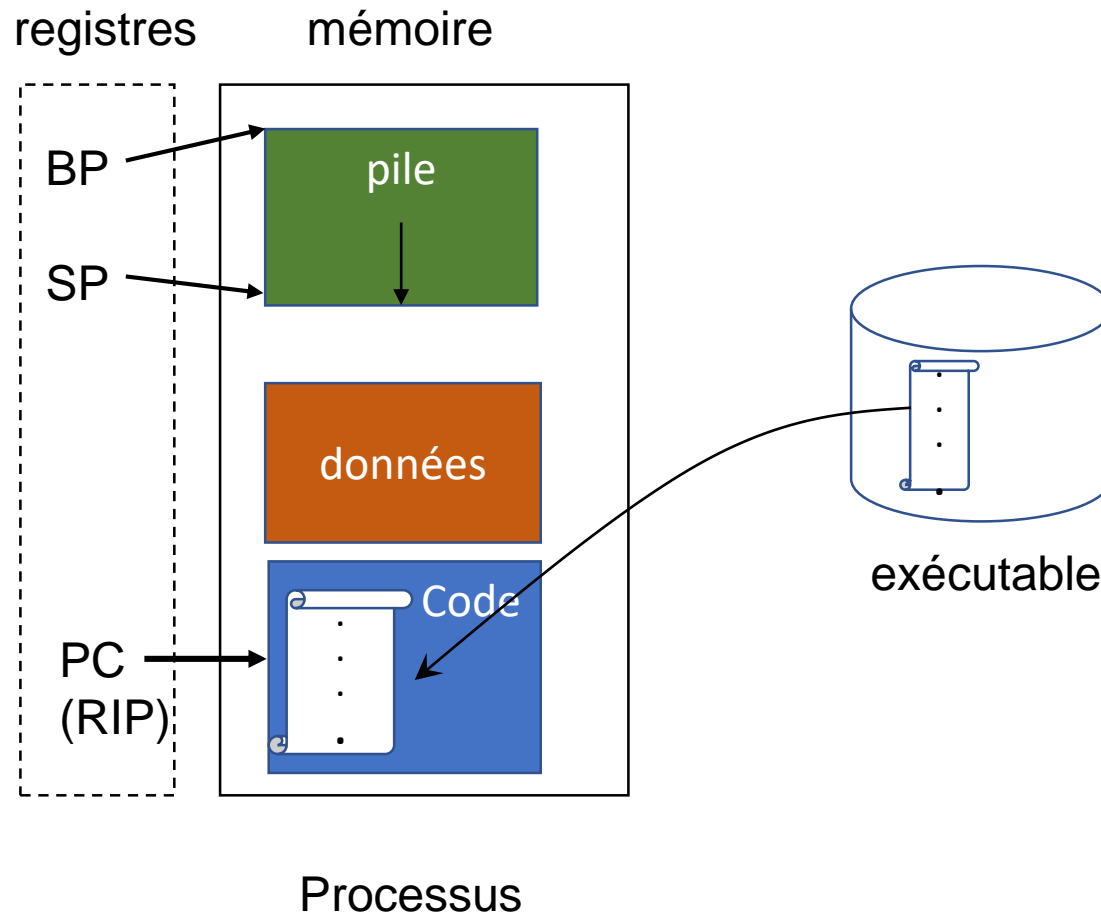


Cours 4 : Processus Unix

Processus

- Processus: entité active du système
 - Correspond à l'exécution d'un programme binaire
 - Chaque processus est indépendant
 - CPU est partagé (temps partagé) => Commutation entre les processus
- Identifié de façon unique par son numéro : **pid**
 - Possède un contexte
 - Mémoire : 3 segments (code, données et pile)
 - Matériel : ensemble des registres
 - Exécuté sous l'identité d'un utilisateur :
 - propriétaire réel **uid** et effectif **euid**
 - Groupe réel et effectif
 - Possède un répertoire courant

