

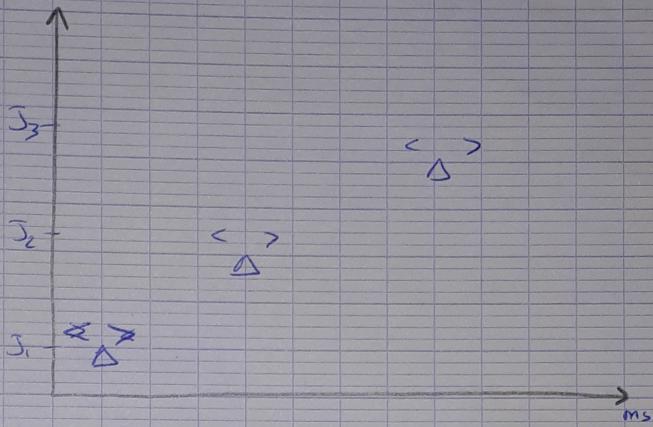
C D >

## Les processus :

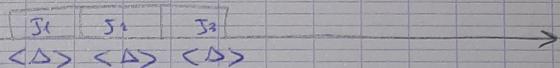
un processus est un programme en cours d'exécution  
un programme est une suite d'instructions

l'exécution de processus est séquentielle

l'impression de simultanéité est due à la multiprogrammation



si  $\Delta$  est suffisamment petit on a l'impression d'une exécution de plusieurs processus en même temps



une exécution concurrente réelle (2 ou plus processus en même temps) s'effectue sur une machine qui dispose de plusieurs processeurs

la création de processus a lieu lors :

- de l'initialisation du système
- de l'exécution de l'appel système de création de processus
  - une requête utilisateur
  - de l'initialisation d'un traitement par lot (script interpréteur de commande)

les processus ont plusieurs identifiants

- PID identifiant de processus
- UID identifiant utilisateur
- GID identifiant de groupe

Termination d'un processus

- Normal (exit (success))  
| return (0)
- erreur (exit (failure))
- erreur fatal (division par 0, erreur segment)
- arrêt par un autre processus

Gestion des processus

- premier init
- un processus est créé par ~~autre~~ un autre processus
- arborescence de processus
- visualisat° : ps, top, pstree
- suppression : kill

Créat° de processus par l'appel système fork()

```
int main (int argc, char *argv[])
{
    pid_t p
    int a=0;
    char c='b';
    ...
    p=fork();
    if (pid == 0)
        ...
}
```

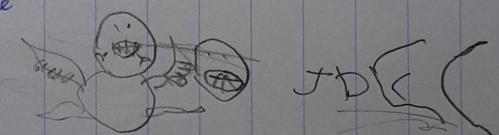
a = 0  
c = 'b'  
int a=0  
char c='b'  
execut° de fork()

processus fils  
(copie de l'espace de processus père)  
processus père

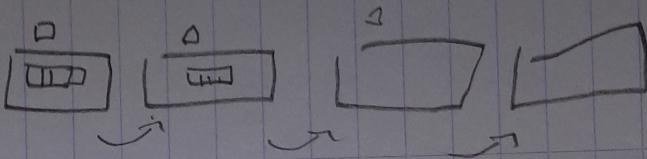
fork crée un processus copié du processus appelant  
la valeur de retour est :

- 0 dans le processus créé
- > 0 dans le processus appelant.

id du processus créé  
if (p == 0) {  
 ...
}



3



### Exercice :

écrire un programme qui crée 4 processus fils :  
chaq processus devra afficher son ordre

```
# include <unistd.h>
int main (int argc, char *argv[])
{
    pid_t *tabP = malloc (4 * sizeof (pid_t));
    for (int i=0; i<4; i++) {
        tabP[i] = fork();
        printf ("ordre = %d", (i+1));
    }
}
```

```
int main (int argc, char *argv[])
{
    pid_t p;
```

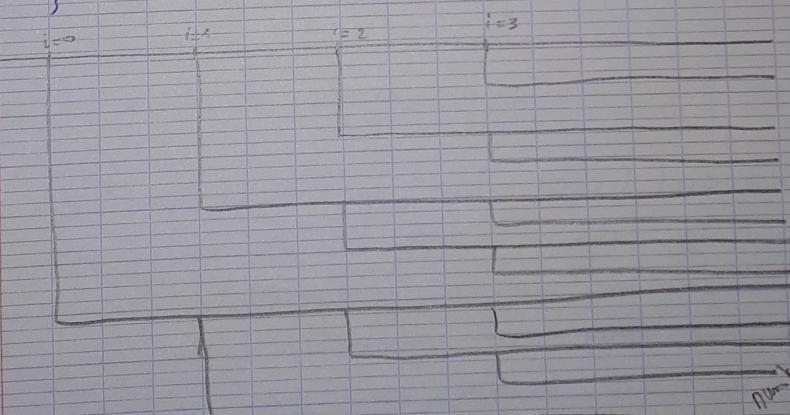
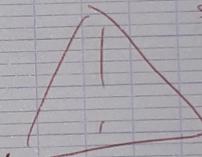
```
    for (int i=0; i<4; i++) {
        if ((p=fork()) == 0) {
```

```
            printf ("%d\n", i);
            exit(0);
```

on doit garder  
sinon il affiche 16  
processus =

```
    }
}
```

```
exit(0);
```

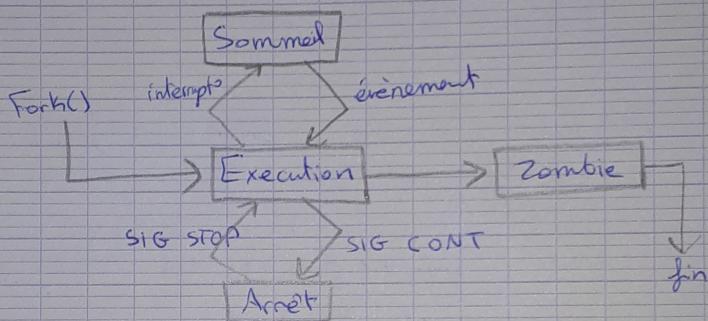


de  
un  
un

### Etat des processus:

les processus peuvent être dans les états suivants :

- Exécution (R)
- Sommeil (S)
  - non interruptible logiquement
  - interruptible
- Arrêt
- Zombie



### Exécution d'un programme:

on utilise la famille d'appels système "exec"

int execv (const char \* app, const char \* argv[])

avec app: chemin complet du programme à exécuter.

argv: permettre de donner à l'application

int exec (const char \* app, const char \* argv, const char \* argc, ...)

avec app: ... (pareil)

argc: 1<sup>e</sup> argument, ...

on termine par NULL

### Exercice

Ecrire un programme qui execute ps :

```
#include <unistd.h>
int main ( int argc, char * argv[3] ) {
    exec( " /bin/ps ", " ps ", NULL );
    exit(0);
}
```

↑ on le met si exec termine en erreur. sinon il n'est pas exec

### Exercice

Ecrire un programme qui execute ps puis ls :

```
int main ( int argc, char * argv[3] ) {
    pid_t p,
    if ( (p = fork()) == 0 ) {
        exec( " /bin/ps ", " ps ", NULL );
        exit(0);
    }
    exec( " /bin/ls ", " ls ", NULL );
    exit(0);
}
```