# EPU - Informatique ROB4
# Informatique Système

## Processus, création de processus avec fork() et primitives de recouvrement

**Miranda Coninx**
Presented by
**Ludovic Saint-Bauzel**
ludovic.saint-bauzel@sorbonne-universite.fr

Sorbonne Université
Institut des Systèmes Intelligents et de Robotique (CNRS UMR 7222)

2023-2024

SORBONNE UNIVERSITÉ

ISIR
INSTITUT DES SYSTÈMES INTELLIGENTS ET DE ROBOTIQUE

# Plan de ce cours

## Processus

- Definition générale
- Exécution d'un programme : illustration (très) simplifiée
- Etat d'un processus
- Attributs d'un processus
- Commandes du shell liées aux processus

## Création de processus avec fork()

- Description
- Exemple : simple fork
- Synchronisation parent / enfant
- Exemple : fork & synchronisation
- Exercices

## Primitives de recouvrement

- Principe
- execv()
- Exemple : simple execv()
- La commande system()
- Exercises

## Distinctions à faire

- **Code exécutable** : instructions machine compréhensibles par le processeur
- **Code source** : instructions dans un langage de programmation (C, C++, ...) compréhensibles par un humain
- **Programme exécutable** : fichier contenant du code exécutable
- **Programme source** : fichier contenant du code source
- **Processus** : programme exécutable en cours d'exécution

## Définition

- Chaque programme (fichier exécutable ou script) en cours d'exécution dans les système (OS) correspond à un (et parfois plusieurs) **processus** du système.
- Un **processus** représente l'image de l'état du processeur et de la mémoire au cours de l'exécution d'un programme : le programme est statique et le processus représente la dynamique de son exécution.

## Process = program in execution

- Program code, resources (e.g. files in I/O), address space, ...
- One or more threads
- Can communicate through pipes, signals, network, files

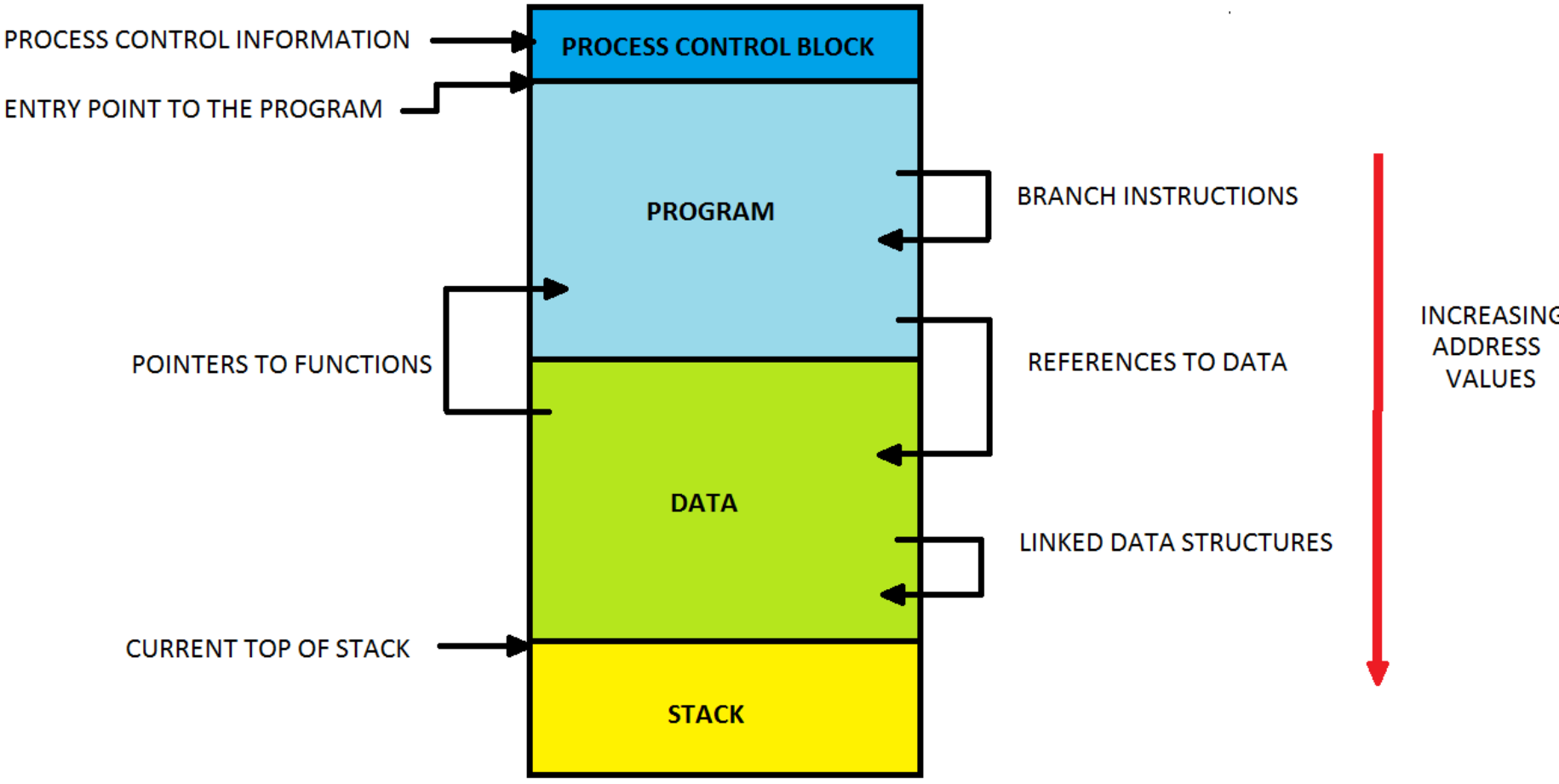
## Process descriptor

- Each process has its Process Descriptor, which is used to keep track of the process in memory
- Stores PID, state, parent process, children, registers, address space information, open files
- Process information is usually stored in the so called Process Table, i.e. an array of PD

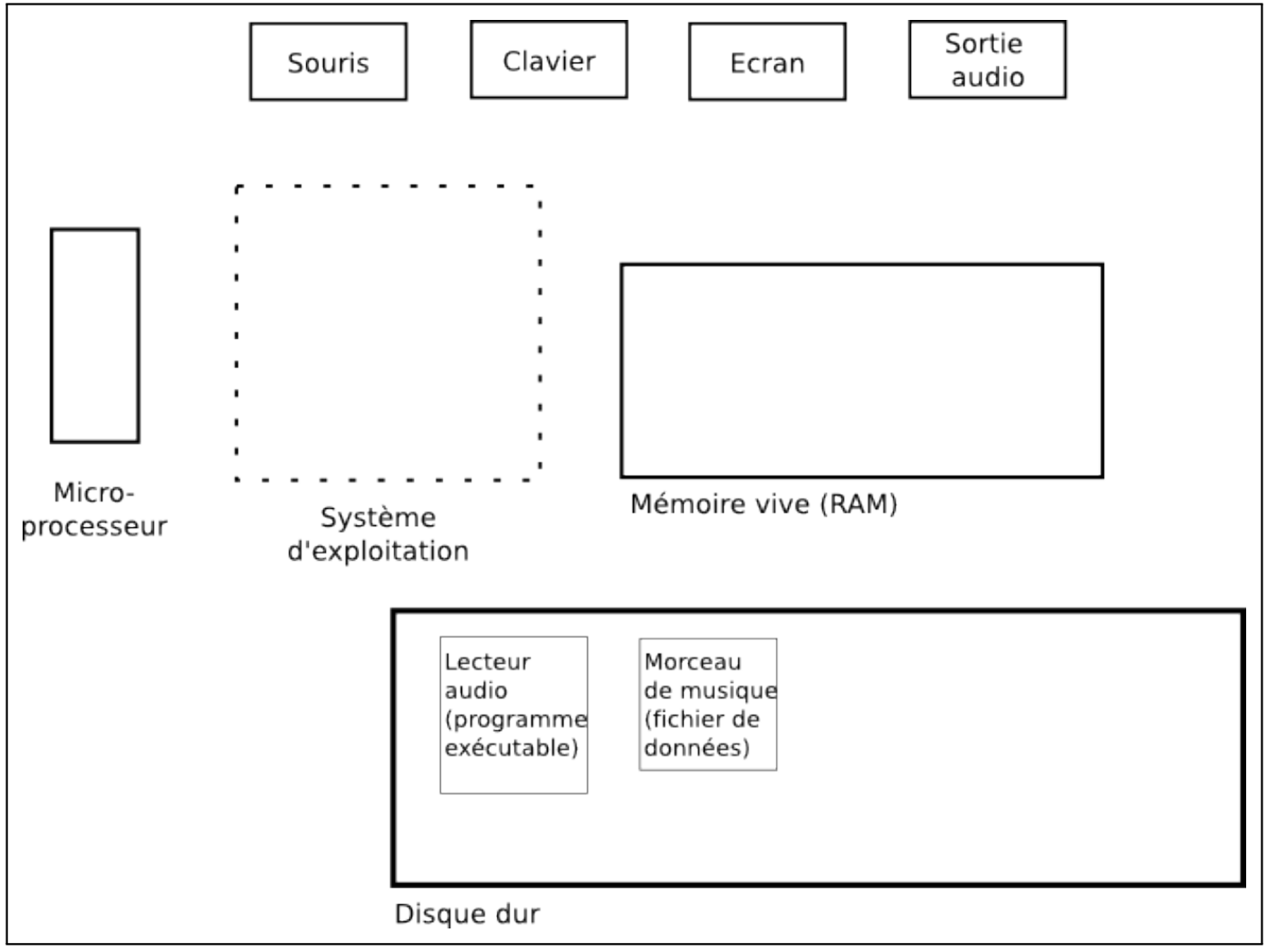
## OS - Process Interaction

- The process interacts with the OS via **system calls**, which are special instructions made available by the OS to perform "privileged" operations
  - user mode vs kernel mode
  - system calls can be I/O (read/write on streams), process control (kill, wait, stop), etc.
- The OS interacts with all processes by **scheduling** the processes and managing each process lifecycle (creation, execution, termination)
  - maximize CPU utilization providing reasonable response time
  - this is crucial : it strongly affects the distinction between hard and soft real time *(you will have a class about real-time programming next year - by L. Saint-Bauzel)*