Processus : contexte



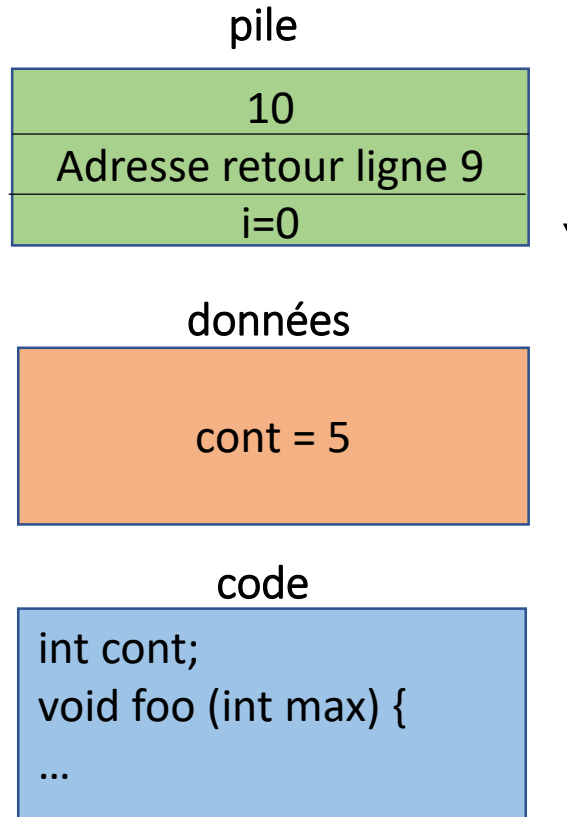
Processus : Mémoire

```
1: int cont;

2: void foo (int max) {
3:   int i;
4:   for (i=0; i++; i<max)
5:     printf ("%d \n", i);
6: }

7: int main (int argc, char* argv []) {
8:   cont=5;
9:   foo(10) ;

10: return EXIT_SUCCESS;
11: }
```



