

Module SY5 – Systèmes d'Exploitation

Dominique Poulalhon

`dominique.poulalhon@irif.fr`

Université Paris Cité

L3 Informatique & DL Bio-Info, Jap-Info, Math-Info

Année universitaire 2023-2024

PROCESSUS (SUITE)

CRÉATION DE PROCESSUS

sous UNIX, la création de processus est scindée en deux étapes :

- le **clonage** = création d'un processus (presque) identique : même état de la mémoire (code, pile, tas), même compteur ordinal, même pointeur de pile, mêmes fichiers ouverts... \Rightarrow `fork()`

(le nouveau processus dispose de son propre espace d'adressage, indépendant de celui de son père, et naturellement de son propre bloc de contrôle)

- le **recouvrement** = remplacement de toute la mémoire par un nouveau segment de code, réinitialisation de la pile et du tas, réinitialisation des registres \Rightarrow (famille) `exec*()`

hiérarchie de processus : un processus et ses descendants forment un **groupe** de processus, auquel on peut envoyer collectivement un signal ; par ailleurs le père est d'une certaine manière « responsable » de ses fils

CRÉATION DE PROCESSUS

```
pid_t fork(void);
```

- retourne -1 en cas d'erreur (et `errno` est positionnée)

sinon :

- crée un nouveau processus (*fil*s) par clonage du processus courant (*père*)
- retourne 0 dans le processus fils
- retourne le pid du fils dans le processus père

autrement dit, *un appel* à cette fonction entraîne *deux retours*

le nouveau processus ne diffère de l'ancien essentiellement que par son identifiant (et celui de son père)

le fils poursuit l'exécution au point où en était son père

CRÉATION DE PROCESSUS

```
pid_t fork(void);
```

Comment différencier le père du fils ? par la valeur de retour de `fork`

```
switch(r = fork()) {  
    case -1:  
        perror("fork");  
        exit(1);  
    case 0: /* code pour le fils */  
        break;  
    default: /* code pour le père */  
}
```

CRÉATION DE PROCESSUS

```
pid_t fork(void);
```

l'*espace d'adressage* du fils est (initialement) une copie de celui du père

la *table des descripteurs* du fils est (initialement) une copie de celle du père

chaque descripteur du fils pointe sur la *même entrée* de la table des ouvertures de fichiers que le descripteur correspondant du père

ils partagent donc la *même position courante* dans le fichier ouvert

ZOMBIES ET SYNCHRONISATION PÈRE-FILS

lorsqu'il a terminé son exécution (appel à `_exit` ou signal), un processus libère toutes ses ressources, sauf son entrée dans la table des processus : c'est l'état « zombie »

cette ligne n'est libérée que lorsque le père du processus prend connaissance de sa terminaison

pour traiter le cas des orphelins, mécanisme d'adoption, traditionnellement par le processus n° 1 exécutant `init`, ou un processus exécutant `systemd` (un par utilisateur)

ZOMBIES ET SYNCHRONISATION PÈRE-FILS

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

- retourne -1 avec `errno=ECHILD` si le processus appelant n'a pas de fils ;
- retourne le pid d'un fils zombie, si le processus appelant en a au moins un ; le fils zombie est alors détruit ;
- bloque en attendant la mort d'un fils si aucun des fils du processus appelant n'est un zombie.

si `wstatus` n'est pas `NULL`, y stocke les informations concernant la mort du fils ; consultable via des macros : `WIFEXITED(status)`, `WEXITSTATUS(wstatus)`...

ZOMBIES ET SYNCHRONISATION PÈRE-FILS

`wait` a donc un triple rôle :

- permettre au père de récupérer des informations de son fils,
- permettre de libérer les dernières ressources utilisées par le fils défunt,
- permettre la [synchronisation](#) du père sur (la terminaison de) son fils

ZOMBIES ET SYNCHRONISATION PÈRE-FILS

`wait` a donc un triple rôle :

- permettre au père de récupérer des informations de son fils,
- permettre de libérer les dernières ressources utilisées par le fils défunt,
- permettre la **synchronisation** du père sur (la terminaison de) son fils

attention, `wait` ne permet pas la synchronisation d'un processus sur un processus quelconque, ni sur son père, ni sur un petit-fils

ZOMBIES ET SYNCHRONISATION PÈRE-FILS

`wait` a donc un triple rôle :

- permettre au père de récupérer des informations de son fils,
- permettre de libérer les dernières ressources utilisées par le fils défunt,
- permettre la **synchronisation** du père sur (la terminaison de) son fils

attention, `wait` ne permet pas la synchronisation d'un processus sur un processus quelconque, ni sur son père, ni sur un petit-fils

le **double fork** : stratégie pour ne pas avoir à attendre le fils, lorsqu'aucune synchronisation n'est nécessaire :