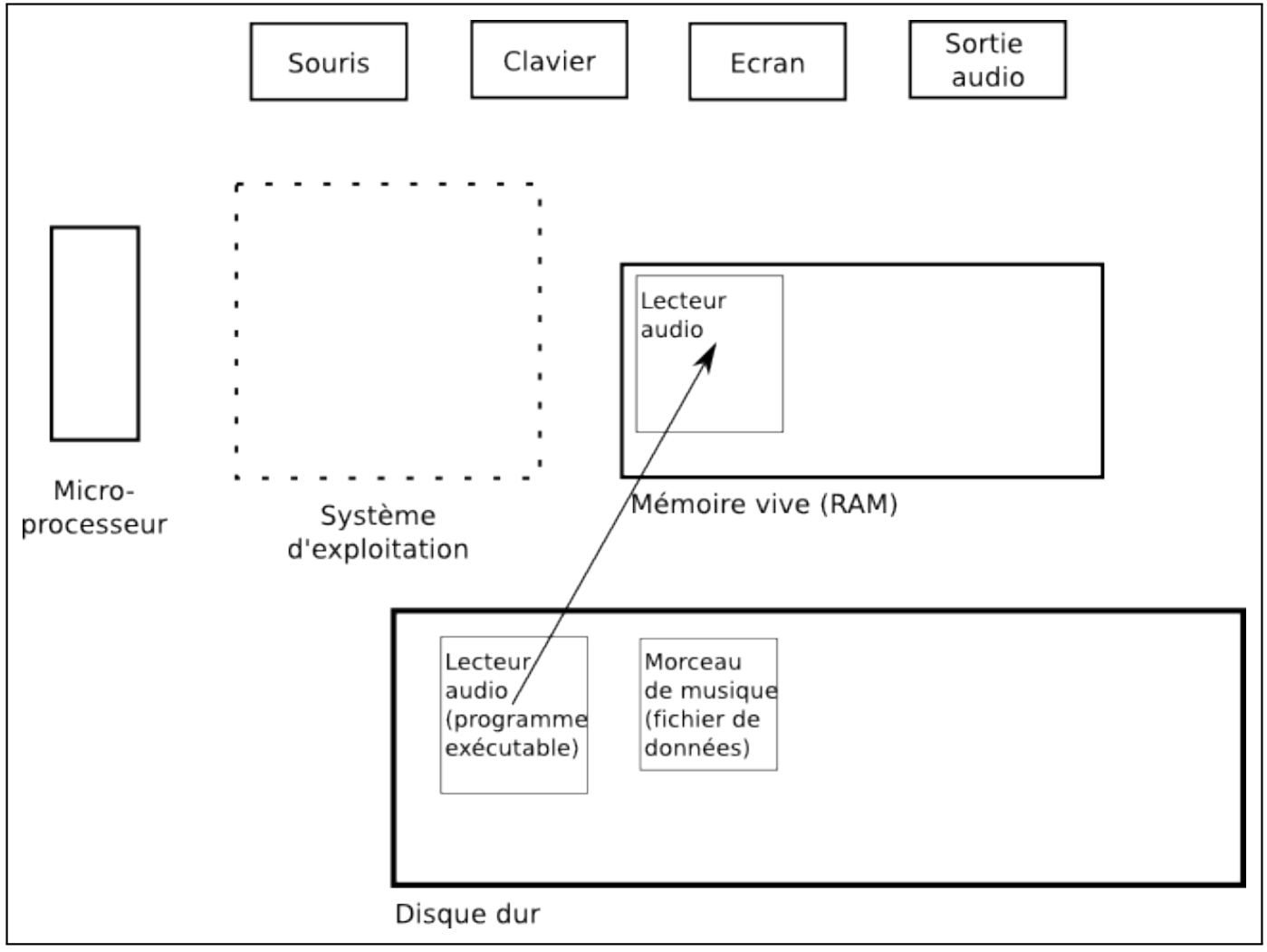
## ↦ a process in memory



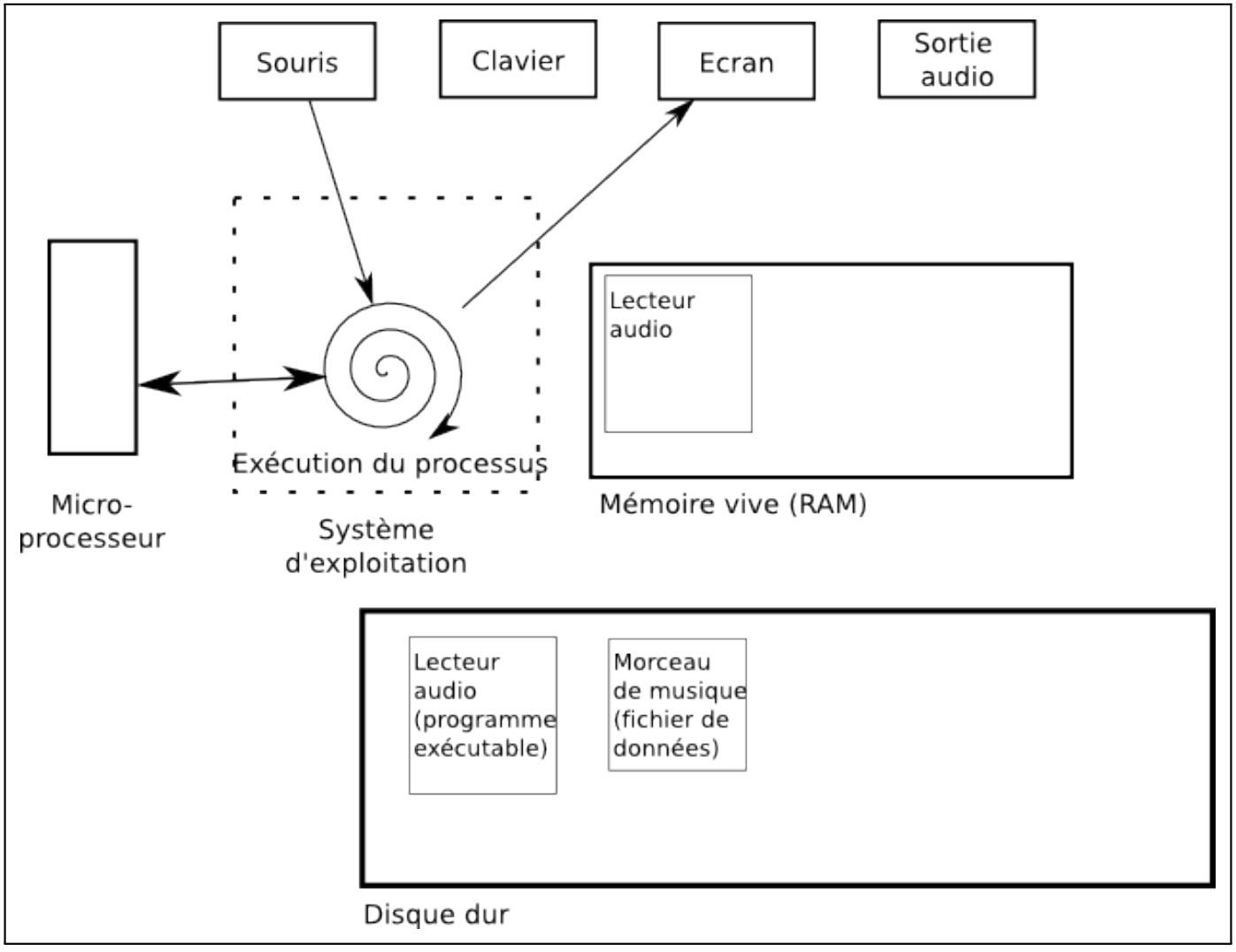
# (1) Lecture d'un fichier audio



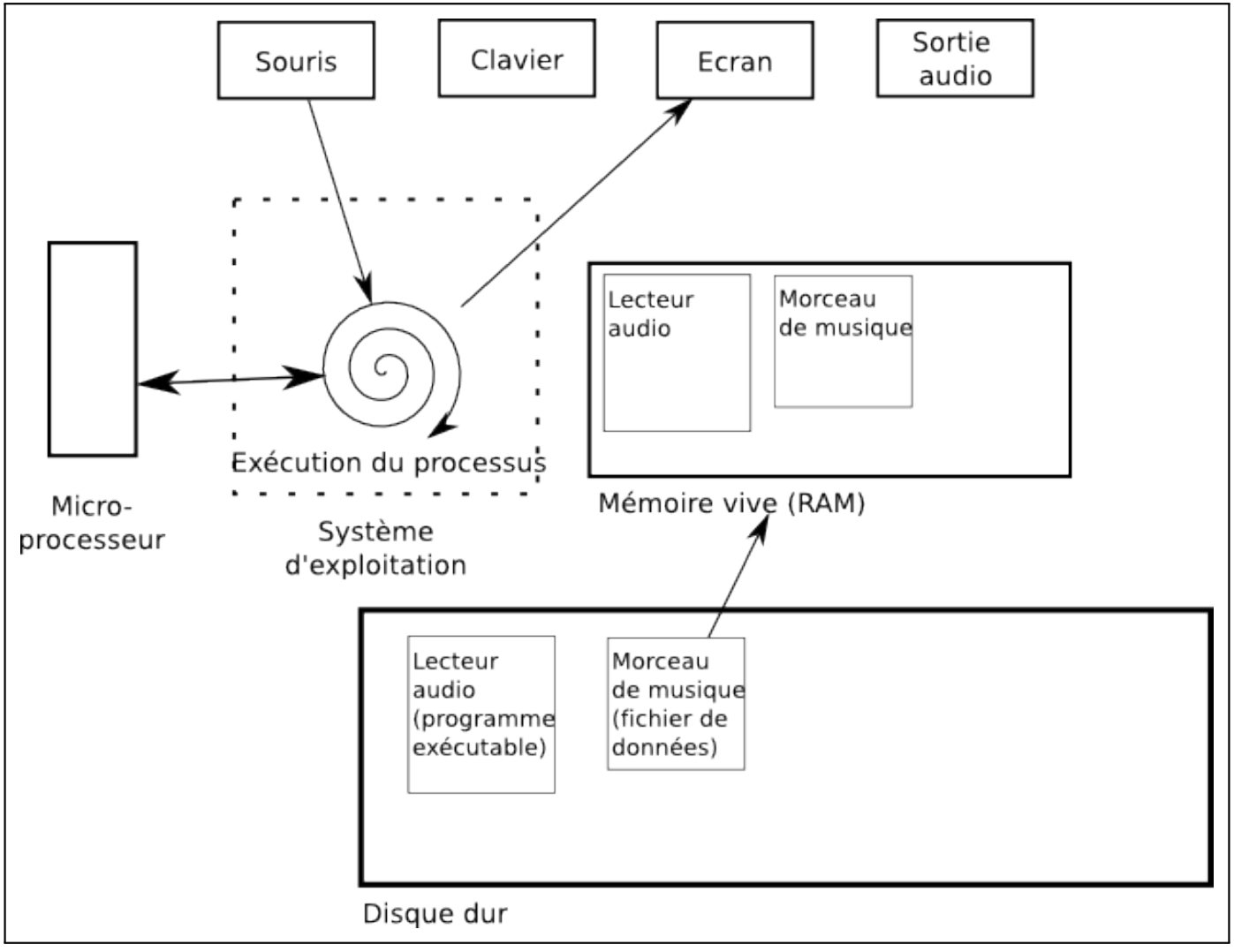
# (2) Chargement du programme en mémoire



