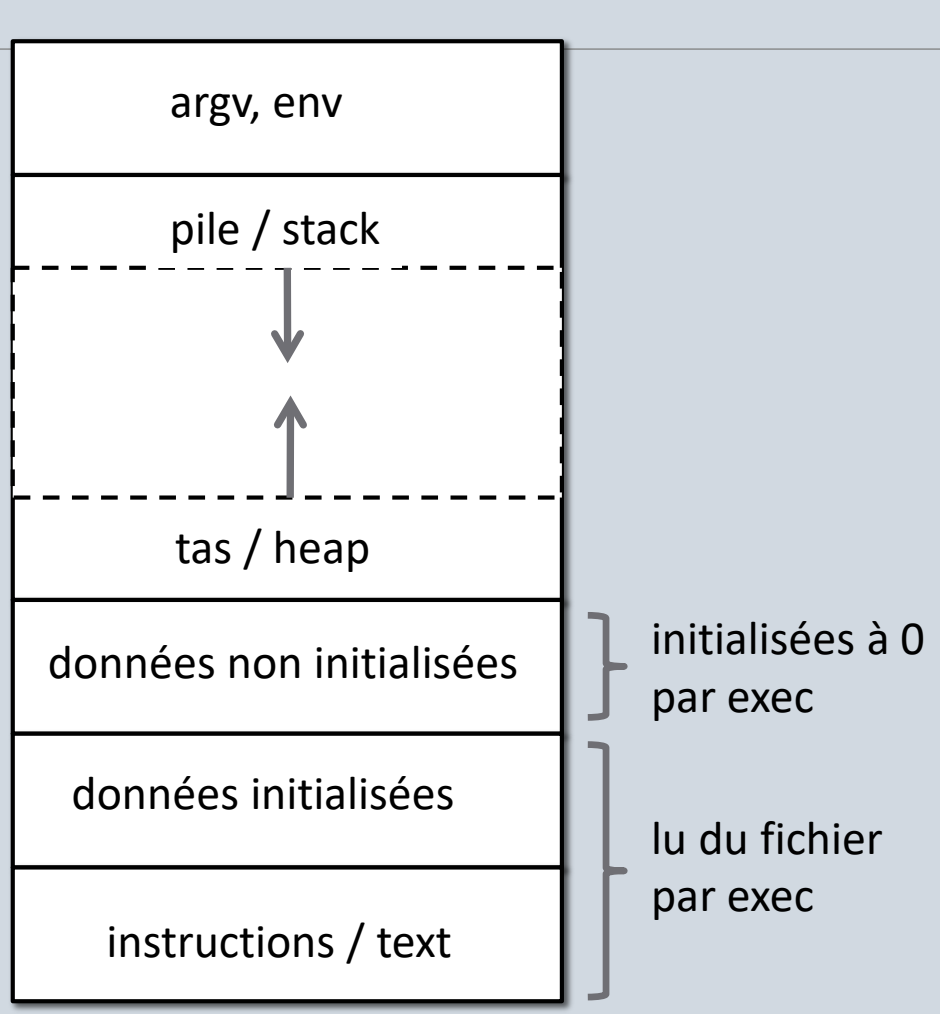
bss segment : non initialisées

```
long sum[];
```