Etats d'un processus

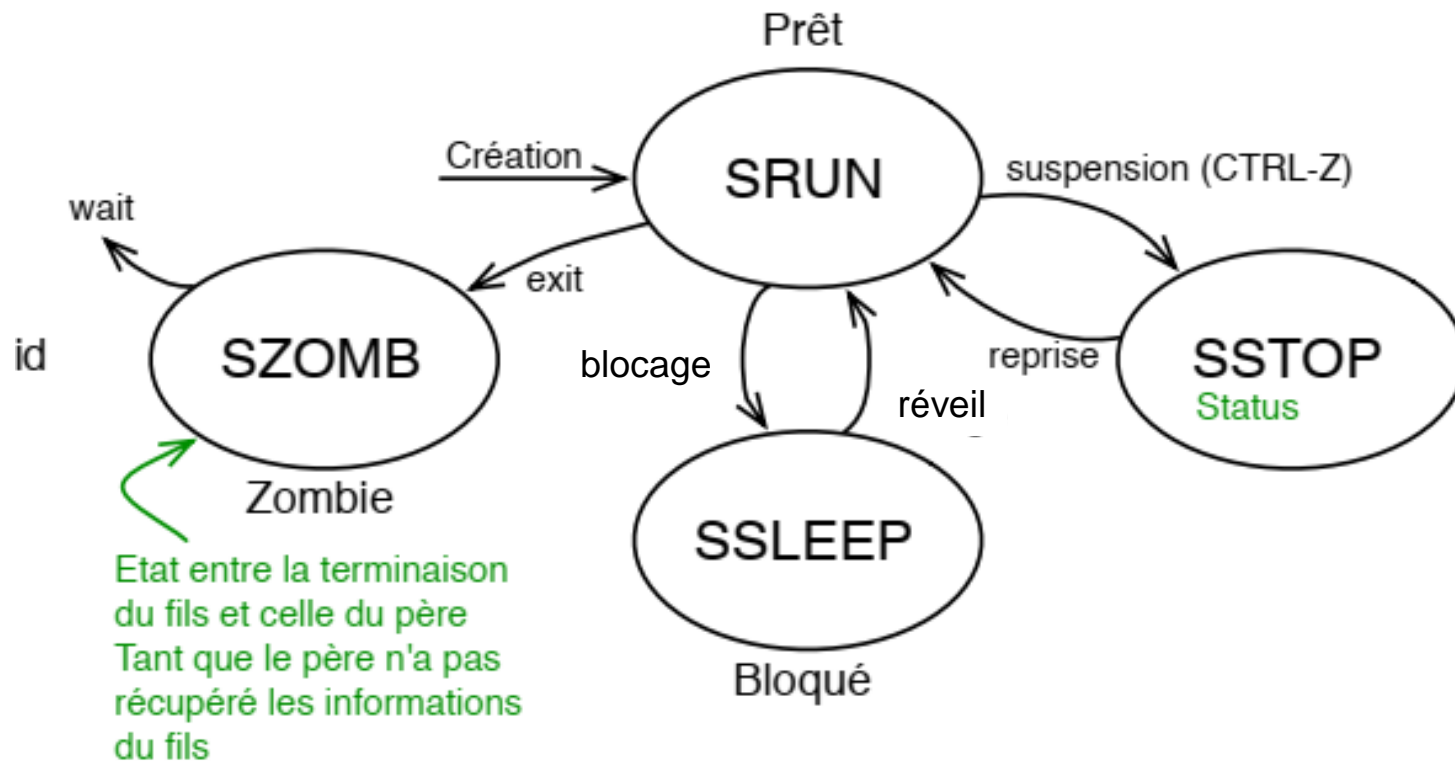
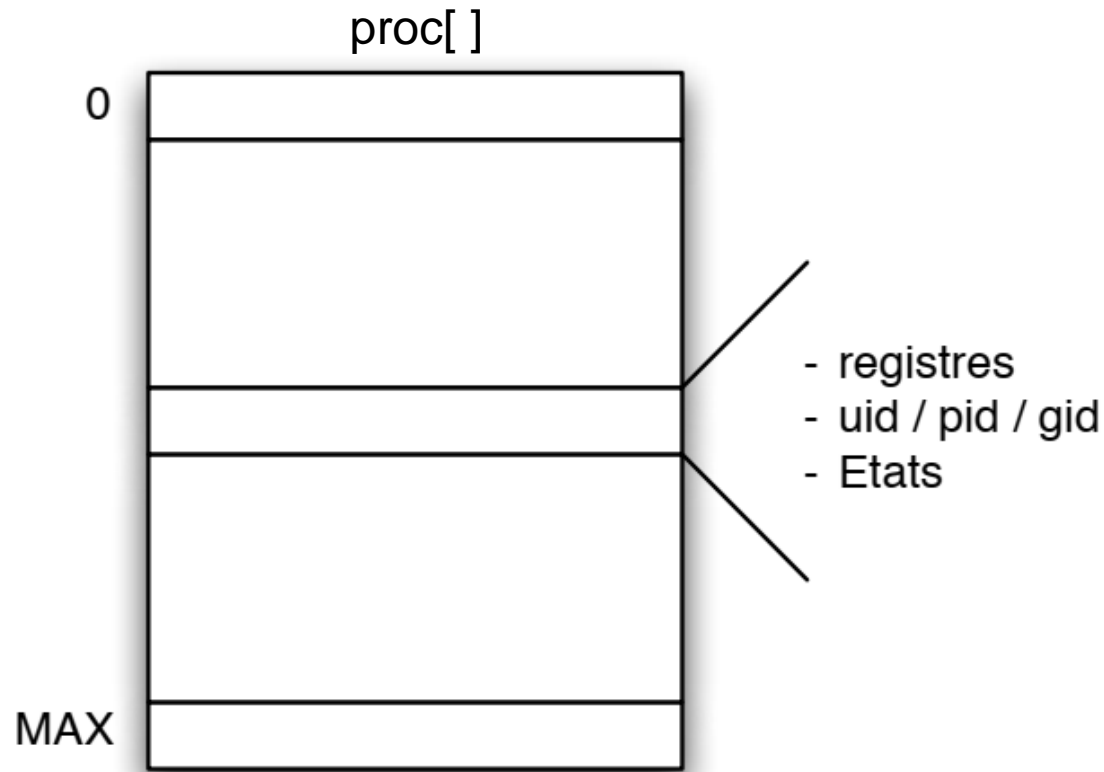