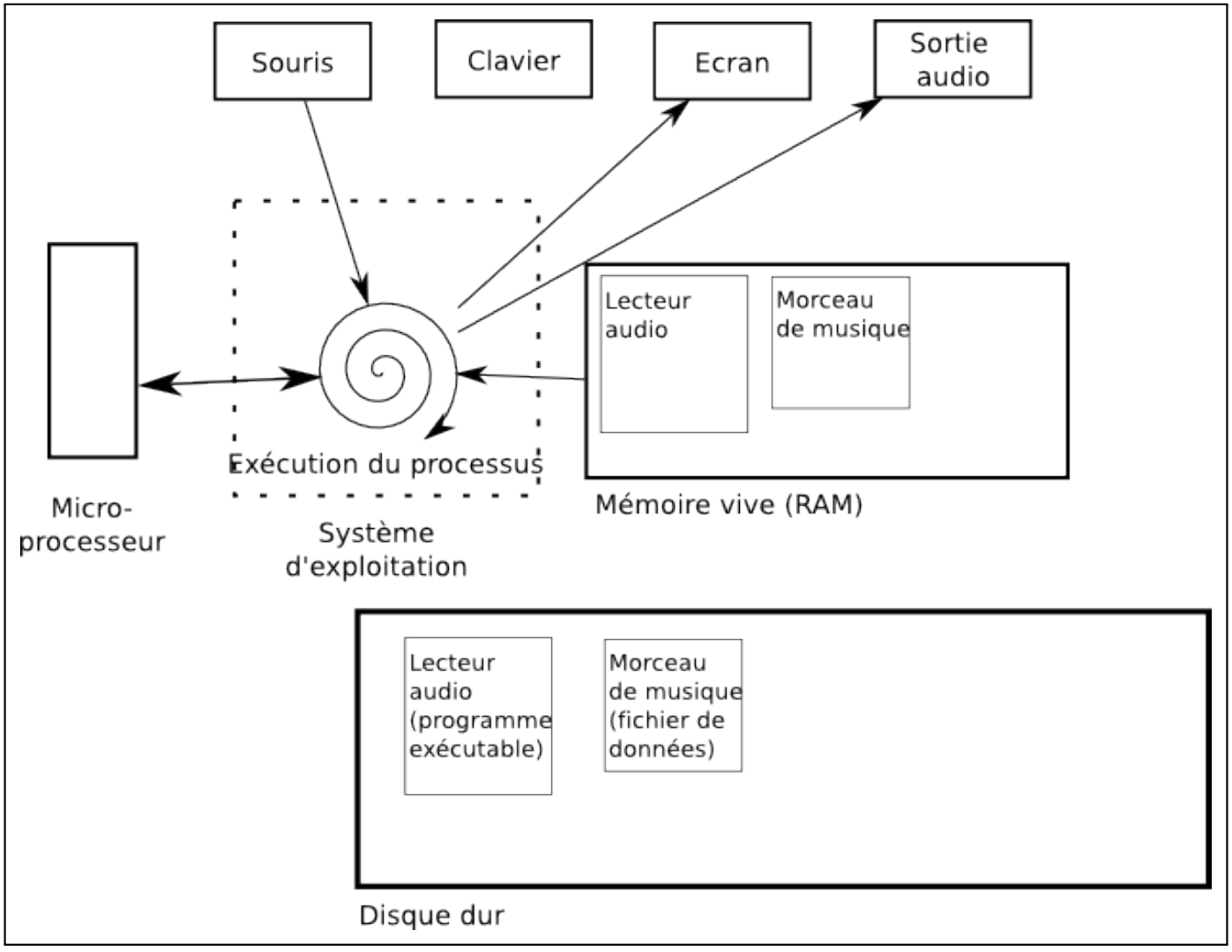
# (3) Naissance du processus



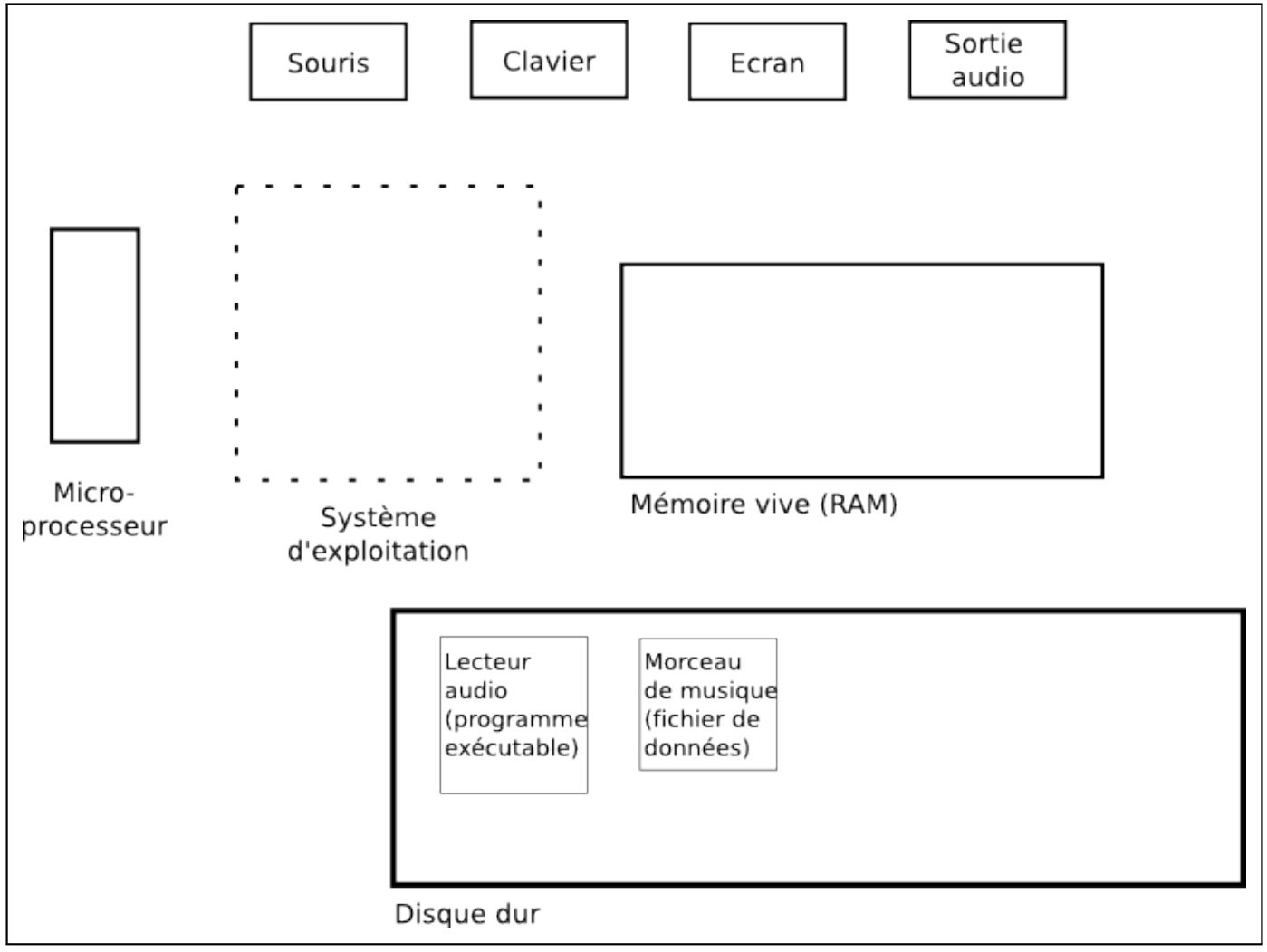
# (4) Chargement du fichier audio en mémoire



# (5) Lecture du fichier audio

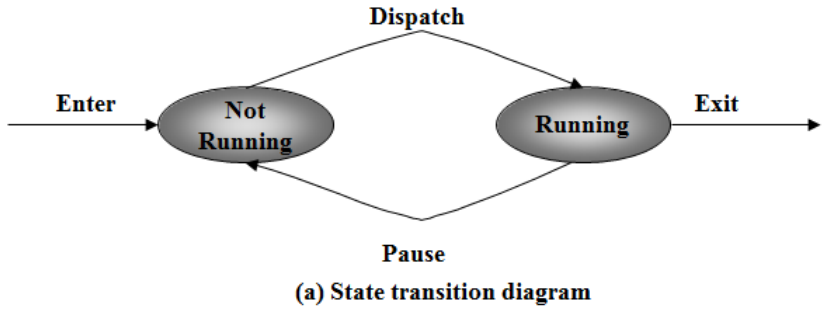


# (6) Mort du processus

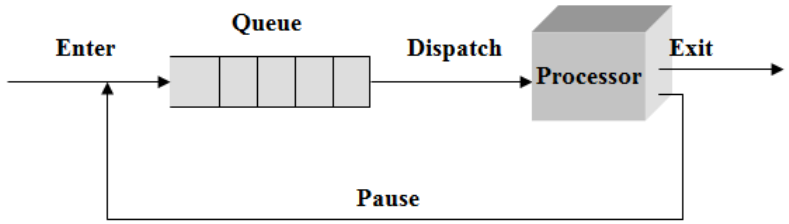


## Scheduler vs dispatcher

- Scheduling and dispatching are usually performed within the same routine. They prevent a single process from monopolizing processor time
- **Scheduler** decides the next process executed by CPU (i.e. the order/sequence of processes)
- **Dispatcher** handles the **process switch**, that is switches the processor from one process to another. It restores the context for the process (program counter etc.)



