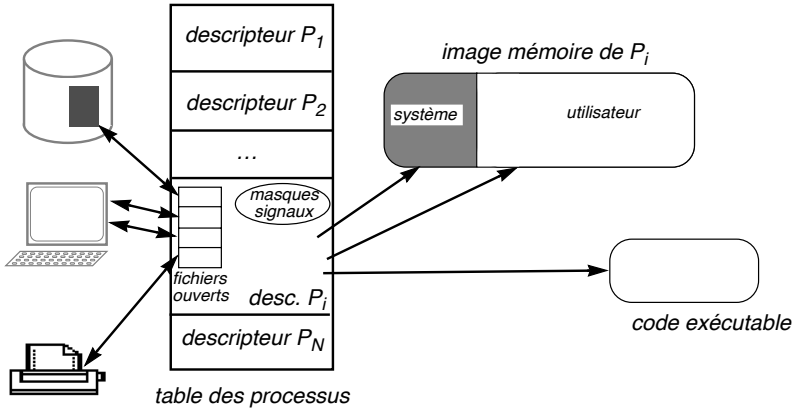
Interface programmatique de gestion des processus UNIX

Plan

- Les processus dans le système UNIX
  - ◊ Image mémoire (vue programmeur)
  - ◊ Descripteur (vue système)
- Opérations sur les processus
  - ◊ Création
  - ◊ Destruction
  - ◊ Identification
  - ◊ Synchronisation
    - Attente d'un laps de temps
    - Envoi/attente d'événements
    - Coordination père/fils

Descripteur de processus (vue système)



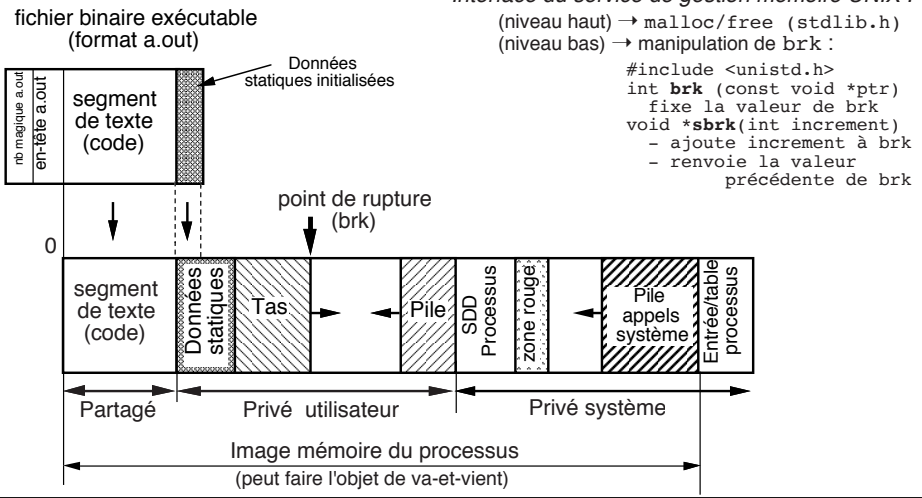
1 – Les processus dans le système UNIX

Image mémoire (vue programmeur)

Interface du service de gestion mémoire UNIX :

(niveau haut) → `malloc/free (stdlib.h)`
(niveau bas) → manipulation de `brk` :

```
#include <unistd.h>
int brk (const void *ptr)
  fixe la valeur de brk
void *sbrk(int increment)
  - ajoute increment à brk
  - renvoie la valeur
        précédente de brk
```



2 – Opérations sur les processus

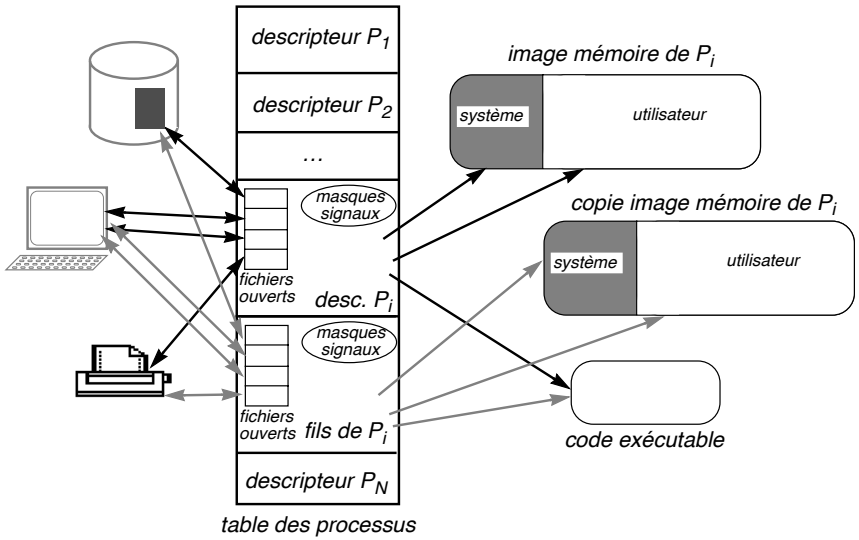
Création

`pid_t fork();`
crée un (descripteur de) processus « clone » du processus appelant (processus *père*).

