

# 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 2022-2023

# PROCESSUS

## DÉFINITION

`processus` = objet dynamique correspondant à une exécution d'un programme

## DÉFINITION

**processus** = objet dynamique correspondant à une exécution d'un programme

au cours du temps, un processus passe en boucle par les **états** suivants :

- état **prêt**
- états **actifs** (actif noyau ou actif utilisateur)
- état **endormi** ou **en attente**

## DÉFINITION

**processus** = objet dynamique correspondant à une exécution d'un programme

au cours du temps, un processus passe en boucle par les **états** suivants :

- état **prêt**
- états **actifs** (actif noyau ou actif utilisateur)
- état **endormi** ou **en attente**

autres états possibles :

- état transitoire initial
- état **suspendu**
- état **zombie**

## IMPLÉMENTATION

un processus a deux constituants :

- son **espace d'adressage** : la zone mémoire où il travaille, divisée en segment de texte (le code à exécuter) et segment de données – statiques et dynamiques (pile et tas)

## IMPLÉMENTATION

un processus a deux constituants :

- son **espace d'adressage** : la zone mémoire où il travaille, divisée en segment de texte (le code à exécuter) et segment de données – statiques et dynamiques (pile et tas)
- son **bloc de contrôle** : toutes les informations dont le système a besoin pour assurer la bonne gestion des processus : entre autres, identifiant, état, compteur ordinal, pointeur de pile, répertoire de travail, descripteurs...  
les informations utiles même lorsque le processus est inactif sont stockées dans la **table des processus** ; les autres (descripteurs par exemple) peuvent être stockées dans l'espace d'adressage.

## QUELQUES ATTRIBUTS DES PROCESSUS

son identifiant, celui de son père

```
pid_t getpid(void);  
pid_t getppid(void);
```

ses propriétaires (réel et effectif)

```
uid_t getuid(void);  
uid_t geteuid(void);  
int setuid(uid_t uid);  
int seteuid(uid_t euid);
```

son répertoire de travail courant

```
char *getcwd(char *buf, size_t size);  
int chdir(const char *path);  
int fchdir(int fd);
```



## CRÉATION DE PROCESSUS

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

## 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...  $\implies$  `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)*

## 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*()`

## 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

## 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 (*p*ère)
- 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



## 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

## 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

## 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