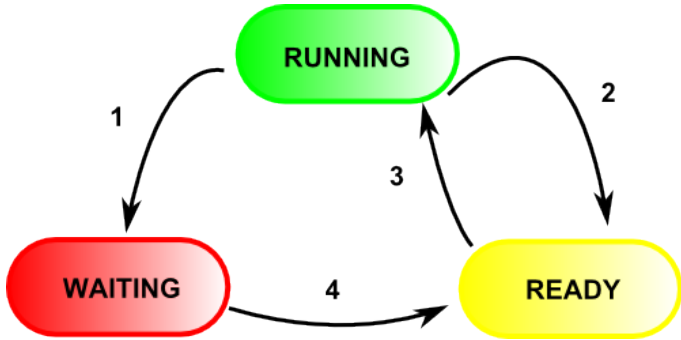

(a) State transition diagram




(a) Queuing diagram

**Three states model (Tanenbaum & Woodhull)**

- Running : execution, process is using the CPU
- Ready : process can be executed but is temporarily waiting, CPU is used by another process
- Waiting : process can't be executed ; it is either blocked because waiting for an external event (e.g. I/O, interrupt from another process) or because scheduler assigned CPU to others




- **Transitions**
  1. process is blocked because waiting for input or interrupt
  2. scheduler assigns CPU to another process
  3. scheduler assigns CPU to the process
  4. an external input unblocks the process

# Unix process model

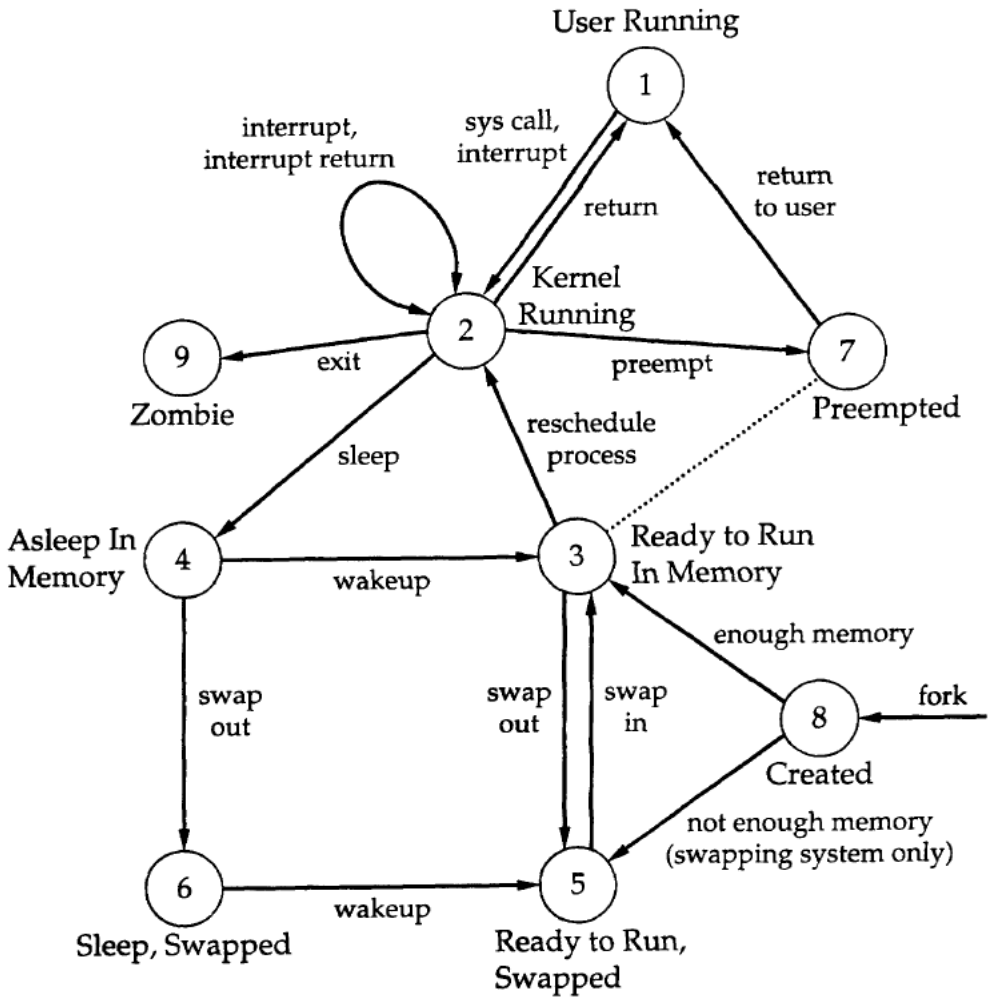



**FIGURE 3.16 UNIX process state transition diagram**

Figure from *Operating Systems*, 2nd edition, Stallings, Prentice Hall, 1995

$\mapsto$ **We will consider the following process model**

## Les 3 états d'un processus

- **Elu** : état d'exécution du processus, i.e. le processeur lui est dédié
- **Bloqué** : état d'attente d'une ressource (indisponible) du processus
- **Prêt** : état d'attente du processeur par le processus (par exemple : le processus a obtenu la main sur la ressource attendue mais le processeur ne lui est plus dédié)

## 4ème état

- Certains OS (dont Linux) permettent aux processus de créer eux mêmes des processus
- On parle alors de **processus parent** et de **processus enfant**
- Cela induit l'existence d'un quatrième état :
  $\hookrightarrow$ **Zombie** : le processus a terminé son exécution mais est toujours existant car son parent n'a pas (encore) pris en compte sa terminaison

## Les attributs principaux d'un processus

- **PID** : l'identifiant numérique propre au processus
- **PPID** : le PID du processus parent (processus à l'origine de la création du processus)
- **UID** : l'identifiant de l'utilisateur qui a lancé le processus
- **GID** : le groupe de l'utilisateur qui a lancé le processus

## Droits du processus

- Les droits du processus à accéder en lecture, écriture ou exécution à certains fichiers ou a certaines commandes (et de manière générale aux ressources de la machine) sont basés sur son UID et son GID.
- Les processus de l'OS s'exécutent en mode superviseur sans restriction de droits.
- Un processus utilisateur peut demander l'exécution d'une routine de l'OS (appel système). Cela induit un changement du mode d'exécution le temps de l'exécution de cet appel système et une **commutation de contexte** (*context switch*) : sauvegarde de l'état du processus utilisateur avant l'appel système et restauration de cet état au retour de cet appel.
- Il existe un cas particulier, appelé `Set - UID`, où un utilisateur possédant les droits d'exécution sur un fichier exécutable peut exécuter ce fichier avec les **droits du propriétaire du fichier**.
- Dans ce cas précis, le GID correspond bien au groupe de l'utilisateur lançant le processus mais l'UID correspond à l'identifiant de l'utilisateur propriétaire du fichier.

```
##include <sys/types.h>
##include <unistd.h>
gid_t getgid (void);
pid_t getpid (void);
pid_t getppid (void);
uid_t getuid (void);
```

- `getpid()` : returns the process ID of the calling process
- `getppid()` : returns the process ID of the parent of the calling process
- `getgid()` : returns the group ID of the calling process
- `getuid()` : returns the user ID of the calling process

Accès à ces informations depuis un programme C : **** test_getpid.c ****

```c
#include <unistd.h>      // getpid(), gettpid(), getuid(), getgid()
#include <sys/types.h>   // pid_t, uid_t, gid_t
#include <stdio.h>       // fprintf()

int main(int argc, char* argv[]){

        pid_t my_pid  = getpid();
        pid_t my_ppid = getppid();
        uid_t my_uid = getuid();
        gid_t my_gid = getgid();

        fprintf(stdout,"Attributs de ce processus :\n");
        fprintf(stdout,"Mon pid vaut : %d\n",my_pid);
        fprintf(stdout,"Mon ppid vaut : %d\n",my_ppid);
        fprintf(stdout,"Mon uid vaut : %d\n",my_uid);
        fprintf(stdout,"Mon gid vaut : %d\n",my_gid);

        return 0;
}
```

```
$ ./my_exec
Attributs de ce processus :
Mon pid vaut : 1859
Mon ppid vaut : 1836
Mon uid vaut : 1000
Mon gid vaut : 1000
```

## chmod

- La commande shell permettant de modifier les droits d'accès à un fichier est `chmod`
- Elle permet de spécifier :
  - les droits de l'utilisateur propriétaire du fichier (u)
  - les droits des utilisateurs membres du même groupe (g)
  - les droits des autres utilisateurs (o)
- Elle permet d'autoriser ou d'interdire la lecture, l'écriture et l'exécution d'un fichier pour chaque type d'utilisateur. Elle permet aussi de donner les droits `Set - UID`.

```
$ ls -l
-rw-r--r-- 1 vincent vincent 9383 2009-09-20 20:31 my_exec
$ chmod +x my_exec
$ ls -l
-rwxr-xr-x 1 vincent vincent 9383 2009-09-20 20:31 my_exec
$ chmod go-x my_exec
$ ls -l
-rwxr--r-- 1 vincent vincent 9383 2009-09-20 20:31 my_exec
$ chmod 755 my_exec
$ ls -l
-rwxr-xr-x 1 vincent vincent 9383 2009-09-20 20:31 my_exec
$ chmod uog=rw my_exec
$ ls -l
-rw-rw-rw- 1 vincent vincent 9383 2009-09-20 20:31 my_exec
$ chmod uog+x my_exec
$ ls -l
-rwxrwxrwx 1 vincent vincent 9383 2009-09-20 20:31 my_exec
$ man chmod
```

## ps

La commande `ps` permet de visualiser les processus en cours d'exécution par l'OS.

