Processus

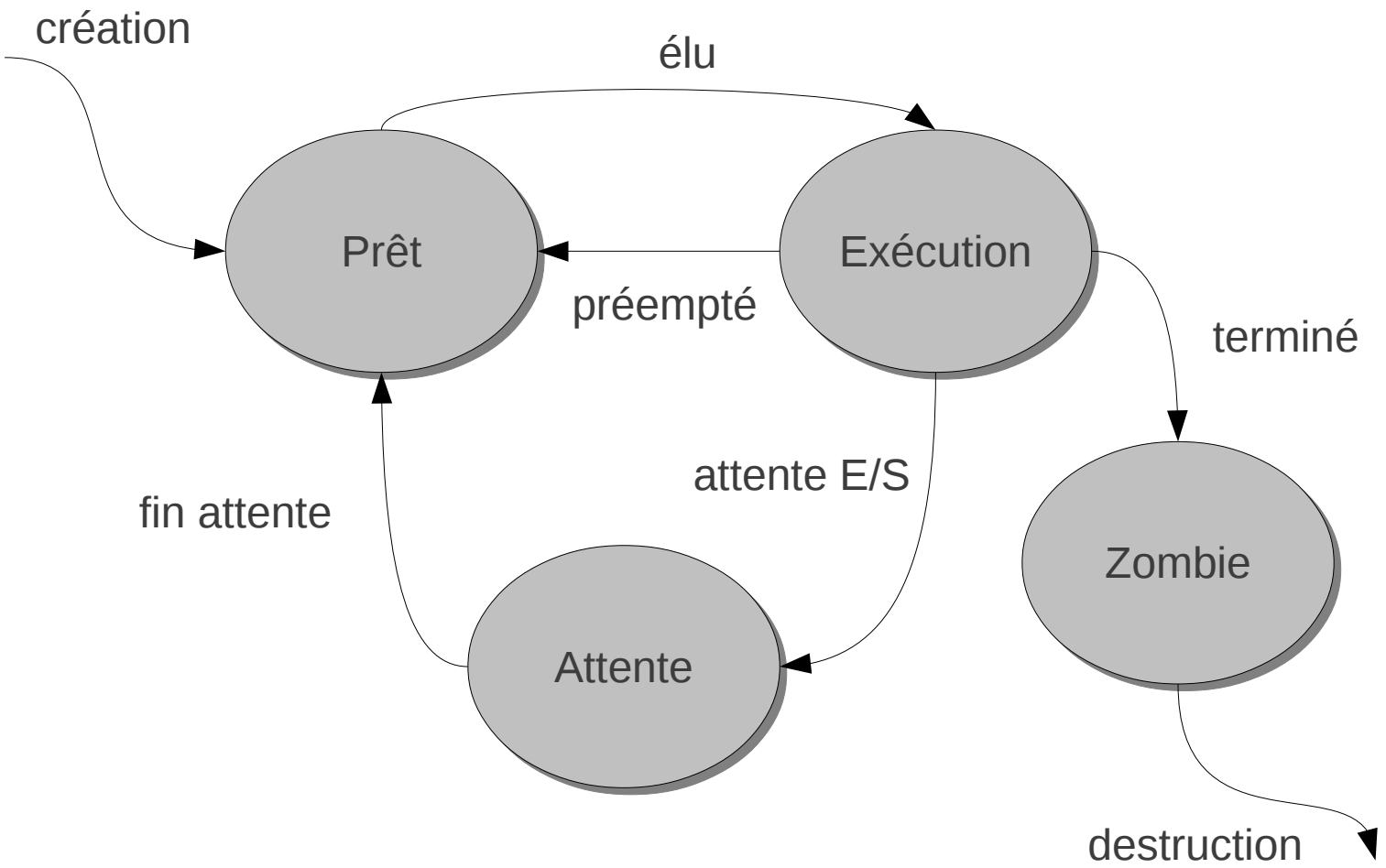
processus ≠ programme

- Programme = fichier exécutable sur disque contenant :
  - en-tête
  - code binaire (instructions machine)
  - données statiques (i.e. variables initialisées)

- Processus = programme en cours d'exécution

  = entrée dans la table des processus + en RAM : instructions machine + pile + tas + variables globales

États processus



Attributs d'un processus

- Chaque processus possède un ensemble d'attributs :
  - numéro (PID : Process Identifier)
  - numéro du père (PPID)
  - chemin de l’exécutable
  - infos utilisation processeur
  - infos ordonnancement (priorité, …)
  - fichiers ouverts
  - localisation mémoire (code, pile, ...)
  - propriétaire (uid, gid)
  - code retour
  - ...