- créer un fils puis un petit-fils par 2 appels consécutifs à `fork`
- tuer le fils : le petit-fils est alors adopté par le processus 1

utile en particulier pour créer des **démons**

ZOMBIES ET SYNCHRONISATION PÈRE-FILS

il existe une variante étendue de `wait` :

```
pid_t waitpid(pid_t pid, int *wstatus, int options);
```

- `pid` précise quel fils attendre :
 - `pid > 0` pour un fils précis,
 - `-1` pour n'importe quel fils,
 - `0` pour n'importe quel fils appartenant au groupe du père,
 - `pid < -1` pour un fils appartenant au groupe `-pid`
- les `options` permettent de moduler le comportement :
 - `WNOHANG` pour ne pas bloquer en l'absence de fils zombie,
 - `WUNTRACED` pour attendre également les fils suspendus.

RECOUVREMENT

les fonctions suivantes permettent de *recouvrir* le processus, c'est-à-dire de changer le programme qu'il doit exécuter

```
int execl(const char *path, const char *arg, ... /* (char *) NULL */);
int execlp(const char *file, const char *arg, ... /* (char *) NULL */);
int execle(const char *path, const char *arg, ...
           /*, (char *) NULL, char *const envp[] */);
int execlv(const char *path, char *const argv[]);
int execlvp(const char *file, char *const argv[]);
int execlve(const char *path, char *const argv[], char *const envp[]);
```

- le premier argument désigne l'exécutable à charger
 - les suivants représentent ses arguments (tableau `argv`)
 - éventuellement suivis d'un nouvel `environnement`
-
- retournent -1 en cas d'erreur (et `errno` est positionnée)
 - sinon, il n'y a *pas de retour* de ces fonctions : le processus exécute la fonction `main` du nouveau programme avec les arguments `argv`

RECOUVREMENT

ces fonctions diffèrent dans la manière dont les paramètres leur sont fournis :

variantes « l » : les arguments sont donnés sous forme de liste, terminée par `NULL`

variantes « v » : les arguments sont donnés sous forme de tableau (avec `NULL` dans la dernière case)

variantes « p » : la recherche de l'exécutable tient compte de la variable `PATH` et peut donc se baser sur le seul `basename` du fichier ; à l'inverse, il faut nécessairement fournir une référence valide explicite aux variantes « non-p »

variantes « e » : le dernier paramètre permet de spécifier un nouvel environnement

ENVIRONNEMENT D'UN PROCESSUS

liste de chaînes de caractères de la forme `VARIABLE= valeur`, selon laquelle le processus module son comportement : `HOME`, `PATH`, `USER`, `LANG`...

par défaut, l'environnement du fils est celui hérité du père

un programme peut y accéder de plusieurs manières :

- la variable `extern char **environ`
- les fonctions `getenv()`, `setenv()`, `unsetenv()`...
- la forme étendue du `main` (non POSIX) :
`int main (int argc, char *argv[], char *envp[])`

POURQUOI SÉPARER CLONAGE ET RECouvreMENT ?

principalement car cela laisse une opportunité pour modifier certaines choses – impérativement parmi celles qui ne seront pas écrasées par le recouvrement

POURQUOI SÉPARER CLONAGE ET RECouvreMENT ?

principalement car cela laisse une opportunité pour modifier certaines choses – impérativement parmi celles qui ne seront pas écrasées par le recouvrement

cela concerne principalement :

- la table des descripteurs : changer les fichiers (avec `dup()` et `dup2()`) associés aux descripteurs « standard » (`STDIN_FILENO`, `STDOUT_FILENO`, `STDERR_FILENO` permet de réaliser des redirections
- la définition des comportements associés à la réception de signaux (cf. cours ultérieur)

DUPLICATION DE DESCRIPTEURS

dupliquer un descripteur, c'est associer à un nouveau descripteur la même ouverture de fichier (*i.e.*, la même entrée de la table des ouvertures) – exactement comme ce qui se passe lors d'un **fork** : tous les descripteurs du processus père sont dupliqués pour définir les descripteurs du processus fils (*et le compteur de descripteur associé à l'ouverture est incrémenté*)

```
int dup(int oldfd);  
int dup2(int oldfd, int newfd);
```

- duplique le descripteur **oldfd**,
- retourne la valeur du nouveau descripteur

la valeur du nouveau descripteur est :

- le plus petit entier disponible, dans le cas de **dup**,
- **newfd**, dans le cas de **dup2**; le cas échéant, **newfd** est d'abord fermé (opération atomique)