Programmation système (1° partie) : Processus et Thread

Thierry Vaira

BTS SN-IR Avignon

v0.1 16 février 2016

Sommaire

Frères des Ecoles Chrétiennes
Saint-Jean-Baptiste de la Salle - Avignon

Qu'est-ce que le multitâche ?

- Un système d'exploitation est **multitâche** (*multitasking*) s'**il permet d'exécuter, de façon apparemment simultanée, plusieurs programmes informatiques** (*processus*).
- La simultanéité apparente est le résultat de l'**alternance rapide d'exécution des processus présents en mémoire** (notion de **temps partagé et de multiplexage**).
- Le passage de l'exécution d'un processus à un autre est appelé **commutation de contexte**.
- Ces commutations peuvent être initiées par les programmes eux-mêmes (**multitâche coopératif**) ou par le système d'exploitation (**multitâche préemptif**).

Frères des Ecoles Chrétiennes
Saint-Jean-Baptiste de la Salle - Avignon

Remarques

- Le **multitâche** n'est pas dépendant du **nombre de processeurs** présents physiquement dans l'ordinateur : un système multiprocesseur n'est pas nécessaire pour exécuter un système d'exploitation multitâche.
- Le **multitâche coopératif** n'est plus utilisé (cf. Windows 3.1 ou MAC OS 9).
- Unix et ses dérivés, Windows et MAC OS X sont des systèmes basés sur le **multitâche préemptif**.

À quoi ça sert ?

- Le multitâche permet de **paralléliser les traitements** par l'exécution simultanée de programmes informatiques.
- Exemples de besoins :
  - permettre à plusieurs utilisateurs de travailler sur la même machine.
  - utiliser un traitement de texte tout en naviguant sur le Web.
  - transférer plusieurs fichiers en même temps.
  - améliorer la conception : écrire plusieurs programmes simples, plutôt qu'un seul programme capable de tout faire, puis de les faire coopérer pour effectuer les tâches nécessaires.

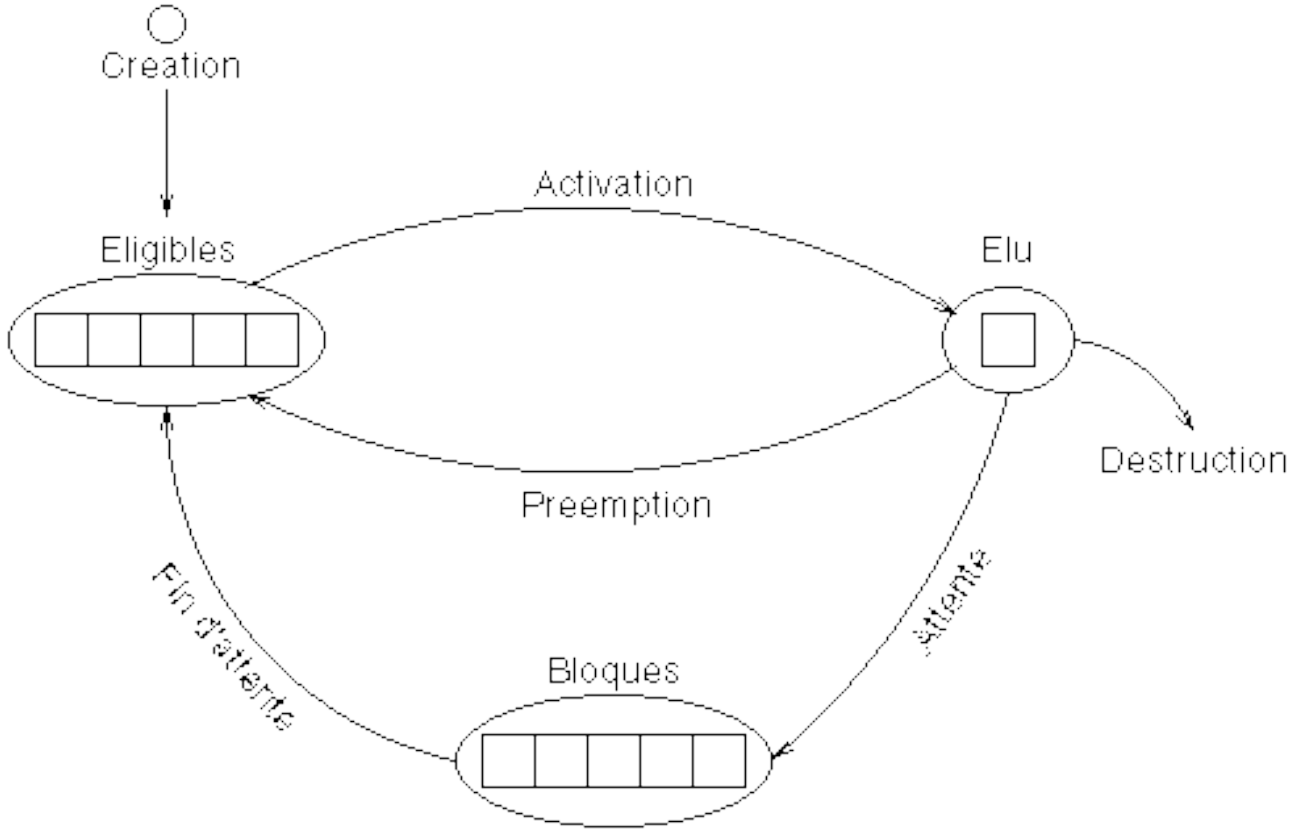
Comment ça marche ?

