Séance 2 – Processus et ordonnancement

Damien MASSON
http://esiee.dajam.fr

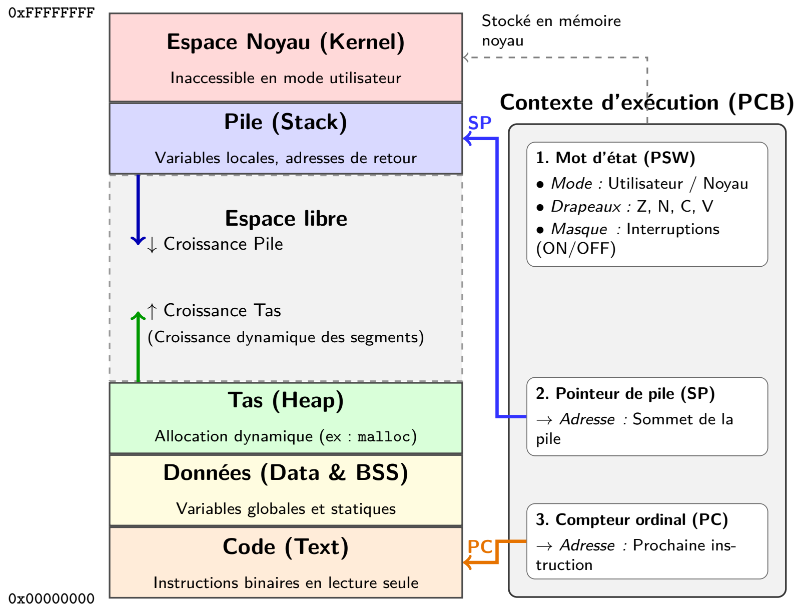
Dernière modification : 9 avril 2026

Qu'est-ce qu'un processus ?

Un processus (*process*) représente un programme en cours d'exécution.

- **Code :** correspond aux instructions (assembleur) du programme à exécuter.
- **Zone de données :** contient les variables globales ou statiques, ainsi que les allocations dynamiques (tas / *heap*).
- **Pile (*stack*) :** sert à empiler les appels de fonctions, avec leurs paramètres et leurs variables locales.
- **Compteur ordinal (*Program Counter*) :** pointe vers la prochaine instruction à exécuter.
- **Pointeur de pile et données :** adresses de la pile et du tas.
- ...

Espace d'adressage virtuel et Contexte



Quel rapport avec le SE ?

Le système d'exploitation est responsable de :

- La création et la destruction des processus.
- L'arrêt et la reprise des processus (ordonnancement).
- La fourniture de mécanismes pour la synchronisation et la communication inter-processus (IPC).

Concept le plus important des SE modernes :

Multi-programmation / Multitâche

Aujourd'hui

- Comment le SE donne-t-il l'impression à l'utilisateur que plusieurs processus s'exécutent en parallèle ?
- Comment réaliser un programme utilisant plusieurs processus ?
  - Déclenchement d'un nouveau processus.
  - Arrêt d'un processus.
  - Synchronisation de plusieurs processus.

Plan de la séance

1. Gestion de la multi-programmation par le SE
   - Multi-programmation
   - Table des processus
   - Changement de contexte
   - Ordonnanceur
   - Visualisations de processus
2. Utiliser la multi-programmation en C (sous Linux)
   - Création de processus
   - Élimination de processus
   - Synchronisation de processus
   - Recouvrement de processus
   - Quelques mots sur Windows

1. Gestion de la multi-programmation par le SE
   - Multi-programmation
   - Table des processus
   - Changement de contexte
   - Ordonnanceur
   - Visualisations de processus

2. Utiliser la multi-programmation en C (sous Linux)
   - Création de processus
   - Élimination de processus
   - Synchronisation de processus
   - Recouvrement de processus
   - Quelques mots sur Windows

Multi-programmation (Multitâche)

Objectif :

Sur un intervalle de temps assez grand, faire progresser tous les processus.

Contrainte :

À un instant donné (sur un cœur de processeur), un seul processus est actif.