- Le processus créé (*fils*) hérite
  - ◊ du même code,
  - ◊ d'une copie de la zone de données du père, amélioration/optimisation : copie limitée aux pages modifiées après **`fork()`** (c.o.w))
  - ◊ de l'environnement,
  - ◊ de la priorité,
  - ◊ des descripteurs de fichiers ouverts,
  - ◊ du traitement des signaux.
- La valeur de retour de `fork()` est cependant différente :
  - ◊ `0` pour le fils
  - ◊ l'identifiant (pid) du fils pour le père

Intérêt

le père peut définir et transmettre simplement et précisément le contexte d'exécution du fils



L'ordonnancement des processus père et fils est indéterminé

Exemple

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main(void) {
    int pid = fork(); /* père et fils */
    printf("Valeur de fork = %d ",pid);
    printf("processus %d de pere %d\n",getpid(),getppid());
    if ( pid == 0 ) { /* processus fils */ printf("fin du fils\n"); }
    else { /* processus pere */ printf("fin du processus pere\n"); }
    return 0;
}
```

Résultats

```
prompt%creer_processus
Valeur de fork = 0 processus 434 de pere 433
fin du fils
Valeur de fork = 434 processus 433 de pere 364
fin du processus pere
prompt%

prompt%creer_processus
Valeur de fork = 0 processus 397 de pere 396
Valeur de fork = 397 processus 396 de pere 364
fin du processus pere
prompt%fin du fils

prompt%creer_processus
Valeur de fork = 440 processus 439 de pere 364
fin du processus pere
prompt%Valeur de fork = 0 processus 440 de pere 1
fin du fils
```


Destruction

```
void exit(int n);
```

termine le processus courant avec le code de retour `n` (en général, `0` si retour normal)

Chargement d'un code à exécuter : recouvrement

```
int execl(char *chemin, char *arg0, char *arg1, char *arg2,…, char *argn, NULL);
int execlp(char *chemin, char *arg0, char *arg1, char *arg2,…, char *argn, NULL);
int execle(char *chemin, char *arg0, char *arg1, …, char *argn, NULL, char *env[]);
int execv(char *chemin, char *argv[]);
int execvp(char *chemin, char *argv[]);
int execve(char *chemin, char *argv[],char *env[]);
```

- décodage
  - ◊ l/v : liste/tableau
  - ◊ p : utilisation de PATH
  - ◊ e : passage de l'environnement
- après recouvrement
  - ◊ une nouvelle image mémoire est allouée
  - ◊ les signaux non ignorés sont associés à leur traitant par défaut
  - ◊ les descripteurs restent ouverts, sauf ceux indiqués par fcntl (FD_CLOEXEC)
  - ◊ les autres attributs sont conservés

Identification

- **`pid_t getpid()`** : pid de l'appelant
- **`pid_t getppid()`** : pid du père de l'appelant
- **`uid_t getuid()`** : id de l'utilisateur ayant lancé le processus
- **`uid_t geteuid()`** : id de l'utilisateur effectif
- **`uid_t getgid()`** : id du groupe de l'utilisateur ayant lancé le processus
- **`uid_t getegid()`** : id du groupe effectif
- **`int setuid(uid_t u)`**
- **`int setgid(gid_t g)`**

Synchronisation

Attente d'un laps de temps

`int sleep(int n)`

suspend l'exécution du processus appelant pour n secondes (au moins)

Envoi/attente d'événement

`kill/pause` : vu plus loin

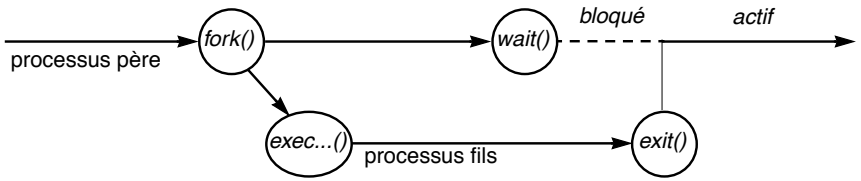
Coordination père/fils

`#include <sys/wait.h>`

`pid_t wait(int *n);`

suspend l'appelant jusqu'à la terminaison d'un fils, et retourne le pid du fils terminé

- si le fils s'est terminé par `exit()`, le deuxième octet de `*n` est le code renvoyé par `exit()`
- si le fils s'est terminé suite à la réception d'un signal, le premier octet de `*n` est le numéro du signal (+128 si un fichier `core` a été engendré)
- des macros (`WIFSIGNALED, WIFEXITED, WTERMSIG, WEXITSTATUS`…) permettent de manipuler plus simplement `*n`
- comme la plupart des appels système, si le père reçoit un signal alors qu'il attend sur `wait`
  - ◊ si un traitant a été défini, le signal est traité et le père reste en attente
  - ◊ sinon, `wait` se termine et renvoie -1
- un appel à `wait` alors qu'aucun fils (zombie ou actif) n'existe, se termine aussitôt et renvoie -1
- **`int waitpid(pid_t p)`** permet d'attendre la fin d'un processus fils donné

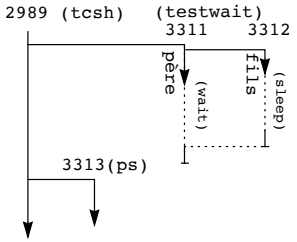


Exemple : Programme testwait.c