stack : variables automatiques, appels de fonction, …

tas : allocation mémoire dynamique

```
$ size /bin/cc
```

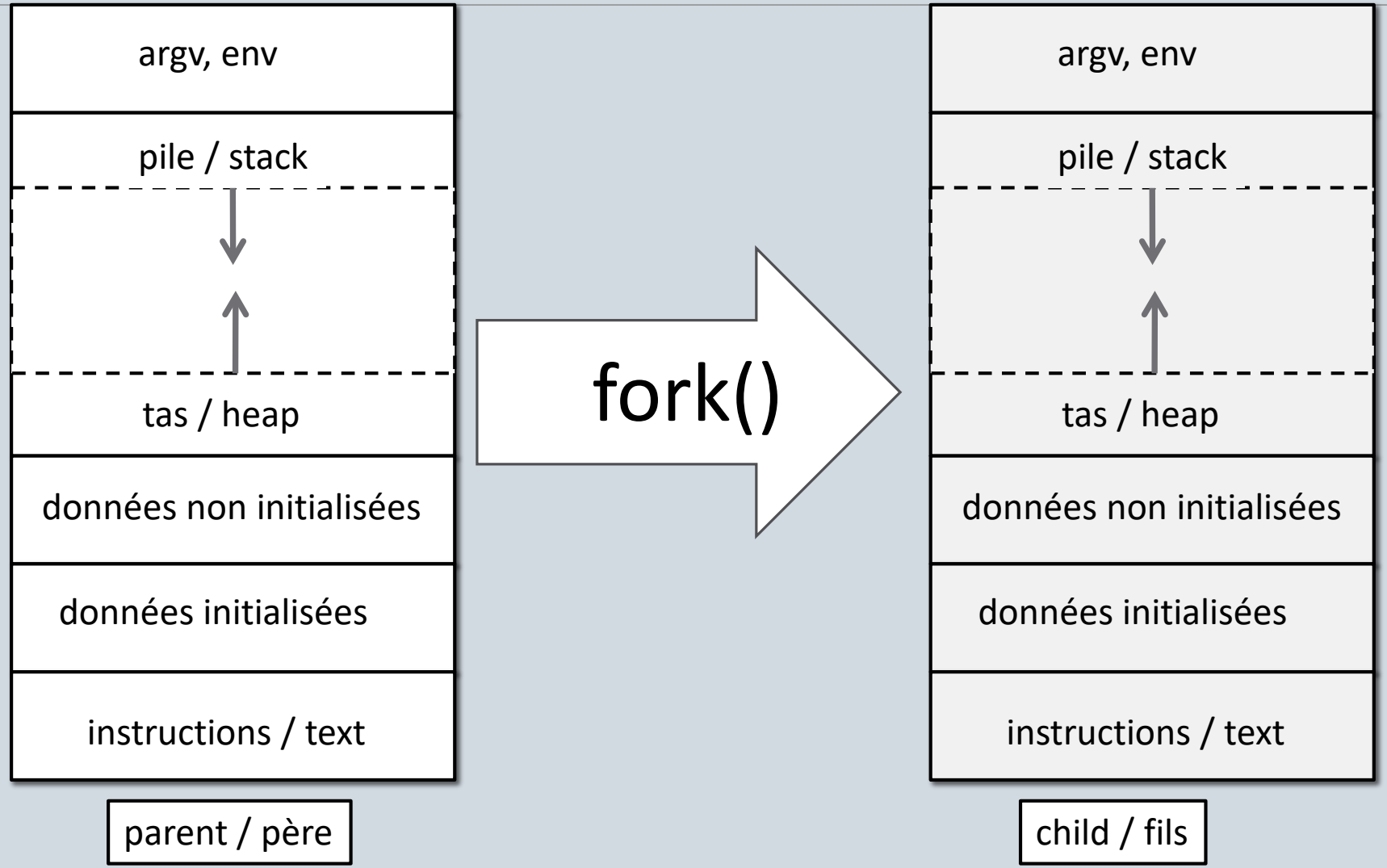


APUE pages 204-207

Processus Unix

- espace d'adressage visible par l'utilisateur
- bloc de contrôle du processus (BCP)
  - entrée dans la table des processus du noyau `struct proc` définie dans `<sys/proc.h>`
  - structure `struct user` appelée zone u définie dans `<sys/user.h>`
- multiplicité des exécutions
- plusieurs processus peuvent être l'exécution d'un même programme
- protection des exécutions
- un processus ne peut exécuter que ses instructions propres et ce de façon séquentielle; il ne peut pas exécuter des instructions appartenant à un autre processus.
- les processus sous UNIX® communiquent entre eux et avec le reste du monde grâce aux appels système.