- `$ ps x` permet de visualiser les processus liés à l'utilisateur
- `$ ps aux` permet de visualiser les processus du système
- `$ ps -f x` permet de voir l'ensemble des attributs des processus de l'utilisateur
- `$ ps -f x | grep firefox` permet de voir les attributs du processus dont la description fait apparaitre la chaine de caractère `firefox`

```
icub@icubTest:~$ ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.0  23680  1952 ?        Ss   14:49   0:00 /sbin/init
root         2  0.0  0.0      0     0 ?        S    14:49   0:00 [kthreadd]
root         3  0.0  0.0      0     0 ?        S    14:49   0:00 [migration/0]
root         4  0.0  0.0      0     0 ?        S    14:49   0:00 [ksoftirqd/0]
...
root       617  0.0  0.0  49260  2524 ?        Ss   14:49   0:00 /usr/sbin/sshd -D
syslog     624  0.0  0.0 125980  1624 ?        Sl   14:49   0:00 rsyslogd -c4
102        640  0.0  0.0  24268  1800 ?        Ss   14:49   0:01 dbus-daemon --system --fo
root       673  0.0  0.1  94112  4244 ?        Ssl  14:49   0:00 NetworkManager
avahi      674  0.0  0.0  33928  1584 ?        S    14:49   0:00 avahi-daemon: registering
avahi      679  0.0  0.0  33928   580 ?        Ss   14:49   0:00 avahi-daemon: chroot help
root       680  0.0  0.0  57868  2516 ?        S    14:49   0:00 /usr/sbin/modem-manager
root       687  0.0  0.1  83100  3664 ?        Ssl  14:49   0:00 gdm-binary
...
```

## top

La commande top fournit le même type d'information concernant les processus avec un formatage différent et des informations d'utilisation de la mémoire en plus. Version plus moderne : htop

```
icub@icubTest:~$ top

top - 17:12:39 up  2:23,  4 users,  load average: 1.05, 0.49, 0.39
Tasks: 166 total,   1 running, 165 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.3%us,  2.3%sy,  0.0%ni, 97.3%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   3081868k total,  1307080k used,  1774788k free,   185768k buffers
Swap:   605176k total,        0k used,   605176k free,   592312k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 1025 root      20   0  335m  88m  14m S    4  2.9   3:05.30 Xorg
 2914 icub      20   0  313m  24m  12m S    3  0.8   0:24.46 gnome-terminal
 1313 icub      20   0  436m  50m  16m S    1  1.7   0:55.08 compiz
 1402 root      20   0 46700  944  588 S    0  0.0   0:06.59 udisks-daemon
 3360 icub      20   0 19224 1440 1052 R    0  0.0   0:00.13 top
    1 root      20   0 23680 1952 1272 S    0  0.1   0:00.98 init
    2 root      20   0     0    0    0 S    0  0.0   0:00.01 kthreadd
    3 root      RT   0     0    0    0 S    0  0.0   0:00.02 migration/0
    4 root      20   0     0    0    0 S    0  0.0   0:00.16 ksoftirqd/0
    5 root      RT   0     0    0    0 S    0  0.0   0:00.00 watchdog/0
    6 root      RT   0     0    0    0 S    0  0.0   0:00.02 migration/1
    7 root      20   0     0    0    0 S    0  0.0   0:00.10 ksoftirqd/1
    8 root      RT   0     0    0    0 S    0  0.0   0:00.00 watchdog/1
    9 root      RT   0     0    0    0 S    0  0.0   0:00.02 migration/2
   10 root      20   0     0    0    0 S    0  0.0   0:00.29 ksoftirqd/2
   ...
```

## kill

- La commande `kill` permet d'envoyer des signaux à un processus
- Par défaut cette commande force la terminaison d'un processus
- Le processus à terminer est passé comme argument à la commande `kill` au travers de son PID.

```
$ firefox &
$ ps -f | grep firefox
vincent  13036 11993 23 11:05 pts/1    00:00:04 /usr/lib/firefox-3.0.14/firefox
vincent  13082 11993  0 11:06 pts/1    00:00:00 grep firefox
$ kill 13036
$ ps -f | grep firefox
vincent  13091 11993  0 11:06 pts/1    00:00:00 grep firefox
```

```
icub@icubTest:~$ kill -l
 1) SIGHUP  2) SIGINT  3) SIGQUIT  4) SIGILL  5) SIGTRAP
 6) SIGABRT  7) SIGBUS  8) SIGFPE  9) SIGKILL 10) SIGUSR1
11) SIGSEGV 12) SIGUSR2 13) SIGPIPE 14) SIGALRM 15) SIGTERM
16) SIGSTKFLT 17) SIGCHLD 18) SIGCONT 19) SIGSTOP 20) SIGTSTP
21) SIGTTIN 22) SIGTTOU 23) SIGURG 24) SIGXCPU 25) SIGXFSZ
26) SIGVTALRM 27) SIGPROF 28) SIGWINCH 29) SIGIO 30) SIGPWR
31) SIGSYS 34) SIGRTMIN 35) SIGRTMIN+1 36) SIGRTMIN+2 37) SIGRTMIN+3
38) SIGRTMIN+4 39) SIGRTMIN+5 40) SIGRTMIN+6 41) SIGRTMIN+7 42) SIGRTMIN+8
43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9 56) SIGRTMAX-8 57) SIGRTMAX-7
58) SIGRTMAX-6 59) SIGRTMAX-5 60) SIGRTMAX-4 61) SIGRTMAX-3 62) SIGRTMAX-2
63) SIGRTMAX-1 64) SIGRTMAX
```

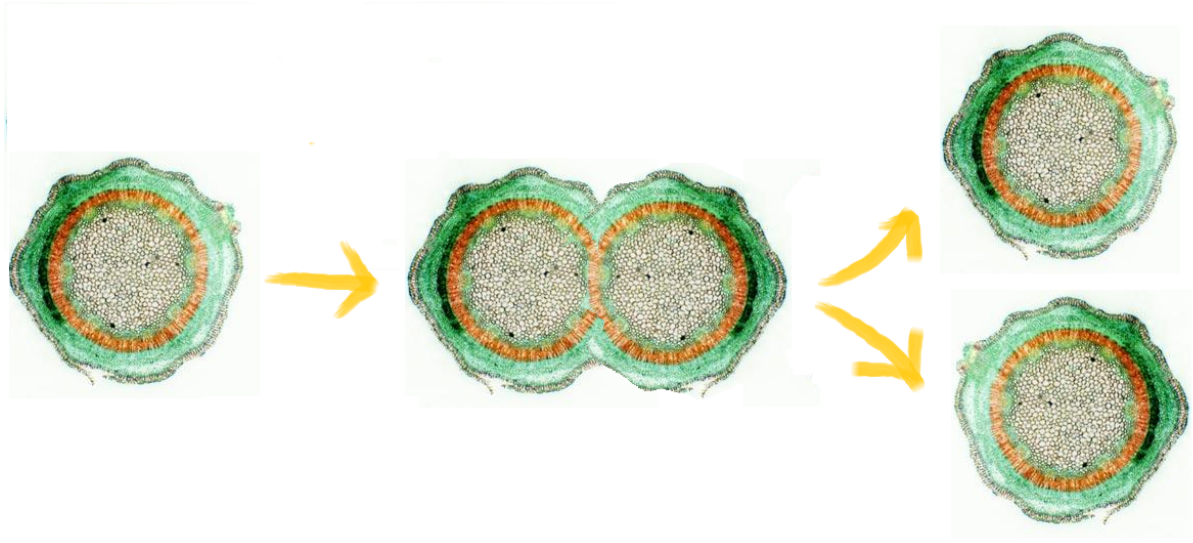
## pstree

La commande pstree permet de visualiser les processus en cours d'exécution par l'OS, organisés dans un arbre qui montre les liens parent/enfant

```
icub@icubTest:~$ pstree --help
pstree: unrecognized option '--help'
Usage: pstree [ -a ] [ -c ] [ -h | -H PID ] [ -l ] [ -n ] [ -p ] [ -u ]
              [ -A | -G | -U ] [ PID | USER ]
       pstree -V
Display a tree of processes.

  -a, --arguments     show command line arguments
  -A, --ascii         use ASCII line drawing characters
  -c, --compact       don't compact identical subtrees
  -h, --highlight-all highlight current process and its ancestors
  -H PID,
  --highlight-pid=PID highlight this process and its ancestors
  -G, --vt100         use VT100 line drawing characters
  -l, --long          don't truncate long lines
  -n, --numeric-sort  sort output by PID
  -p, --show-pids     show PIDs; implies -c
  -u, --uid-changes   show uid transitions
  -U, --unicode       use UTF-8 (Unicode) line drawing characters
  -V, --version       display version information
  PID    start at this PID; default is 1 (init)
  USER   show only trees rooted at processes of this user
```

```
icub@icubTest:~$ pstree -A -p
init(1)-+-NetworkManager(673)-+-dhclient(699)
        |                     `-{NetworkManager}(703)
        |-avahi-daemon(674)---avahi-daemon(679)
        |-clock-applet(1434)
        |-cron(839)
        |-cupsd(2875)
        ...
        |-gdm-binary(687)-+-gdm-simple-slav(783)-+-Xorg(1025)
        |                 |                      |-gdm-session-wor(1107)-+-gnome-session(1223)-+-bluetoot
        |                 |                      |                       |                      |-compiz(1
        |                 |                      |                       |                      |-evolutio
        |                 |                      |                       |                      |-gdu-noti
        |                 |                      |                       |                      |-gnome-pa
        |                 |                      |                       |                      |-gnome-po
        |                 |                      |                       |                      |-nautilus
        |                 |                      |                       |                      |-nm-apple
        |                 |                      |                       |                      |-polkit-g
        |                 |                      |                       |                      |-python(1
        |                 |                      |                       |                      |-ssh-agen
        |                 |                      |                       |                      |-update-n
        |                 |                      |                       |                      `-{gnome-s
        |                 |                      |                       `-{gdm-session-wo}(1224)
        |                 |                      `-{gdm-simple-sla}(1026)
        |                 `-{gdm-binary}(789)
        |-gedit(3193)
        |-gnome-terminal(2914)-+-bash(3124)
        |                      |-bash(3159)---pstree(3275)
        |                      |-gnome-pty-helpe(2915)
        |                      `-{gnome-terminal}(2917)
        |-rtkit-daemon(1115)-+-{rtkit-daemon}(1116)
        |                    `-{rtkit-daemon}(1117)
        |-sshd(617)
        .....
```

# fork



## Description

- La fonction `fork()` permet à un processus (programme en cours d'exécution) de créer un nouveau processus.
- Le nouveau processus (**enfant**) se voit attribuer un PID qui lui est propre et s'exécute de manière concurrente avec le processus **parent** qui l'a créé et dont le PID reste inchangé.
- Le processus parent et le processus enfant ont le même code source mais ne partagent pas leurs variables au cours de l'exécution.
- Tout processus Linux (excepté le processus racine de PID 0) est créé à l'aide de cet appel système.