- Un processus est une succession de phases de calcul et d'Entrées/Sorties (E/S).
- Il faut optimiser la progression du ou des processus.
- L'objectif est que le processeur soit le moins inactif possible.
- **Idée de base :** recouvrir une phase d'E/S d'un processus avec des phases de calcul d'autres processus.

Optimiser la progression des processus

Le système cherche à optimiser plusieurs grandeurs :

- **Taux d'utilisation de l'Unité Centrale (UC) :** rapport entre la durée où l'UC est active et la durée totale (en pratique, entre 40 et 95%).
- **Débit :** nombre de programmes utilisateurs traités en moyenne dans le temps.
- **Temps de traitement moyen :** moyenne des intervalles de temps séparant la soumission d'une tâche de sa fin d'exécution.
- **Temps de traitement total** d'un ensemble de processus donné.
- **Temps de réponse maximum :** maximum des durées séparant la soumission d'une requête de son accomplissement.

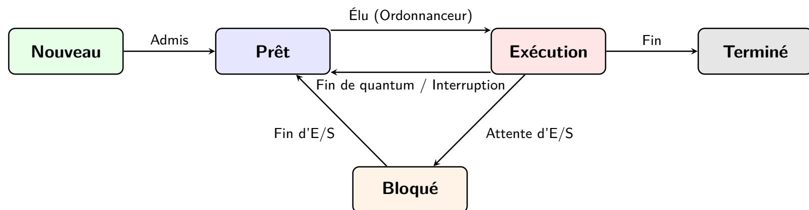
Multi-programmation : mécanismes

Pour la multi-programmation préemptive, le système nécessite :

- La connaissance/description de tous les processus :
  - **Bloc de contrôle des processus (PCB)**.
- Un moyen de changer le processus actif :
  - **Commutation de contexte** (mots d'état).
- Un algorithme qui détermine l'ordre d'exécution des processus :
  - **Ordonnanceur (*Scheduler*)**.

Processus : approche et états

- Essentiellement, c'est un programme en cours d'exécution.
- À tout moment, il se trouve dans un **état** précis.
- Par exemple, il peut être en cours d'*exécution*, *prêt*, ou *bloqué* (en attente).



Processus et noyau

- Pour commuter d'un état à l'autre, le noyau maintient une table des processus contenant les **PCB (Process Control Block)**.
- Le nombre d'emplacements dans cette table est limité pour chaque système et chaque utilisateur (ex : `ulimit` sous Linux).
- **En bref :** Une entrée de la table PCB contient tout ce qui doit être sauvegardé lorsque l'exécution est suspendue, et qui sera nécessaire pour sa reprise exacte.

Process Control Block (PCB)

Structure de données du noyau représentant l'état d'un processus donné. Elle contient en général :

- L'ID du processus (`PID`), l'ID du processus parent (`PPID`) et l'ID de l'utilisateur (`UID`).
- Les valeurs des registres matériels (l'état courant du processus).
- Le compteur ordinal du processus.
- L'espace d'adressage en mémoire.
- La liste des descripteurs de fichiers ouverts.
- La table de gestion des signaux.
- D'autres informations statistiques (temps CPU accumulé, priorité, etc.).

Structure de la table des PCB dans le noyau

Table des PCB (noyau)

Index	PID	Pointeur vers PCB
⋮		
i	101	→ 0xABCD0100
j	203	→ 0xBCDE0200
k	155	→ 0xCDEF0300
⋮		

Pointeur 1

Process Control Block (PID 101)

A. Identificateurs

• PID : **101** | Parent (PPID) : **1** (init) | Utilisateur (UID) : **1000** (bob)

B. Contexte CPU

→ **PSW** : 0x00000000 → **PC** : 0x08048C00 (Code) → **SP** : 0xBFFFCF00 (Sommet Pile)

C. Mémoire & Ressources

• Tables des Pages, Limites segment... | Fichiers ouverts (FDs : 0, 1, 2, 3...) | Signaux (Masque)

D. État

• État : **EN ATTENTE** (Blocked) | Priorité : 20

Pointeur 2

Process Control Block (PID 203)

A. Identificateurs

• PID : **203** | Parent (PPID) : **101** | Utilisateur (UID) : **1000** (bob)