Création d'un processus: clonage par fork()



man 2 fork

FORK(2) Manuel du programmeur Linux FORK(2)

NOM

fork - Créer un processus fils

SYNOPSIS

#include <unistd.h>

pid_t fork(void);

DESCRIPTION

fork() crée un nouveau processus en copiant le processus appelant. Le nouveau processus, qu'on appelle fils (« child »), est une copie exacte du processus appelant, qu'on appelle père ou parent, avec les exceptions suivantes :

* Le fils a son propre identifiant de processus unique, et ce PID ne correspond à l'identifiant d'aucun groupe de processus existant (setpgid(2)).
* L'identifiant de processus parent (PPID) du fils est l'identifiant de processus (PID) du père.

Exemple hello world!

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

int main(void){
   int res = fork();
   printf("Résultat : %d, %d, %d\n", res, getpid(), getppid());
   //    sleep(5); // no zombie
   return EXIT_SUCCESS;
}
```



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

int main(void){
   int res = fork();
   printf("Résultat : %d, %d, %d\n", res,
                getpid(), getppid());
   //    sleep(5); // no zombie
   return EXIT_SUCCESS;
}
```

Exécution #1 :	Résultat : 32389, 32388, 32329	Résultat : 0, 32389, 1
Exécution #2 :	Résultat : 32492, 32491, 32329	Résultat : 0, 32492, 32491

Commandes ps et top

```
$ ps
  PID TTY          TIME CMD
32329 pts/0    00:00:00 usershell
32493 pts/0    00:00:00 ps
$ ps x
  PID TTY      STAT   TIME COMMAND
32328 ?        S      0:00 sshd: syska@pts/0
32329 pts/0    Ss     0:00 -usershell
32494 pts/0    R+     0:00 ps x
$ ps ax
  PID TTY      STAT   TIME COMMAND
    1 ?        Ss     0:39 init [2]
    2 ?        S      0:00 [kthreadd]
```

```
$ pstree
init─┬─NetworkManager───{NetworkManager}
     ├─accounts-daemon───{accounts-daemon}
     ├─acpid
     ├─at-spi-bus-laun─┬─dbus-daemon
     │                 └─3*[{at-spi-bus-laun}]
     ├─at-spi2-registr───{at-spi2-registr}
     ├─avahi-daemon───avahi-daemon
     ├─bluetoothd
. . .
     ├─sshd─┬─3*[sshd───sshd───sftp-server]
     │      └─sshd───sshd───usershell───pstree
     ├─udevd───2*[udevd]
     └─upowerd───2*[{upowerd}]
```

Système de fichiers /proc

```
$ sleep 1000 &
[1] 32539
$ ps
  PID TTY          TIME CMD
32329 pts/0    00:00:00 usershell
32539 pts/0    00:00:00 sleep
32540 pts/0    00:00:00 ps
$ cd /proc/32539
$ ls
attr        comm              fd        mem         numa_maps      root        statm
autogroup   coredump_filter   fdinfo    mountinfo   oom_adj        sched       status
auxv        cpuset            io        mounts      oom_score      sessionid   syscall
cgroup      cwd               limits    mountstats  oom_score_adj  smaps       task
clear_refs  environ           loginuid  net         pagemap        stack       wchan
cmdline     exe               maps      ns          personality    stat
$
```

Exemple fork()

```
int main(void) {
   pid_t pid_fils = fork();

   if (pid_fils == -1) {
            fprintf(stderr, "fork() impossible, errno=%d\n", errno);
            return EXIT_FAILURE;
   }
   if (pid_fils == 0) {
            /* processus fils */
            fprintf(stdout, "Fils : PID=%ld, PPID=%ld\n", (long) getpid(), (long) getppid());
            sleep(5); // sinon le fils peut terminer avant le début du père
            return EXIT_SUCCESS;

   } else { // pid_fils contient le PID du processus créé par fork()
            /* processus père */
            fprintf(stdout, "Père : PID=%ld, PPID=%ld, PID fils=%ld\n", (long) getpid(), (long) getppid(),
            (long) pid_fils);
            wait(NULL); // on attend le premier fils qui termine
            return EXIT_SUCCESS;
   }
}
```