- La **préemption** est la capacité d'un système d'exploitation multitâche **à suspendre un processus au profit d'un autre**.
- Le **multitâche préemptif** est assuré par l'**ordonnanceur** (*scheduler*).
- L'**ordonnanceur** distribue le **temps du processeur entre les différents processus**. Il peut aussi interrompre à tout moment un processus en cours d'exécution pour permettre à un autre de s'exécuter.
- Une **quantité de temps définie** (*quantum*) est attribuée par l'**ordonnanceur** à chaque processus : les processus ne sont donc pas autorisés à prendre un temps non-défini pour s'exécuter dans le processeur.

Frères des Ecoles Chrétiennes
Saint-Jean-Baptiste de la Salle - Avignon

L'ordonnancement en action

Dans un ordonnancement (statique à base de priorités) avec préemption, un processus peut être préempté (remplacé) par n'importe quel processus plus prioritaire qui serait devenu prêt.



Synthèse

- Multitâche : exécution en parallèle de plusieurs tâches (*processus* ou *threads*).
- Commutation de contexte : passage de l'exécution d'un processus à un autre.
- Multitâche préemptif : mode de fonctionnement d'un système d'exploitation multitâche permettant de partager de façon équilibrée le temps processeur entre différents processus.
- Ordonnancement : mécanisme d'attribution du processeur aux processus (blocage, déblocage, élection, préemption).

Définitions

- Un **processus** (*process*) est un **programme en cours d'exécution** par un système d'exploitation.
- Un **programme** est une **suite d'instructions permettant de réaliser un traitement**. Il revêt un caractère statique.
- Une **image** représente l'**ensemble des objets (code, données, ...) et des informations (états, propriétaire, ...)** qui peuvent donner lieu à une exécution dans l'ordinateur.
- Un **processus** est donc l'**exécution d'une image**. Le processus est l'aspect dynamique d'une image.
- Un **fil d'exécution** (*thread*) est l'**exécution séquentielle d'une suite d'instructions**.