B. Contexte CPU

→ **PSW** : 0x00000000 → **PC** : 0x08049100 (Code) → **SP** : 0xBFFFD000 (Sommet Pile)

C. État

• État : **PRÊT** (Runnable) | Priorité : 22

Pointeur 3

Process Control Block (PID 155)

A. Identificateurs

• PID : **155** | Parent (PPID) : **1** | Utilisateur (UID) : **0** (root)

B. Contexte CPU (exécuté)

→ **PSW** : 0x00000000 (Chargé) → **PC** : 0x08048A00 (Chargé) → **SP** : 0xBFFFC900 (Chargé)

C. État

• État : **ÉLU** (Running) | Priorité : 10

Contexte matériel et Mot d'état (PSW)

- Dans l'unité centrale, l'exécution d'un processus est caractérisée par un ensemble de registres (**le contexte CPU**) :
  - Son **compteur ordinal** (PC - Program Counter).
  - Son **pointeur de pile** (SP - Stack Pointer).
  - Ses registres d'adressage (ex : adresse de base) et de données.
  - Son **mot d'état** (PSW - Processor/Program Status Word).
- Le **PSW** est donc un registre spécifique du contexte CPU. Il contient l'état matériel instantané (drapeaux arithmétiques, mode noyau/utilisateur, masque d'interruptions).
- Un processus est donc entièrement caractérisé par :
  - Le programme sous-jacent et ses données (**contexte en mémoire**).
  - Les valeurs de tous ses registres au moment de l'interruption (**contexte unité centrale**).

Le changement de contexte (Context Switch)

Étapes d'un changement de contexte complet entre deux processus :

1. Sauvegarde de l'intégralité du **contexte CPU** courant (PSW, PC, SP et registres généraux) dans la zone mémoire du processus (le PCB).
2. Chargement du **nouveau contexte CPU**, à partir du PCB du prochain processus élu.

Opération critique et protection

Le changement de contexte est une opération logicielle coûteuse gérée par le noyau. L'interruption qui le déclenche repose, elle, sur des mécanismes matériels **atomiques**.

Définition : Opération atomique

Une opération atomique est un ensemble d'instructions matérielles indivisibles. On a la garantie qu'aucune interruption ne se produira pendant son exécution.

L'interruption

- C'est un signal généré par le matériel (ou le logiciel) qui force le CPU à suspendre l'exécution courante.
- **Le mécanisme matériel (atomique) :** le processeur sauvegarde de lui-même le strict minimum (PC et PSW courants) et bascule en mode noyau.
- À chaque catégorie d'événement, on associe un **vecteur d'interruption** :
  - Il contient l'adresse (nouveau PC) de la routine de traitement et le nouveau mot d'état (PSW) cible.
- Après ce basculement matériel initial, c'est la routine d'interruption qui s'exécute (code logiciel du noyau).

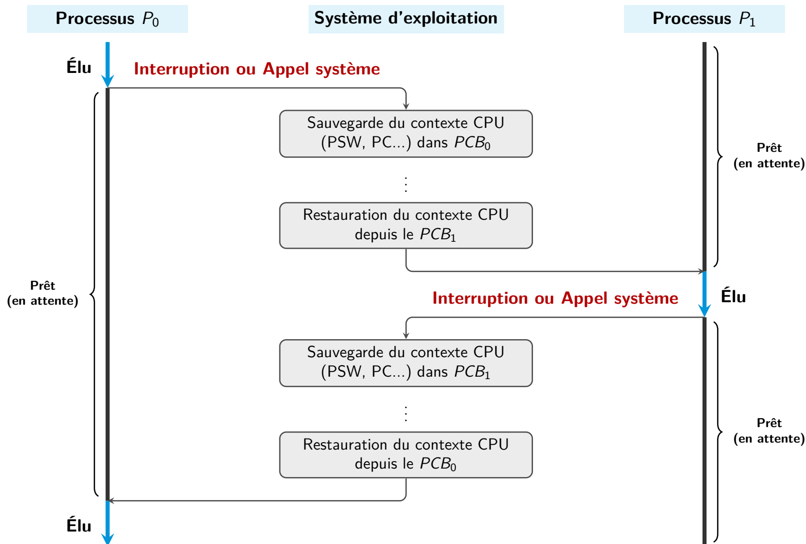
Exemple : Niveaux d'interruption UNIX

Numéro	Nature	Traitant
0	Interruption d'horloge	`clockintr`
1	Interruption disque	`diskintr`
2	Interruption console	`ttyintr`
3	Interruption périphérique	`devintr`
4	Appel système (Interruption logicielle)	`softintr`
5	Autre interruption	`otherintr`