Table de processus

- Le système maintient une table des processus en cours (proc[] ou task[] dans linux) pour stocker les informations sur les processus



Programmation des processus

```
#include <sys/types.h>           pid_t
#include <unistd.h> }
#include <stdlib.h> }           prototype fonctions standards
```

```
int main(int argc, char *argv[], [char *envp] ) {
    ...                               0 => OK (EXIT_SUCCESS)
    return ret;
}                               ≠ 0 => KO (EXIT_FAILURE)
```

argc : nombre d'arguments

argv : liste des arguments

argv[0] = nom exécutable

argv[1] = 1^{er} argument

...

argv[argc-1] = dernier argument

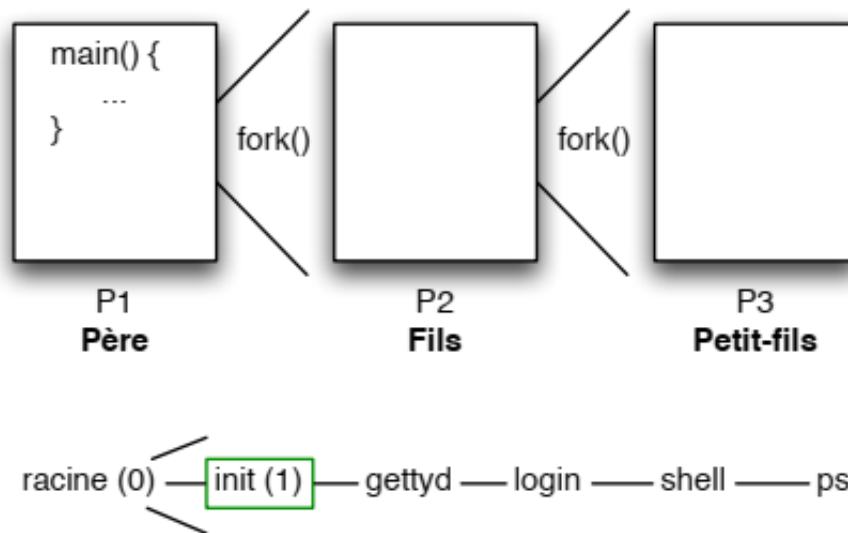
argv[argc] = NULL

envp : liste des variables d'environnement

Création de processus

- Un processus **Père** peut créer un processus un processus **Fils**

=> un arbre de processus

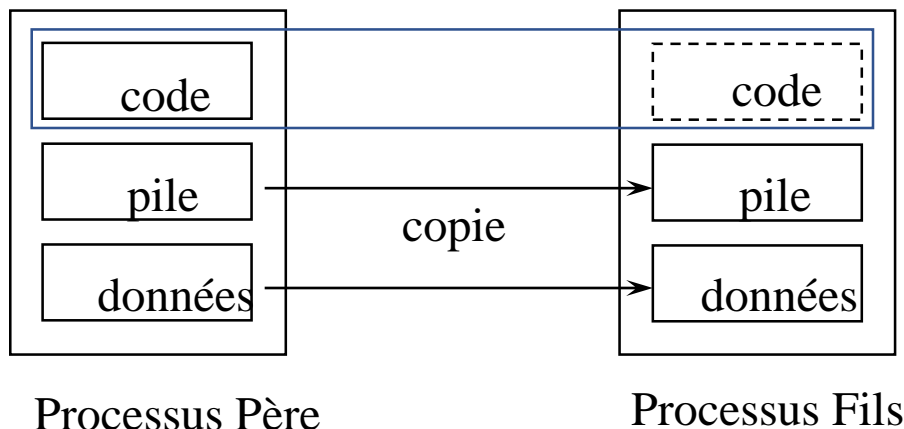


Fork : création d'un processus

- Création dynamique d'un nouveau processus (*fils*) qui s'exécute de façon concurrente avec le processus qui l'a créé (*père*).