```
#include <sys/types.h>
#include <sys/wait.h>
int main() {
  if (fork() != 0) { /* père */
    int statut; pid_t fils);
    printf("je suis le père %d, j'attends mon fils\n", getpid());
    fils = wait(&statut);
    if (WIFEXITED(statut)) {
        printf("%d : mon fils %d s'est terminé avec le code %d\n",
           getpid(), fils, WEXITSTATUS(statut); }
     exit(0);
   } else { /* fils */
    printf("je suis le fils, mon PID est %d\n", getpid());
    sleep(2)  /* blocage pendant 2 secondes */
    printf("fin du fils\n"); exit(1);
  }}
```

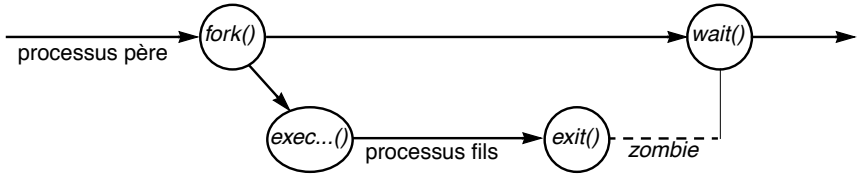
Exécution

```
<mozart> ./testwait
je suis le fils, mon PID est 3312
je suis le père 3311, j'attends mon fils
fin du fils
3311: mon fils 3312 s'est terminé avec le code 1
<mozart> ps
 PID TTY          TIME CMD
2989 pts/0    00:00:00 tcsh
3313 pts/0    00:00:00 ps
<mozart>
```



Processus zombis

Tant que son père n'a pas pris connaissance de sa terminaison par `wait` ou `waitpid`, un processus terminé reste dans un état dit zombi.



- *Intérêt* : signification du wait : attente de la terminaison d'un fils, indépendamment de la durée d'exécution de ce fils
- *Inconvénient* : un processus zombi ne peut plus s'exécuter, mais consomme encore des ressources (tables)

  → éviter de conserver des processus dans cet état.

  → un traitement standard est d'exécuter **`wait`** dans un traitant associé au signal `SIGCHLD`

Processus orphelins

- lorsqu'un processus se termine, ses fils (dits orphelins) sont rattachés au processus 1 (`init`)
- le processus `init` élimine les zombies en appelant systématiquement `wait`

Observation des zombies : exemple

Programme testzombie.c

```
int main() {
  if (fork() != 0) {
    printf("je suis le père, mon PID est %d\n", getpid());
    while (1) ; /* boucle sans fin sans attendre le fils */
  } else {
    printf("je suis le fils, mon PID est %d\n", getpid());
    sleep(2)  /* blocage pendant 2 secondes */
    printf("fin du fils\n");
    exit(0);
  }
}
```

Exécution

```
<mozart> gcc -o testzombie testzombie.c
<mozart> ./testzombie
je suis le fils, mon PID est 3271
je suis le père, mon PID est 3270
fin du fils
==>frappe de <control-Z> (suspendre)
Suspended
<mozart> ps
 PID TTY          TIME CMD
2989 pts/0    00:00:00 tcsh
3270 pts/0    00:00:03 testzombie
3271 pts/0    00:00:00 testzombie <defunct>
3272 pts/0    00:00:00 ps
<mozart>
```