Illustration du schéma mémoire

```
int glob = 6; /* external variable in initialized data */
char buf[] = "a write to stdout\n";

int main(void) {
        int var; /* automatic variable on the stack */
        pid_t pid;

        var = 88;
        if (write(STDOUT_FILENO, buf, sizeof(buf) - 1) != sizeof(buf) - 1)
                fprintf(stderr, "write error");
        printf("before fork\n"); /* we don't flush stdout */

        if ((pid = fork()) < 0)
                fprintf(stderr, "fork error");
        else if (pid == 0) { /* child */
                glob++; /* modify variables */
                var++;
        } else
                printf("pid = %d\n", pid);
        sleep(2); /* parent */

        printf("pid = %d, glob = %d, var = %d\n",
                                getpid(), glob, var);
        exit(0);
}
```

No zombie : wait()

```
int main(void) {
        pid_t resfork, reswait;
        if ((resfork = fork()) < 0)
                fprintf(stderr, "fork error");
        else if (resfork == 0) { /* child */
                sleep(2);
        } else {
                sleep(2); /* parent */
                reswait = wait(NULL);
        }
        printf("resfork= %d, getpid = %d, ppid = %d, reswait = %d\n", resfork, getpid(), getppid(), reswait);
        exit(EXIT_SUCCESS);
}
$ ./a.out
pid = 0, getpid = 32593, ppid = 32592, reswait = 0
pid = 32593, getpid = 32592, ppid = 32329, reswait = 32593
```

waitpid() - 1/2

```
int main(void) {
        pid_t pid;
        if ((pid = fork()) < 0) fprintf(stderr, "fork error");
        else if (pid == 0) { /* first child */
                if ((pid = fork()) < 0)
                        fprintf(stderr, "fork error");
                else if (pid > 0) {
                        printf("Parent from second fork, pid = %d\n", getppid());
                        exit(EXIT_SUCCESS); /* parent from second fork == first child */
                }
                /* We're the second child; our parent becomes init as soon
                 as our real parent calls exit() in the statement above.
                 Here's where we'd continue executing, knowing that when
                 we're done, init will reap our status. */
                sleep(2);
                printf("second child, parent pid = %d\n", getppid());
                exit(EXIT_SUCCESS);
        }
```

waitpid() - 2/2

```
        if (waitpid(pid, NULL, 0) != pid) /* wait for first child */
                fprintf(stderr, "waitpid error");

        /* We're the parent (the original process); we continue executing,
         knowing that we're not the parent of the second child. */
        printf("Parent, pid = %d\n", getppid());

        exit(EXIT_SUCCESS);
}
```

```
$ gcc fork_waitpid.c -o fork_waitpid
$ ./fork_waitpid
Parent from second fork, pid = 32673
Parent, pid = 32329
$ second child, parent pid = 1
```

Exit status du fils

```
int main(int argc, char *argv[]) {
        pid_t pid;
        pid_t reswait;
        int status;

        if ((pid = fork()) < 0)
                fprintf(stderr, "fork error");
        else if (pid == 0) { /* child */
                sleep(2);
                exit(atoi(argv[1]));
        } else {
                sleep(2); /* parent */
                reswait = wait(&status);
                if (WIFEXITED(status))
                  printf("pid = %d, getpid = %d, reswait = %d status = %d\n", pid, getpid(), reswait, WEXITSTATUS(status));
        }
        return EXIT_SUCCESS;
}
```

Boucle de création de processus

```
int main (int argc, char * argv[]) {
   pid_t   pid;
   int       fils;
   int NB_FILS = atoi(argv[1]);

   for (fils = 0; fils < NB_FILS; fils ++) {
            if ((pid = fork()) < 0) {
               perror("fork");
               exit(EXIT_FAILURE);
            }
            if (pid != 0) continue;
            printf("Je suis le fils %d (%d) du père %d\n", fils, getpid(), getppid());
            exit(EXIT_SUCCESS);
   }
   while (wait(NULL) > 0)
            printf("…\n");
   exit(EXIT_SUCCESS);
}
```

À suivre

- exécution d'une commande dans un processus
- contrôle des processus par des signaux
- mesure du parallélisme (ordonnancement)