Table des processus

- Le noyau maintient une table de structures *task_struct*, une entrée = structure contenant tous les attributs du processus
- Pour visualiser les informations de la table :
  - `ps (options eaux)`
  - `pstree`
  - `top/htop`
  - `/proc/<pid>/`

    `(status, fd, fdinfo, ...`

    `exemple : watch -n1 tail status )`

Arborescence de processus

- Chaque processus possède un père (de numéro PPID) = processus créateur

→ arbre de processus

racine de l'arbre = le processus init (pid : 1)

- Chaque processus rend un code retour

(`echo $?` sous bash)

Appels système processus

- Comment lancer un processus depuis un programme ?

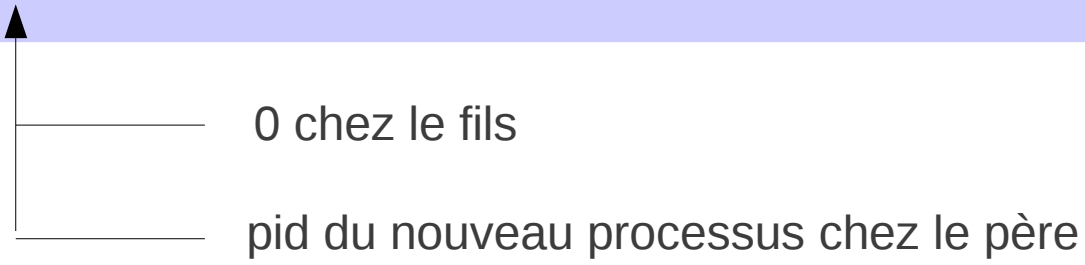
```
main() {
   printf("Exécution de la commande ls -l\n");
   system("ls -l");
   printf("Fin de la commande\n");
}
```

- Mais la fonction `system` n'est pas directement l'appel système

Appels système processus

- 4 appels système : fork, exec, wait, exit

```
pid_t  fork(void); /** >duplication du processus courant */
```




```
int main() {
   int f;
   f = fork();
   printf( "pid : %d\n", f );
   return( 0 );
}
```

```
Prompt > ./main
pid : 0
pid : 25992
Prompt >
```

Fork

Lors d'un fork :

- une entrée disponible de la table des processus est attribuée au nouveau processus
- les 2 processus partagent le même code
- le processus fils travaille sur une copie des données du père, toute modification dans un processus n'est pas visible de l'autre
- le père transmet les descripteurs de fichiers à son fils, les descripteurs du père et du fils pointent sur les mêmes entrées dans la table des fichiers ouverts (→ même position dans le fichier).

Exit : fin d'un processus

```
void exit(int code_de_retour);
```

- chaque processus renvoie un code de retour soit par exit, soit implicitement (valeur de retour de la fonction main)
- dans le bash : echo $?
- 0 : ok, sinon échec