`pid_t fork (void)`

- Processus fils est **une copie du processus**
 - variables du fils possèdent les mêmes valeurs que celles du père au moment du *fork*
 - toute modification d'une variable par l'un des processus n'est pas visible par l'autre.



Fork : création d'un processus

- Chaque processus reprend son exécution à l'instruction suivant le fork
- Valeurs de retour différentes selon le processus
 - **0** : renvoyé au processus fils
 - **pid (>0) du processus fils** : renvoyé au processus père
 - **-1** : appel à la primitive a échoué
 -
- `pid_t getpid(void);`
 - obtenir son pid
- `pid_t getppid (void);`
 - obtenir le pid du père

Fork : exemple

```
int main(int argc, char * argv[]) {  
    pid_t p;  
    int a = 10;  
    p = fork();  
    if(p == -1) {  
        perror("Erreur sur le fork.\n");  
        return 1;  
    }  
    if(p == 0) { /* le fils */  
        a++;  
        printf("Je suis le fils %d, pere : %d, a = %d\n",  
               getpid(),getppid(),a);  
        return 0;  
    } else { /* le pere */  
        printf("Pere de %d, a = %d\n", p,a);  
        return 0;  
    }  
}
```

Pere de 1538, a = 10

Je suis le fils 1538, pere : 1537, a = 11

Terminaison d'un processus : exit

```
void exit(int val);
```

val = valeur récupérée par le processus père

⇔ return (val) dans le main

- Le processus est détruit en mémoire et devient **Zombie**:
 - Etat d'un processus terminé tant que son père n'a pas pris connaissance de sa terminaison (wait).

Synchronisation père/fils – Wait

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

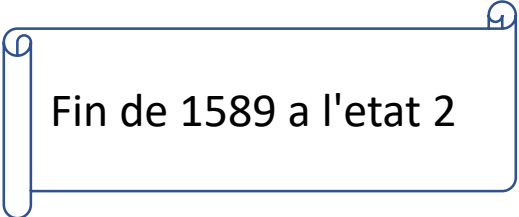
pid_t wait (int* status);
```

Attendre la fin d'un seul fils quelconque

- Argument et valeur de retour :
 - Valeur de retour : **pid** du processus fils terminé, retourne directement -1 si aucun fils.
 - Paramètre de sortie **status** passé par référence : état du processus fils (valeur d'exit).
- Macros définies dans <sys/wait.h> manipulant le paramètre **status** :
 - WEXITSTATUS(status) : valeur de retour ;
 - WIFEXITED(status) : valeur ≠ 0 si fin normale.

Wait – exemple

```
int main() {
    pid_t p, d;
    int e;
    p = fork();
    if(p == 0){ /* le fils */
        exit(2);
    } else { /* le pere */
        d = wait(&e);
        if(WIFEXITED(e)){
            printf("Fin de %d a l'etat %d \n", d, WEXITSTATUS(e));
        } else {
            printf("Probleme fils %d\n",d);
        }
    }
    return 0;
}
```



Fin de 1589 a l'etat 2

Wait – exemple zombie

```
int main() {  
    pid_t p, d;  
    int e;  
    p = fork();  
    if(p == 0){ /* le fils */  
        exit(0);  
    } else { /* le pere */  
        sleep(30);  
        wait(NULL);  
    }  
    return 0;  
}
```

Créer un processus zombie pendant environ 30 secondes

Remarque : Si le père est terminé, ses fils (orphelins) sont “adoptés” par *init* (processus1).

Waitpid : attendre un fils précis

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid (pid_t pid, int* status, int options );
```