## Fonctionnement

- A l'issu de l'exécution de l'appel `fork()` par le processus parent, chaque processus reprend son exécution au niveau de l'instruction suivant le `fork()`.
- Afin de différencier les traitements à réaliser dans le cas du processus parent et du processus enfant, on utilise la valeur de retour de la fonction `fork()` :
  - Si l'appel à `fork()` échoue, le processus enfant n'est pas créé et la valeur de retour est $-1$.
  - Dans le processus enfant, la valeur de retour vaut 0.
  - Dans le processus parent, la valeur de retour vaut le PID de l'enfant qui vient d'être créé.

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid (void);
pid_t getppid (void);
pid_t fork (void);
```

- `fork()` : create a new process
- the new process (child) is an exact copy of the calling process (parent)
- the child has a new unique process ID, and a new (of course) parent process ID
- the child process inherits a copy of the parent's file descriptors, message catalog descriptors, semaphores, ...
- the child does not inherit its parent's memory locks

**** fork_simple.c ****

```c
#include <unistd.h>     // fork()
#include <sys/types.h>  // pid_t
#include <stdio.h>      // printf()

int main(void) {

        pid_t retour = 0;
        int x = 42;
        retour = fork();

        switch(retour){
        case 0 :
                printf("Je suis le processus enfant, de PID %d.\n",getpid()
                printf("Le PID de mon parent est %d.\n",getppid());
                printf("enfant : x = %d\n", x);
                break;
        case -1 :
                printf("Echec de la creation d'un processus enfant\n.");
                break;
        default :
                printf("Je suis le processus parent, de PID %d.\n",getpid()
                printf("Le PID de mon enfant est %d.\n",retour);
                x = 5;
                printf("parent : x = %d\n", x);
                break;
        }

        while(1);
        /*{
        if(retour == 0)
```

The output of the process :

```
$ ./fork_simple
Je suis le processus parent, de PID 1818.
Le PID de mon enfant est 1819.
Je suis le processus enfant, de PID 1819.
Le PID de mon parent est 1818.
```

The process tree :

```
icub@icubTest:~$ pstree -p 1818
fork_simple(1818)---fork_simple(1819)
```

See the process tree :

```
icub@icubTest:~$ pstree -p 1869
fork_simple(1869)---fork_simple(1870)
```

## ctrl+z

**Ctrl+z** : the process is stopped, still alive. The process tree doesn't change.

```
^Z
[1]+  Stopped                 ./fork_simple
```

## ctrl+c

**Ctrl+c** : the process is closed

```
$ ./fork_simple
Je suis le processus parent, de PID 1869.
Le PID de mon enfant est 1870.
Je suis le processus enfant, de PID 1870.
Le PID de mon parent est 1869.
^C
```

after ctrl+c the process terminates (does not exist anymore)

```
icub@icubTest:~$ pstree -p 1869
icub@icubTest:~$
```

$\mapsto$ **different behavior when killing father or son !**

```
icub@icubTest:~$ ./fork_simple
Je suis le processus parent, de PID 1873.
Le PID de mon enfant est 1874.
Je suis le processus enfant, de PID 1874.
Le PID de mon parent est 1873.
```

```
icub@icubTest:~$ pstree -p 1873
fork_simple(1873)---fork_simple(1874)
```

**Kill the parent ... what happens to the pstree ?**

```
icub@icubTest:~$ kill 1873
icub@icubTest:~$ pstree -p 1873
icub@icubTest:~$ pstree -p 1874
fork_simple(1874)
```

... the parent is dead, the child still lives

↦ **different behavior when killing father or son !**

```
icub@icubTest:~$ ./fork_simple
Je suis le processus parent, de PID 1873.
Le PID de mon enfant est 1874.
Je suis le processus enfant, de PID 1874
Le PID de mon parent est 1873.
```

```
icub@icubTest:~$ pstree -p 1873
fork_simple(1873)---fork_simple(1874)
```

**Kill the child ... what happens to the pstree ?**

```
icub@icubTest:~$ kill 1874
icub@icubTest:~$ pstree -p 1873
fork_simple(1873)---fork_simple(1874)
```

.. the child process is **zombie** (because the father does not know it has been terminated)

**Then kill the father ... what happens ?**

```
icub@icubTest:~$ kill 1873
icub@icubTest:~$ pstree -p 1873
icub@icubTest:~$
```

.. both processes are now terminated

## Synchronisation

- La terminaison du processus parent **n'entraine pas** la terminaison de ses enfants.
- Lorsqu'un processus enfant se termine avant son parent, il passe à l'état **zombie**.
- Afin de terminer complètement ce processus enfant zombie et donc libérer les ressources associées, le processus parent peut faire appel à une fonction de synchronisation de type `wait()` ou `waitpid()`.

## `wait()`

- `pid_t wait(int *status)` suspend l'exécution du processus parent (appelant) jusqu'à ce que l'un de ses enfants se termine.
- `pid_t waitpid(pid_t pid, int *status, int options)` permet de spécifier le PID de l'enfant dont la fin est attendue.
- La synchronisation du parent et de l'ensemble de ses enfants nécessite donc autant d'appel à `wait()` que d'enfants.
- L'argument `status` permet de récupérer une information relative à l'exécution de l'enfant dont on l'attend la terminaison.
- Cette information peut être transmise par le processus enfant au moyen de la fonction `void exit(int status)` qui permet la terminaison normale du processus.

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

- `wait(int *status)` : suspends execution of the calling process until one of its children terminates
  - on success, it returns the ID of the terminated child ; on error, it returns *-1*
  - note that in case of a terminated child, if the parent process does not call `wait()`, the resources associated to the child cannot be released, hence the terminated child remains in a "zombie" state (can be seen in the system by typing `ps aux` and looking for a "Z" in the STAT column)
  - the kernel maintains a minimal set of information about the zombie process (PID, termination status, resource usage information) in order to allow the parent to later perform a wait to obtain information about the child. As long as a zombie is not removed from the system via a wait, it will consume a slot in the kernel process table, and if this table fills, it will not be possible to create further processes. If a parent process terminates, then its "zombie" children (if any) are adopted by `init()`, which automatically performs a `wait` to remove the zombies.
- `waitpid(pid_t pid, int *status, int options)` : suspends execution of the calling process until a child specified by *pid* has changed state
  - *pid* = *-1* wait for any child process
  - *pid* = *0* wait for any child process whose group ID is equal to that of the calling process
  - *pid* > *0* wait for the child process with process ID equal to *pid*
  - by default it waits for terminated children, but can be modified by *options*
  - for example *options* = *WNOHANG* return immediately if no child has exited
  - on success, it returns the ID of the child whose state has changed ; on error, it returns *-1* ; if *WNOHANG* is specified and one or more children specified by *pid* exist, but have not yet changed state, it returns *0*

Note that :

```
wait(&status) == waitpid(-1, &status, 0)
```

**** fork_wait.c ****

```
#include <unistd.h>// fork()
#include <sys/wait.h>// waitpid()
#include <stdio.h>// printf()
#include <stdlib.h>// exit(), srand()
#include <time.h>// sleep()

int hasard() {

double valeur = 0.0;

srand(time(NULL));

valeur = (double) rand() / RAND_MAX;
printf("Le hasard dit : %f\n",valeur);

if( valeur > 0.5)
return(0);
else
return(1);
}

int main(void) {

pid_t retour = 0;
sleep(20);
retour = fork();

//2 valeurs possibles 0 ou > 0; pour un code portable
// on utilise les constantes EXIT_SUCCESS ou EXIT_FAILURE
int exit_status = EXIT_FAILURE;
```

**** fork_wait.c ****

```
switch(retour){
case 0 :
printf("Je suis le processus enfant de PID %d.\n",getpid());
printf("Le PID de mon parent est %d.\n",getppid());
sleep(30);
if( hasard() == 0) exit(EXIT_SUCCESS);
else exit(EXIT_FAILURE);
break;
case -1 :
printf("Echec de la creation d'un processus enfant\n.");
exit(EXIT_FAILURE);
break;
default :
printf("Je suis le processus parent de PID %d.\n",getpid());
printf("Le PID de mon enfant est %d.\n",retour);

printf("J'attends que mon enfant %d se termine.\n",retour);
waitpid(retour,&exit_status,0);

if(exit_status == EXIT_SUCCESS)
printf("Mon enfant %d a termine avec succes.\n",retour);
else
printf("Mon enfant %d a termine en echec.\n",retour);
exit(exit_status);
break;
}
}
```

**** fork_wait.c ****

Output :