Conséquences de l'interruption

- L'interruption permet de suspendre un processus à n'importe quel moment de son exécution (préemption) et de **redonner la main au système d'exploitation**.
- Le SE (via sa routine de traitement, puis l'ordonnanceur) peut alors décider :
  - De reprendre le même processus une fois l'événement traité (simple restauration matérielle).
  - Ou d'activer un nouveau processus en fonction des priorités contenues dans les `PCB`.
- L'activation d'un nouveau processus nécessite alors d'effectuer un **changement de contexte complet** (et non pas une simple commutation de PSW).

Le mécanisme du changement de contexte

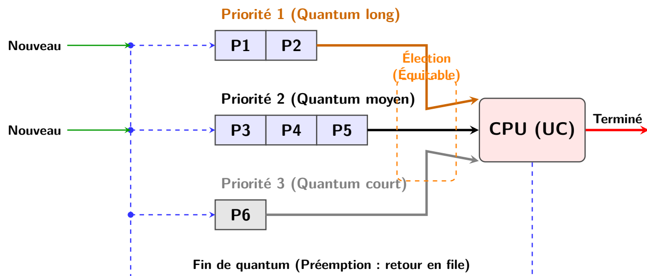


Construction d'un ordonnanceur

- Pour concevoir un ordonnanceur préemptif, il suffit de disposer :
  - D'un mécanisme d'interruption.
  - D'un mécanisme de commutation de contexte.
  - D'une horloge matérielle (Timer).
- On choisit un intervalle de temps (**quantum**) et l'horloge génère une interruption à la fin de chaque quantum.
- Cette interruption redonne le contrôle à l'ordonnanceur.
- L'ordonnanceur évalue dynamiquement les priorités des processus prêts et choisit celui qui doit s'exécuter.

Algorithme du tourniquet avec priorités

- **Files multiples :** Les processus prêts sont répartis dans plusieurs files FIFO selon leur niveau de priorité.
- **Prévention de la famine :** L'ordonnanceur attribue des quantums plus longs pour les processus plus prioritaires.
- **Tourniquet intra-file :** Au sein d'une file, chaque processus s'exécute pendant son *quantum*. S'il l'épuise, il est préempté et renvoyé à la fin de sa file ou dans une file "expirée".



Au-delà du Round Robin : les limites

- Les algorithmes classiques (comme le Round Robin avec priorités) présentent des limites sur les systèmes modernes :
  - Temps de calcul figé par le *quantum*.
  - Difficile de différencier efficacement un processus interactif (interface graphique) d'un processus lourd (calcul).
- **La philosophie Linux :** Plutôt que de distribuer des "tranches de temps" fixes, le noyau cherche à modéliser un "multitâche idéal" où le processeur serait parfaitement divisé entre tous les processus.
- C'est le passage d'une logique de **files d'attente stricts** à une logique de **temps d'exécution virtuel**.

L'approche historique récente : CFS (Completely Fair Scheduler)

- **Principe central :** Chaque processus possède un temps d'exécution virtuel (`vruntime`).
- L'ordonnanceur choisit toujours le processus ayant le `vruntime` le plus bas (celui qui a été le plus "défavorisé" jusqu'à présent).
- **Structure de données :** Les processus prêts sont stockés dans un **arbre Rouge-Noir** (Red-Black Tree), permettant de trouver le plus petit `vruntime` très rapidement ($O(\log n)$).
- **Gestion des priorités :** La priorité (valeur *nice* sous Unix) ne change pas l'ordre direct, mais modifie la *vitesse* à laquelle le `vruntime` s'écoule (un processus prioritaire voit son temps virtuel s'écouler plus lentement).