- Attendre la fin en fonction de la valeur de *pid* :
 - > 0 du processus fils pid
 - 0 d'un processus fils quelconque du même groupe que l'appelant
 - 1 d'un processus fils quelconque
 - < -1 d'un processus fils quelconque dans le groupe |pid|
- options :
 - WNOHANG : appel non bloquant ..
 - 0 : pas d'options
- Retour
 - -1 : erreur
 - 0 : si non bloquant et processus n'a pas terminé
 - pid du processus terminé

Waitpid - Exemple

```
int main() {
    pid_t pid_fils;
    int status;
    if ((pid_fils=fork ()) == 0) {
        printf("FILS: pid = %d\n", getpid());
        sleep (1);
        exit (2);
    }
    else {
        if (waitpid(pid_fils,&status,WNOHANG) == 0)
            printf ("PERE: fils n'a pas terminé \n");
        else
            if (WIFEXITED (status) {
                printf ("PERE: fils  %d terminé, status= %d \n",
                    pid_fils, WEXITSTATUS (status));
                return EXIT_FAILURE;
            }
    }
    return EXIT_SUCCESS;
}
```

FILS : pid =19078
PERE: fils n'a pas terminé

Wait3 – récupération des statistiques du fils

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

pid_t wait3(int *status, int options, struct rusage *rusage);
```

- Attend la fin d'un fils quelconque
- *status* et *options* identiques à *waitpid*
- *struct rusage* définie dans *sys/resource* (man *getrusage*)

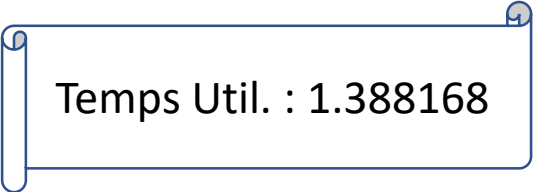
```
struct rusage {
    ...
    struct timeval ru_utime; /* Temps CPU fils mode U*/
    struct timeval ru_stime; /* Temps CPU fils mode S*/
    ...
}

└─┬──────────┘
   │
   └─┬────────┘ tv_sec
      │
      └────────┘ tv_usec
```

Wait3 - exemple

Afficher temps CPU consommé par un fils en mode U