```
$ ./fork_wait
Je suis le processus parent de PID 1885.
Le PID de mon enfant est 1886.
J'attends que mon enfant 1886 se termine.
Je suis le processus enfant de PID 1886.
Le PID de mon parent est 1885.
Le hasard dit : 0.217699
Mon enfant 1886 a termine en echec.
```

# TEST 1

**** fork_3.c ****

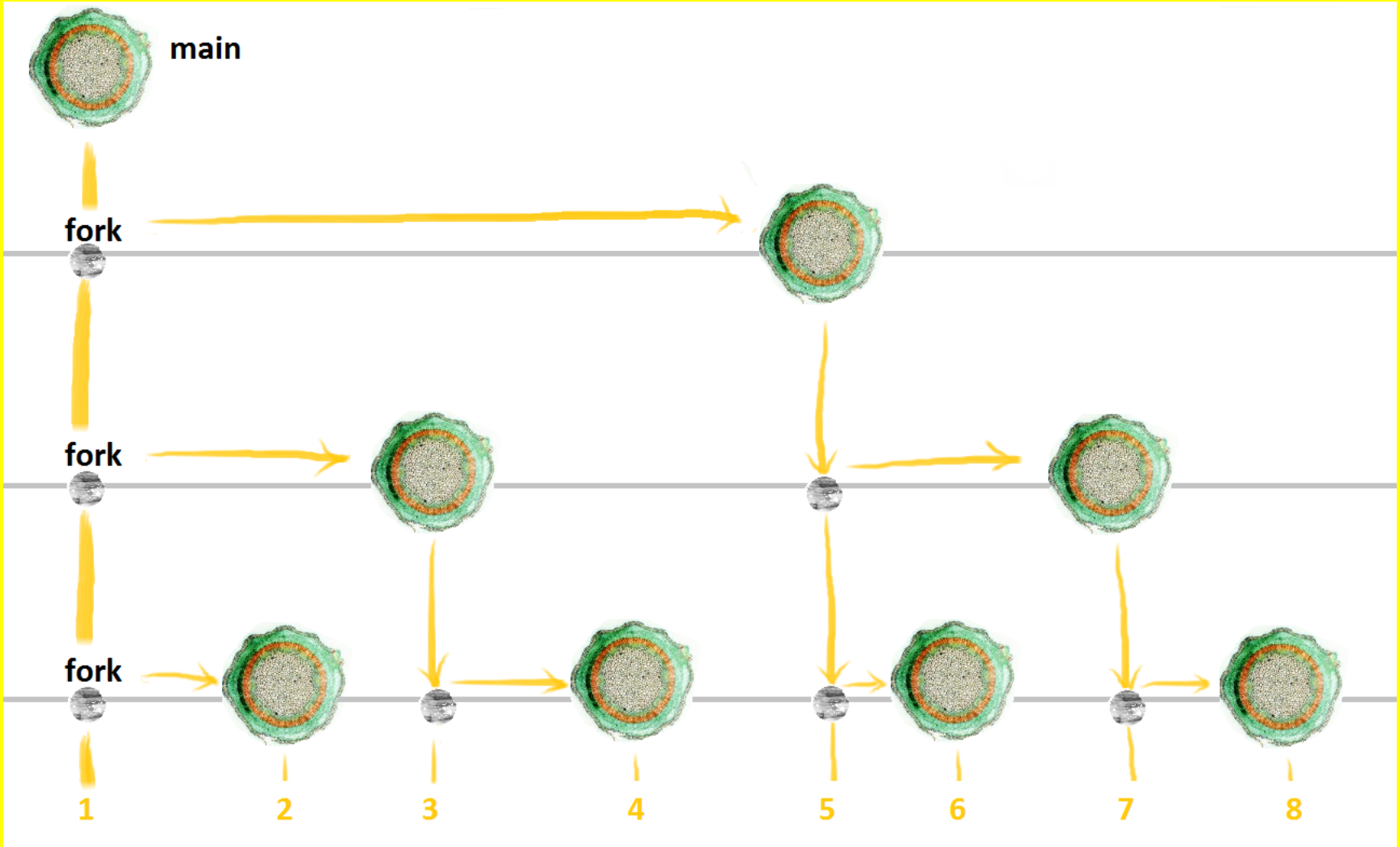
```c
#include <unistd.h>
#include <stdlib.h>

int main(void)
{
        fork();
        fork();
        fork();

        while(1);

        exit(0);
}
```

On exécute ce programme. Combien de processus sont créés ?
8 : *main* (le premier fork_3) et 7 enfants

```
|-gnome-terminal(1521)-+-bash(1523)---pstree(1900)
       |                     |-bash(1726)-+-fork_3(1889)-+-fork_3(1890)-+-fork_3(1892)---fork_3(1896)
       |                     |            |              |              `-fork_3(1894)
       |                     |            |              |-fork_3(1891)---fork_3(1895)
       |                     |            |              `-fork_3(1893)
       |                     |            `-fork_simple(1818)---fork_simple(1819)
```

# TEST 1

Graphical representation :



# TEST 2

**** fork_test2.c ****

```c
#include <unistd.h>      // fork()
#include <stdio.h>       // printf()

int main(void)
{
        int i=0;
        fork();
        fork();
        fork();
        printf("hello world - %d \n",i++);
        return 0;
}
```

Quelle est la sortie du programme suivant ?

```
hello world - 0
hello world - 0
hello world - 0
hello world - 0
hello world - 0
hello world - 0
hello world - 0
hello world - 0
```

# TEST 3

Ecrivez un programme qui genère seulement trois processus et affiche le PID du parent et des enfants.

**** fork_test3.c ****

```c
#include <unistd.h>     // fork()
#include <sys/types.h>  // pid_t
#include <stdio.h>      // printf()

int main(void) {

        pid_t retour = 0;
        retour = fork();

        switch(retour){
        case 0 :
                fork();
                printf("ENFANT: mon PID=%d, ",getpid());
                printf("PID parent=%d \n",getppid());
                break;
        case -1 :
                printf("Echec de la creation d'un enfant\n.");
                break;
        default :
                printf("PARENT: mon PID=%d, ",getpid());
                printf("PID enfant=%d \n",retour);
                break;
        }

        while(1);
        return 0;
}
```

## TEST 3

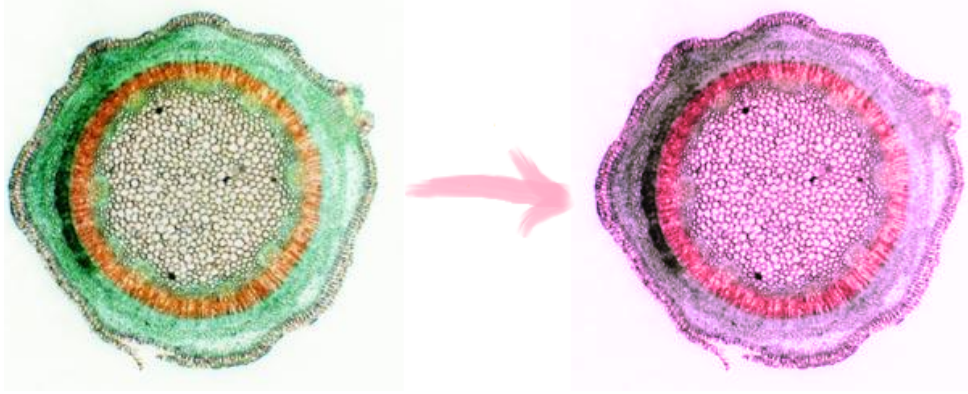
Output :

```
PARENT: mon PID=2448, PID enfant=2449
ENFANT: mon PID=2449, PID parent=2448
ENFANT: mon PID=2450, PID parent=2449
```

Process tree :

```
fork3(2448)---fork3(2449)---fork3(2450)
```

**exec***



## Principe

- Les primitives de recouvrement constitue un ensemble d'appels système (de type `exec*()`) permettant à un processus de charger en mémoire un nouveau code exécutable.
- Le code de remplacement, spécifié comme argument de l'appel système de type `exec*()`, écrase le code du processus en cours (lui même éventuellement hérité au moment de la création du processus par `fork()`).
- Des données peuvent être passées au nouveau code exécutable qui les récupère via les arguments de la fonction `exec*()` utilisée.
- Ces données sont récupérées au travers du tableau `argv[]`.

## La famille `execv()`

- Il existe 6 primitives de recouvrement de type `exec*()`.
- Leur différence principale réside dans la manière de passer le code de remplacement.
- De manière générale, il s'agit d'une chaine de caractère consitutant le chemin vers l'exécutable correspondant.
- Ces fonctions retournent $-1$ lorsqu'une erreur s'est produite (normalement si le recouvrement se passe bien, le code d'origine n'est plus exécuté et `execv()` ne doit pas "revenir").
- L'utilisation de ces primitives nécessitent l'inclusion de `unistd.h`.

```
#include <unistd.h>

int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```

- `exec*()` functions replace the current process image with a new process image
- *file* must be either a binary executable, or a script starting with a line of the form `#!` *interpreter* [*optional-args*]
- *argv* is an array of arguments passed to the new program
  - conventionally, the first argument is the filename associated with the file being executed
  - the array of pointers must be terminated with a NULL pointer
- `exec*()` functions only return if an error has occurred

## Principe

- `exec*()` allows starting a program, but doesn't create a new process
- Can be seen as a **process mutation**

A typical scheme :

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

int main(){
  execl("/bin/ls","ls",NULL) ;
  printf("Cette commande n'existe plus du fait de la mutation\n") ;
  return 0 ;
}
```

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

- Une des primitives de recouvrement.
- Ses arguments d'entrées ne sont pas modifiables (mot clé `const`).
- `path` représente le chemin (relatif ou absolu) du programme exécutable de remplacement.
- `argv[]` contient les arguments du code de recouvrement dans un format similaire à celui des arguments du `main()` : `arvg[0]` contient la chaine de caractère correspondant au nom du fichier exécutable, `arvg[1]` la chaine de caractère correspondant au premier argument, ...
- En l'absence d'argument de type `argc`, la dernière case de `argv` doit contenir `NULL` afin de pouvoir déterminer la fin de la liste d'arguments.

**** execv_simple.c ****

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

int main (void)
{
    char *argv1[3];

    switch (fork())
    {
        case -1:
            puts("Erreur");
            exit(-1);
        case 0: // enfant
            argv1[0] = "ls";
                       argv1[1] = "-l";
            argv1[2] = NULL; // toujours terminer avec NULL
            execv("/bin/ls", argv1);
            exit(0);
        default:
            puts("\nJe␣suis␣le␣parent\n");
            wait(NULL);
            break;
    }
    return 0;
}
```

What is the output of the following program ?