L'état de l'art aujourd'hui : EEVDF (depuis Linux 6.6)

- Malgré ses qualités, CFS gérait parfois mal la **latence** (le temps de réaction) des processus très courts face aux processus très gourmands.
- Le standard actuel s'appelle **EEVDF** (*Earliest Eligible Virtual Deadline First*).
- **Comment ça marche ?** Il calcule deux choses :
  1. *Éligibilité* : Le processus a-t-il reçu sa part équitable de CPU ?
  2. *Date butoir (Deadline)* : Quel est le délai maximum tolérable pour qu'il s'exécute ?
- **Élection :** Parmi les processus "éligibles", l'ordonnanceur choisit celui dont la date butoir est la plus proche, éliminant ainsi les micro-gels d'interface sans sacrifier l'équité globale.

Visualisation des processus en pratique

- **Ligne de commande (Linux/Unix) :**
  - La commande `ps` affiche une capture instantanée des processus. Sans option, elle se limite aux processus du terminal courant.
  - La commande `pstree` permet de voir la hiérarchie (l'arbre) des processus.
  - Pour une visualisation en temps réel : `top` ou `htop` (plus lisible).
- **Interface graphique (Windows) :**
  - Raccourci `Ctrl + Maj + Échap` (ouvre directement le Gestionnaire des tâches).
  - L'onglet "Détails" permet de voir les `PID` et l'état des processus.

1. Gestion de la multi-programmation par le SE
   - Multi-programmation
   - Table des processus
   - Changement de contexte
   - Ordonnanceur
   - Visualisations de processus

2. Utiliser la multi-programmation en C (sous Linux)
   - Création de processus
   - Élimination de processus
   - Synchronisation de processus
   - Recouvrement de processus
   - Quelques mots sur Windows

Identifiants de processus

Avant même de créer de nouveaux processus, il faut savoir les identifier :

- `pid_t getpid()` : Retourne l'identifiant du processus courant.
- `pid_t getppid()` : Retourne l'identifiant du processus père (*Parent PID*).
- Sous Linux, le type `pid_t` (défini dans `<sys/types.h>`) correspond généralement à un entier signé (`int` ou `long int`).

Création de processus avec `system()`

- **Bibliothèque standard C (`<stdlib.h>`) :**
  - `int system(const char *command);`
- **Fonctionnement :** Il crée un processus en lançant un shell (ex : `/bin/sh -c`) qui exécutera la commande passée en argument.
- **Attention :** `system()` n'est pas un appel système direct, mais une fonction de la bibliothèque C qui encapsule plusieurs appels (dont `fork` et `exec`).
- Cela rend `system()` lourd et potentiellement risqué (sécurité) par rapport à une gestion manuelle via appels système POSIX.

L'arbre des processus POSIX

- Sous Unix/Linux, les processus sont créés dynamiquement et forment une arborescence (dont la racine est le processus `init` ou `systemd`, PID 1).
- **Filiation :** Un processus peut créer d'autres processus (ses *fils*), qui à leur tour peuvent en créer d'autres.
- Un processus peut partager certaines ressources avec son père ou disposer d'une copie de ses ressources.
- Le processus père garde un certain contrôle sur ses processus fils :
  - Il peut leur envoyer des signaux (ex : pour les éliminer ou les suspendre).
  - Il peut se mettre en attente de leur fin d'exécution.

L'appel système `fork()`

- La fonction POSIX pour créer un processus est `pid_t fork(void);`.
- **Fonctionnement :** Crée un processus fils dont l'espace d'adressage est une copie **exacte** de celui du père au moment de l'appel.
- **Valeur de retour de `fork()`** (permet de différencier le code du père et du fils) :
  - `0` : code exécuté dans le processus fils.
  - `> 0` (le PID du fils) : code exécuté dans le processus père.
  - `-1` : échec de la création.
- Une fois le `fork()` réussi, père et fils s'exécutent en parallèle, le SE décidant de l'ordonnancement.