```
...  
struct rusage r;  
if (fork() == 0) {  
    ...  
    exit(0);  
}  
wait3(NULL, 0, &r);  
printf("Temps Util. : %f\n",  
       r.ru_utime.tv_sec + 1E-6*r.ru_utime.tv_usec);
```



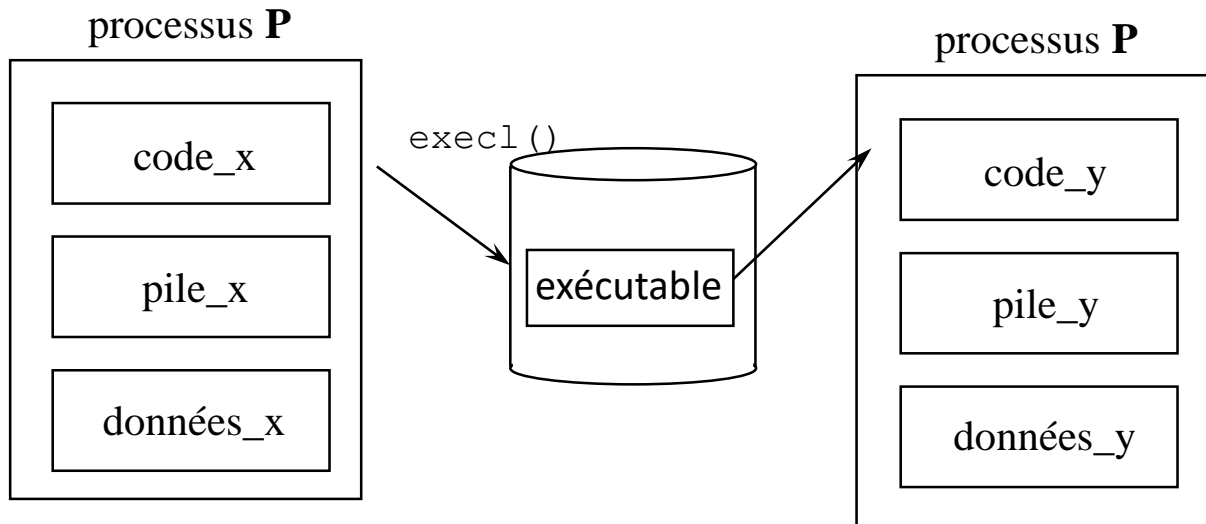
Temps Util. : 1.388168

Recouvrement - exec

- Le recouvrement permet à un processus de changer (écraser) son code et ses variables (pile et données)

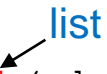
=> le processus va exécuter **un nouveau main**. Si pas d'erreur, un exec ne retourne pas


Ne créer pas de nouveau processus (écrasement du contenu)




Le processus conserve : uid, gid, pid, ppid,
terminal, fichiers ouverts, signaux masqués

Primitives exec

 `int execl(char * pathname, char * arg0, ... , NULL);`

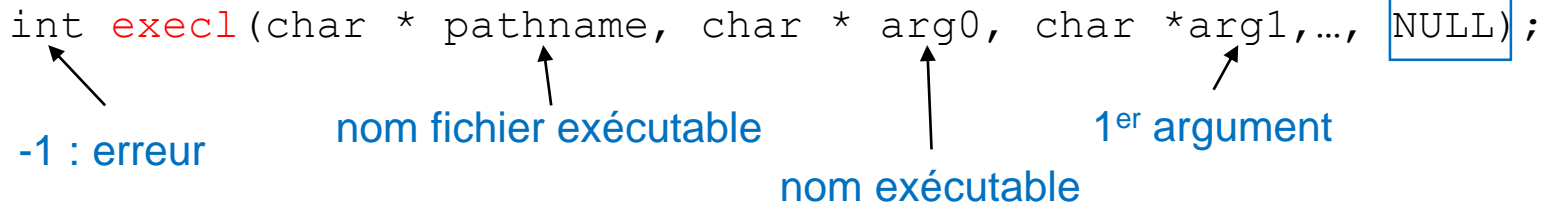
 `int execv(char * pathname, char * argv[]);`

 `int execlp(char * filename, char * arg0, ... , NULL);`

`int execvp(char * filename, char * argv[]);`

Exec - list

```
int execl(char * pathname, char * arg0, char * arg1, ..., NULL);
```



-1 : erreur nom fichier exécutable nom exécutable 1^{er} argument

exemple : créer un processus qui exécuté "ls -l f1"

```
1.  if (fork()==0) {  
2.      execl("/usr/bin/ls", "ls", "-l", "f1", NULL);  
3.      perror("Erreur exec");  
4.      exit(1);  
5.  }
```

execlp identique avec l'exécutable recherché dans *PATH*

```
2.      execlp("ls", "ls", "f1", NULL);
```

Exec - vecteur

```
int execv(char * filename, char * argv[]);
```

-1 : erreur nom fichier exécutable vecteur d'arguments

example

```
1.  if (fork()==0) {
2.      char *argfiles[] = {"ls",  "-l", "f1",  NULL};
3.      execv("/usr/bin/ls", argfiles);
4.      perror("Erreur exec");
5.      exit(1);
6.  }
```

Fonction – system()

```
#include <stdlib.h>  
int system(char *command);
```

Créer un fils qui lance un shell pour exécuter *command*

exemple

```
system("sleep 5");
```