Wait

```
pid_t wait(int *status);
```

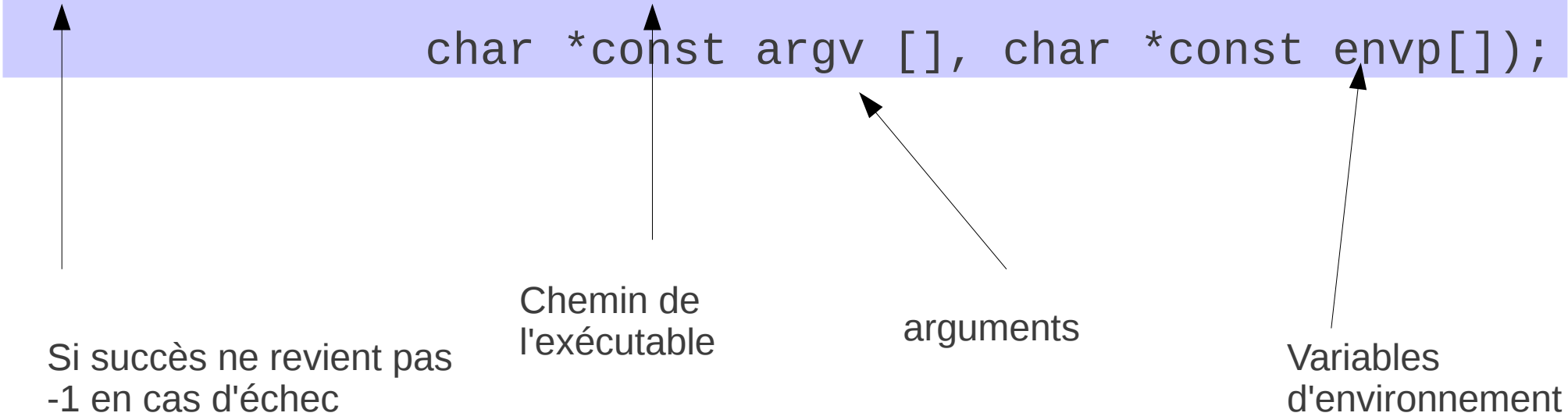

Pid du fils mort

Contient le code de retour du fils, NULL pour l'ignorer

- attend la terminaison d'un processus fils
- `status` : doit être exploité avec les macros fournies (voir la page man)
- voir aussi **`waitpid`**`()`

Execve : recouvrement de proc

```
int execve(const char *filename,
                char *const argv [], char *const envp[]);
```



Execve recouvre le code, la pile, le tas et ne conserve que les descripteurs

Frontaux pour exec

Plusieurs frontaux pour execve :

exec +

l (liste) ou **v** (vecteur) : pour les arguments

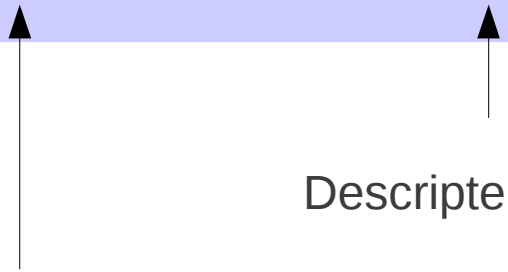
p (path) : utiliser PATH ou non

e (environnement) : précision des variables d'environnement

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

Dup : redirection de fichiers

```
int dup(int desc);
```



Descripteur à dupliquer

Nouveau desc (le premier libre dans la table), ancien et nouveau desc partagent position, flags
-1 si erreur

```
int dup2(int ancien_desc, int nouv_desc);
```

Idem mais force la valeur du nouveau descripteur

Tubes nommés

un premier mécanisme d'échange entre proc

- Fichier géré en mode FIFO (First In First Out) = file => pas de lseek, lecture destructrice
- Fichier apparaissant dans le FS (type 'p' pour pipe)
- Création :
  - commande mkfifo
  - fonction int mkfifo (const char *pathname, mode_t mode);
- Se manipule comme un fichier : `read, write, close`

Tubes non nommés

- Permet la communication entre processus parents
- Fichier « interne » (uniquement en RAM) n'apparaissant pas dans le FS (visible avec lsof)
- Création : primitive `pipe`
- Accès : `read, write`

Primitive pipe

```
int pipe(int fd[2]);
```

0 = succès
-1= échec

Pointeur sur un tableau de deux descripteurs de fichiers (int)

- Crée un tube non nommé
- Écriture dans fd[1], lecture dans fd[0]

Comportement d'un tube en lecture/écriture

- Par défaut, la lecture dans un tube est bloquante s'il n'y a pas de caractères à lire
- La lecture n'est plus blocante (et rend 0) s'il n'y a plus aucun écrivain

- une écriture dans un tube n'ayant plus de lecteur =>

  interruption du processus (signal SIGPIPE), le shell de lancement affiche "broken pipe"
- **Règle** : pour éviter les erreurs d'étourderie provoquant des interblocages : ne conserver que les descripteurs utiles, fermer les autres