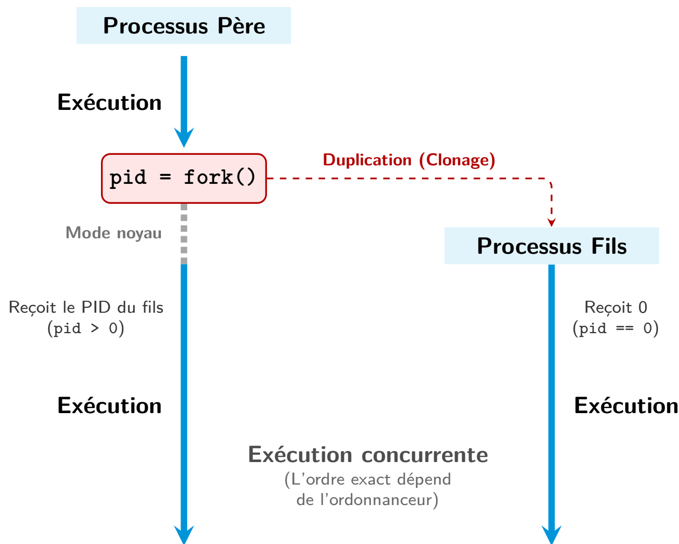
Différences entre Père et Fils après `fork()`

Au retour de `fork()`, on a deux processus presque identiques (le fils commence son exécution juste après l'instruction `fork()`).
Le fils **hérite** des descripteurs de fichiers ouverts.

Parmi les **différences** (reflétées dans la table des processus), on trouve :

- Le `PID` (unique) et le `PPID` (qui pointe vers le père).
- Les signaux en attente ne sont pas hérités.
- Les statistiques sur l'utilisation des ressources (temps CPU) sont remises à zéro pour le fils.
- Les verrous de fichiers posés par le père ne sont pas possédés par le fils.

Création de processus : l'appel système `fork()`



Exemple d'utilisation de `fork()`

Exemple de structure classique :

```
pid_t pid = fork();
if (pid == -1) {
    // Gestion d'erreur
} else if (pid == 0) {
    // Code du processus fils
} else {
    // Code du processus père
}
```

Remarque très importante sur `fork()`

Séparation de la mémoire

`fork()` duplique le contexte du processus. Dès lors, l'espace mémoire est séparé ! **Les variables du père et du fils ne sont PAS partagées.**

- Si le fils modifie une variable, cette modification n'est pas visible par le père (et inversement).
- Pour partager des données, il faut utiliser des mécanismes spécifiques d'IPC (mémoire partagée, tubes/pipes, etc.) ou recourir aux *threads* (processus légers).

Élimination d'un processus

- Un processus se termine normalement par lui-même en appelant :
  - `void exit(int status);` (ou via un `return` dans le `main()`).
- Un processus peut en éliminer un autre (s'il en a les droits, ex : même UID ou root) via l'envoi d'un signal :
  - `int kill(pid_t pid, int sig);` (ex : signal `SIGTERM` ou `SIGKILL`).
- Lorsqu'un processus se termine, ses ressources mémoire, CPU, et fichiers ouverts sont automatiquement désallouées par le système d'exploitation.

Le concept de processus Zombie

- **Processus Zombie :** Sous Unix/Linux, lorsqu'un processus se termine, son entrée dans la table des processus (le PCB réduit) est conservée **tant que** son processus père n'a pas récupéré son code de retour.
- Pour récupérer ce code de retour, le père doit utiliser les appels `wait()` ou `waitpid()`.
- Un zombie ne consomme pas de CPU ni de mémoire, mais occupe inutilement un emplacement dans la table des PID.
- Dans la commande `ps` ou `top`, les zombies sont indiqués par la lettre **Z** ou l'état *defunct*.

Attente de la fin d'un processus fils

Deux appels système principaux pour attendre la fin d'un fils :

- `pid_t wait(int *status);`
- `pid_t waitpid(pid_t pid, int *status, int options);`

Principe du `wait()` :

- L'appel bloque le processus père jusqu'à ce qu'un de ses fils se termine.
- La valeur de retour est le PID du fils qui vient de se terminer.
- L'entier `status` permet de savoir comment le fils s'est terminé, analysable via des macros dédiées.