Output :

```
icub@eva:~/InfoSys_2012/c2/code$ ./execv_simple

Je suis le parent

total 248
-rwxr-xr-x 1 icub icub 10076 2012-08-16 16:51 appel_syst
-rw-r--r-- 1 icub icub   179 2012-08-14 16:06 appel_syst.c
-rw-r--r-- 1 icub icub  4104 2012-08-16 16:51 appel_syst.o
-rwxr-xr-x 1 icub icub  9889 2012-08-16 16:51 compte_args
-rw-r--r-- 1 icub icub   221 2012-08-14 16:06 compte_args.c
-rw-r--r-- 1 icub icub  3984 2012-08-16 16:51 compte_args.o
drwxr-xr-x 3 icub icub  4096 2012-08-16 16:58 exec
-rwxr-xr-x 1 icub icub 10108 2012-08-16 16:51 execv_2
-rw-r--r-- 1 icub icub   456 2012-08-16 16:04 execv_2.c
...
```

```
#include <stdlib.h>

int system(const char *command);
```

- `system()` executes a shell command, specified by calling `/bin/sh -c` *command*
- returns after the command has been completed
  - $-1$ if there is an error (e.g. a fork fail)
  - on success, the return status of the command, in the `wait()` format (`WEXITSTATUS(status)`, that is the exist status of the child

**** appel_syst.c ****

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

int main(void) {

        // Lance gedit sur le fichier "fichier.txt"
        system("gedit fichier.txt");
    printf("apres system\n");

        exit(0);
}
```

## Remarque

Si la commande est dans le PATH, pas besoin de spécifier le chemin (au contraire de `execv()`).

# TEST 4

**** compte_args.c **** a

```
1  #include <stdio.h>
2
3  int main(int argc, char* argv[]){
4
5          printf("Ce programme s'appelle %s et il compte le nombre d'
               arguments qui lui sont passes.\n",argv[0]);
6          printf("Ici ce nombre vaut : %d\n",argc-1);
7
8          return 0;
9  }
```

Given the above program, counting the number of arguments, can we write a simple program to execute it ?

# TEST 5

**** program_info.c ****

```c
#include <unistd.h>     // getpid(), gettpid()
#include <sys/types.h>  // pid_t, uid_t, gid_t
#include <stdio.h>      // printf()
#include <stdlib.h> // exit

int main(int argc, char* argv[])
{
        if(argc != 2)
        {
        puts("\nUsage : ./program_info name\n");
        exit(-1);
        }

        pid_t my_pid  = getpid();
        pid_t my_ppid = getppid();

        printf("%s:\n",argv[1]);
        printf("pid = %d\n",my_pid);
        printf("ppid = %d\n",my_ppid);
        return 0;
}
```

Given the above program, providing information about a process, can we write a simple program that generates two children that mutate to print their information ?

For example, Anakin (father - main) creates two children, Luke and Leia. We want to know their *pid* and *ppid* exploiting `program_info`.

# TEST 5

**** exec_launcher.c ****

```c
// the main creates two children, both mutates in program_info

#include <stdio.h> //printf
#include <stdlib.h>
#include <unistd.h> // fork
#include <sys/wait.h>
#include <sys/types.h>// pid_t


int main (void)
{
        char *argv1[3];
        char *argv2[3];

        argv1[0] = "program_info"; argv1[1] = "Luke"; argv1[2] = NULL;
        argv2[0] = "program_info"; argv2[1] = "Leia"; argv2[2] = NULL;

        pid_t my_pid  = getpid();
        pid_t my_ppid = getppid();

        printf("Anakin:\n");
        printf("pid = %d\n",my_pid);
        printf("ppid = %d\n",my_ppid);

        printf("I will now generate my children ..\n\n");

        my_pid = fork();
        if(my_pid == -1)
        {
                puts("Fork error on first child"); exit(-1);
        }
```

# TEST 5

**** exec_launcher.c ****

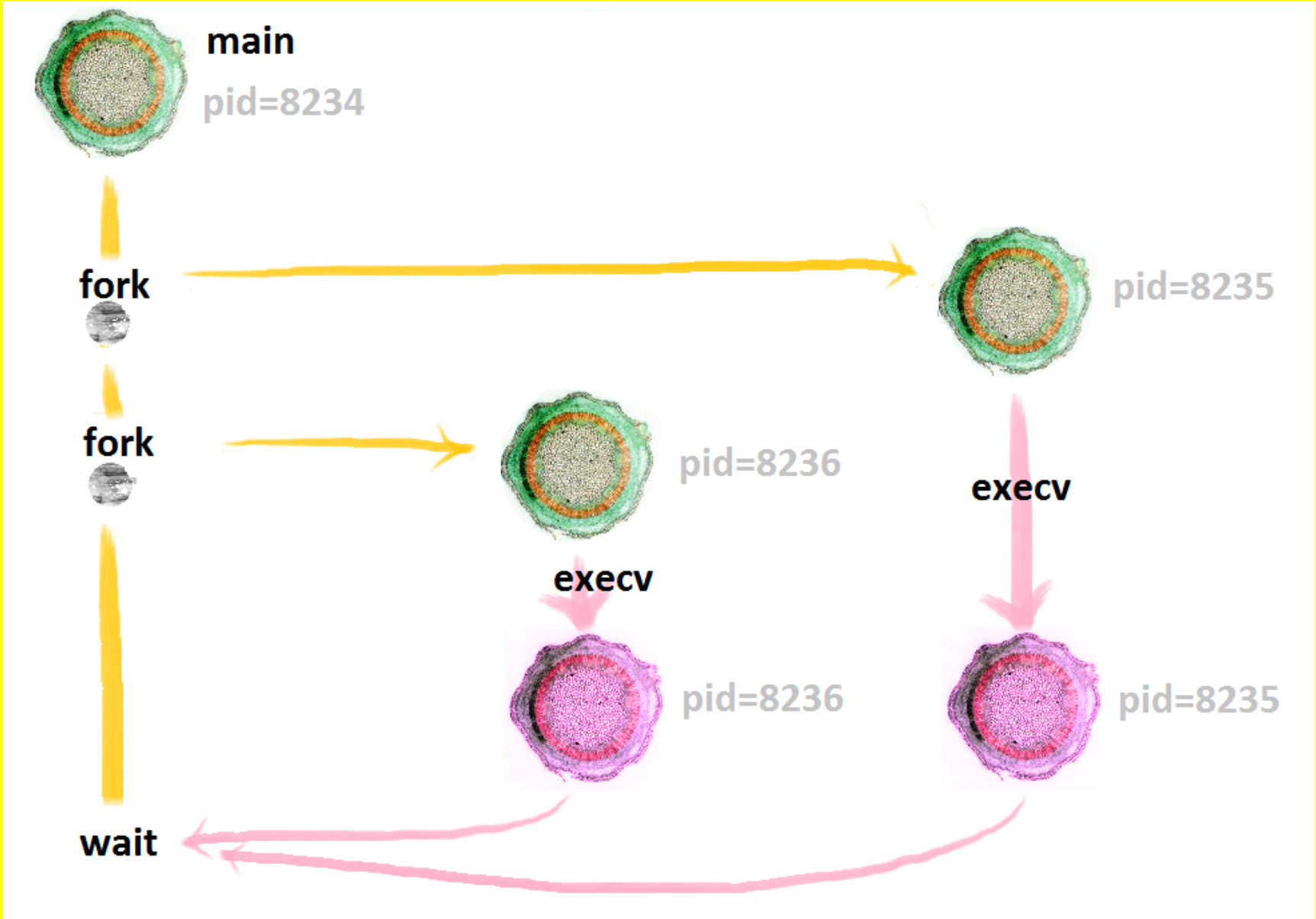
```
1         else if(my_pid == 0) //first child
2         {
3                 printf("First child:\n");
4                 printf("pid = %d\n",getpid());
5                 printf("ppid = %d\n",getppid());
6                 execv("./program_info", argv1);
7                 puts("Luke created");
8                 exit(0);
9         }
10        else
11        {
12                my_pid = fork(); // go on and make the second child
13                if(my_pid == -1)
14                {
15                        puts("Fork error on second child"); exit(-1);
16                }
17                else if(my_pid == 0) //second child
18                {
19                        printf("Second child:\n");
20                        printf("pid = %d\n",getpid());
21                        printf("ppid = %d\n",getppid());
22                        execv("./program_info", argv2);
23                        puts("Leia created");
24                        exit(0);
25                }
26                else
27                {
28                        // the parent
29                }
30        }
31        puts("Anakin now waits the children ..\n");
32        wait(NULL);
33        wait(NULL);
34        puts("\nAnakin now dies ..\n");
35        return 0;
36  }
```

# TEST 5

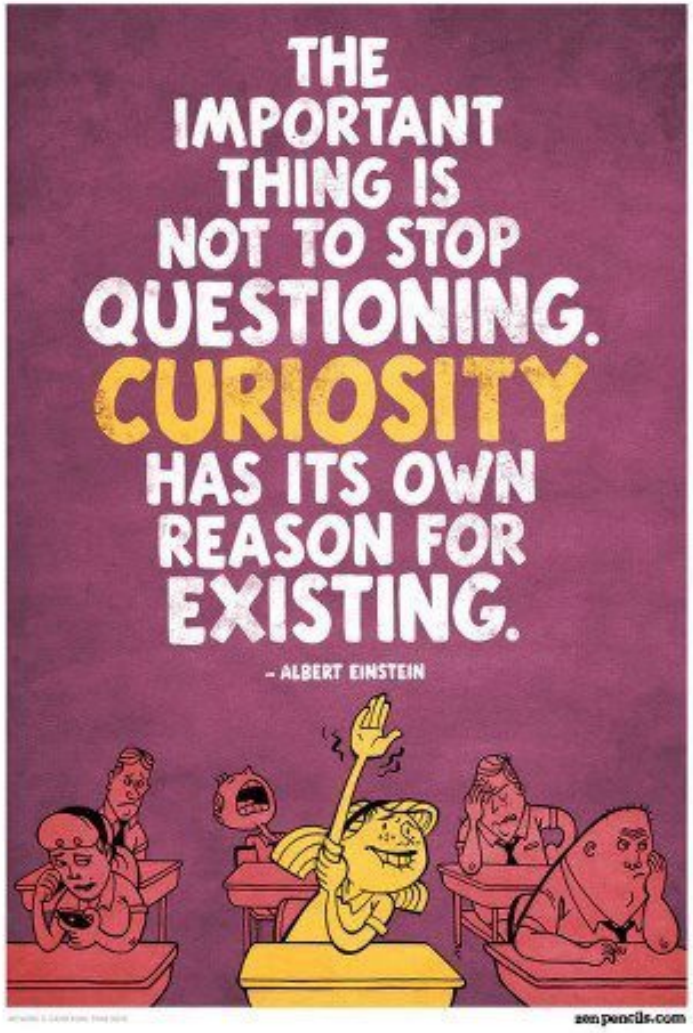
Note the output :

```
icub@eva:~/InfoSys_2012/c2/code/exec$ ./exec_launcher
Anakin:
pid = 8234
ppid = 1616
I will now generate my children ..
Anakin now waits the children ..
First child:
pid = 8235
ppid = 8234
Second child:
pid = 8236
ppid = 8234
Luke:
pid = 8235
ppid = 8234
Leia:
pid = 8236
ppid = 8234
Anakin now dies ..
```

# TEST 5


main
pid=8234
fork
fork
pid=8235
pid=8236
execv
execv
pid=8236
pid=8235
wait

# Questions ?


THE IMPORTANT THING IS NOT TO STOP QUESTIONING. CURIOSITY HAS ITS OWN REASON FOR EXISTING.
– ALBERT EINSTEIN
zenpencils.com