Analyse du status de retour

Macros à utiliser sur la variable `status` (définies dans `<sys/wait.h>`) :

- `WIFEXITED(status)` : Vrai si le fils s'est terminé normalement (via `exit()` ou `return`).
  - Si vrai, `WEXITSTATUS(status)` donne le code de retour exact (0 à 255).
- `WIFSIGNALED(status)` : Vrai si le fils a été tué par un signal non intercepté (ex : `SIGKILL`, `Segfault`).
- `WIFSTOPPED(status)` : Vrai si le fils est temporairement stoppé (ex : via `Ctrl+Z`).

Détail sur `waitpid()`

- Le premier argument (`pid`) cible un processus précis :
  - Si `pid == -1`, on attend n'importe quel fils (équivalent exact de `wait()`).
  - Si `pid > 0`, on attend spécifiquement le fils portant ce PID.
- Le troisième argument (`options`) permet de modifier le comportement bloquant :
  - L'option `WNOHANG` rend l'appel **non-bloquant**. Si le processus attendu n'a pas encore terminé, la fonction rend la main immédiatement et retourne 0.

[Exemple de code avec waitpid à commenter en cours]

Remplacement d'espace d'adressage : `exec()`

- **Objectif :** Faire exécuter à notre processus un programme totalement différent de celui en cours.
- Linux offre une famille de fonctions (`execl`, `execv`, `execvp`, etc.) regroupées sous le terme générique **`exec()`**.
- **Fonctionnement de `exec()`** :
  - Écrase l'image mémoire (code, données, tas, pile) du processus actuel.
  - Charge en mémoire le nouveau programme dont le chemin est passé en paramètre.
  - Conserve le **même PID**, le même parent, et les mêmes descripteurs de fichiers ouverts.

Le duo classique : `fork()` + `exec()`

- Pour créer un nouveau processus exécutant un programme différent (ce que fait votre Shell à chaque commande), il faut combiner les deux mécanismes :
  1. Appeler `fork()` pour créer un fils.
  2. Le père fait généralement un `wait()` pour attendre la fin du programme.
  3. Le fils appelle `exec()` pour charger et exécuter le nouveau programme.
- **Attention :** Si `exec()` réussit, la fonction ne retourne **jamais** ! Le code source situé après la ligne `exec()` ne sera exécuté que si l'appel a échoué (ex : commande introuvable).

Création de processus sous Windows (API Win32)

- Pas de `fork()`. Sous Windows, on utilise la fonction `CreateProcess()` (qui demande une dizaine d'arguments !).
- Parmi ces arguments, on fournit directement :
  - Un pointeur vers le nom de l'exécutable.
  - Un booléen indiquant si le nouveau processus doit hériter des descripteurs (*handles*) du créateur.
  - Des pointeurs vers des structures qui recevront les informations du nouveau processus (PID, Handle).
- **Pas de hiérarchie stricte :** Contrairement à UNIX, Windows n'a pas d'arbre de processus rigide. Le processus créateur reçoit un *Handle* (jeton d'accès) sur le processus créé, qu'il peut librement transmettre à un tiers. Les notions de *Zombies* et de `wait` existent via des mécanismes différents (comme `WaitForSingleObject`).

Création de processus sous Windows : l'API native

- Pas de `fork` / `exec`, remplacés par `CreateProcess()`.
  - Crée à la fois l'espace mémoire du nouveau processus **et** le thread principal qui va exécuter le programme.
  - Fonction complexe : elle prend une dizaine de paramètres !
- **Pas de hiérarchie stricte (Parent/Enfant) :**
  - Sous UNIX, un processus enfant est lié à son parent (risques d'orphelins ou de processus "zombies").
  - Sous Windows, une fois créé, le processus est indépendant. Si le parent se termine, l'enfant continue son exécution normalement sans gestion particulière.
- **L'API Windows (Win32) toujours au cœur :** Elle reste le socle fondamental pour dialoguer avec le noyau NT, même sur les versions récentes.
- **Le rôle des surcouches modernes :** Les frameworks récents (.NET, WinUI) simplifient le code pour le développeur, mais invoquent toujours l'API native en arrière-plan.