Frères des Ecoles Chrétiennes Saint-Jean-Baptiste de la Salle - Avignon

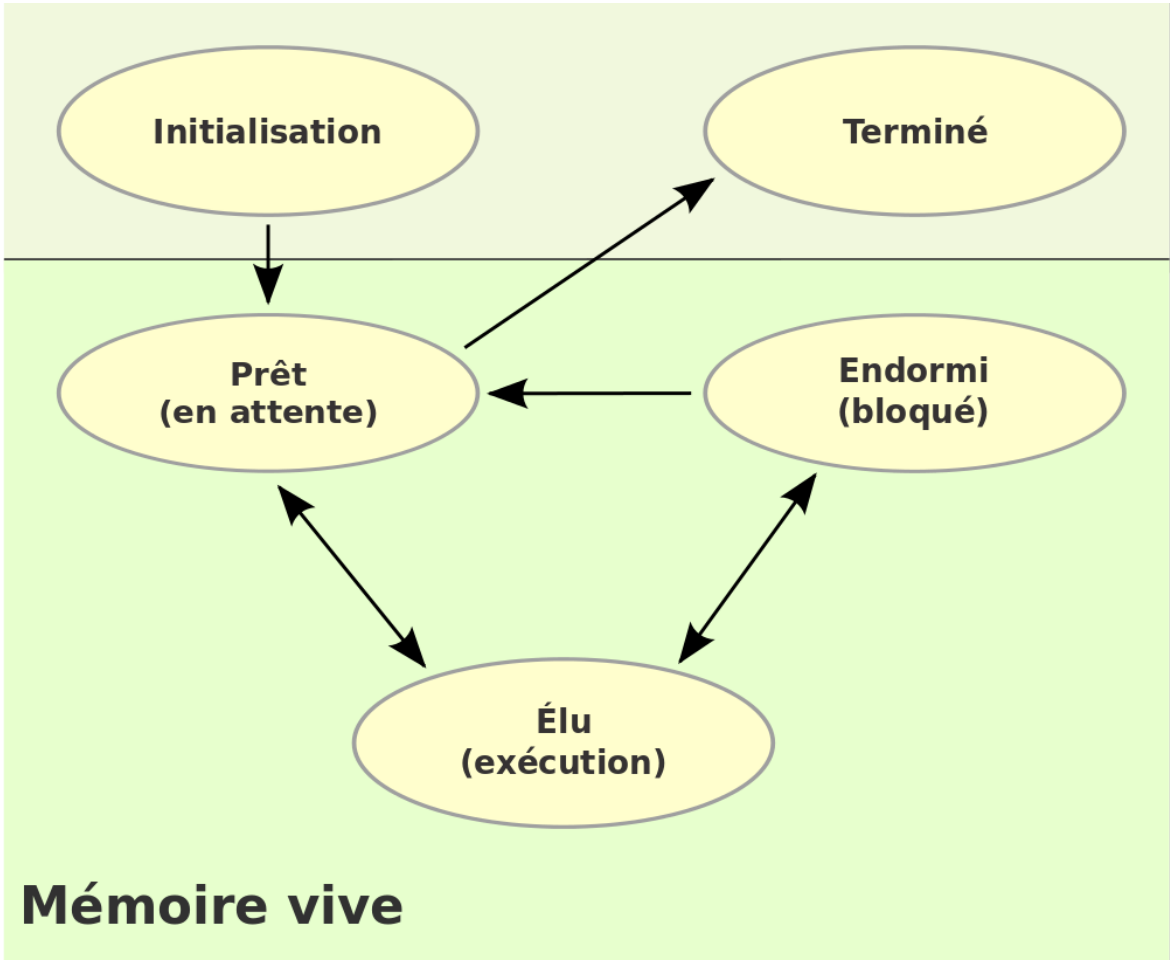
Rôle du système d'exploitation

- *Rappel : un système d'exploitation multitâche est capable d'exécuter plusieurs processus de façon quasi-simultanée.*
- Le **système d'exploitation** est chargé d'**allouer les ressources** (mémoires, temps processeur, entrées/sorties) nécessaires aux processus et d'**assurer son fonctionnement isolé** au sein du système.
- Un des rôles du système d'exploitation est d'**amener en mémoire centrale l'image mémoire d'un processus avant de l'élire et de lui allouer le processeur**. Le système d'exploitation peut être amené à sortir de la mémoire les images d'autres processus et à les copier sur disque. Une telle gestion mémoire est mise en oeuvre par un algorithme de **va et vient appelée aussi *swapping***.
- Il peut aussi fournir une **API** pour **permettre leur gestion et la communication inter-processus (IPC)**.

Frères des Ecoles Chrétiennes Saint-Jean-Baptiste de la Salle - Avignon

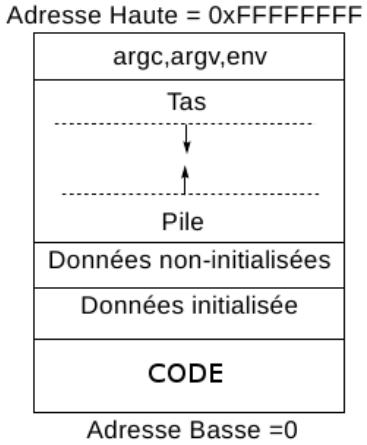
États d'un processus

- Ces états existent dans la plupart des systèmes d'exploitation :



Qu'est-ce qu'un processus ?

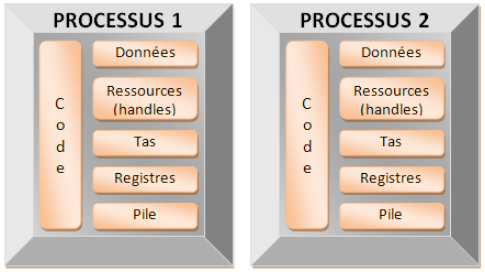
- L'image d'un **processus** comporte du **code machine exécutable**, une **zone mémoire** pour les données allouées par le processus, une **pile** ou *stack* (pour les variables locales des fonctions et la gestion des appels et retour des fonctions) et un **tas** ou *heap* pour les allocations dynamiques (`new`, `malloc`).



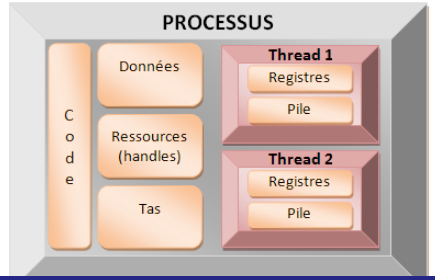
- Ce processus est une entité qui, de sa création à sa mort, est **identifié par une valeur numérique** : le **PID** (*Process IDentifier*).
- Chaque processus a un **utilisateur propriétaire**, qui est utilisé par le système pour **déterminer ses permissions d'accès aux ressources** (fichiers, ports réseaux, ...).

Processus lourd et léger

- *Rappel : L'exécution d'un processus se fait dans son contexte. Quand il y a changement de processus courant, il y a une commutation ou changement de contexte.*
- En raison de ce contexte, la plupart des systèmes offrent la distinction entre :
  - **« processus lourd »**, qui sont complètement isolés les uns des autres car ayant chacun leur contexte, et



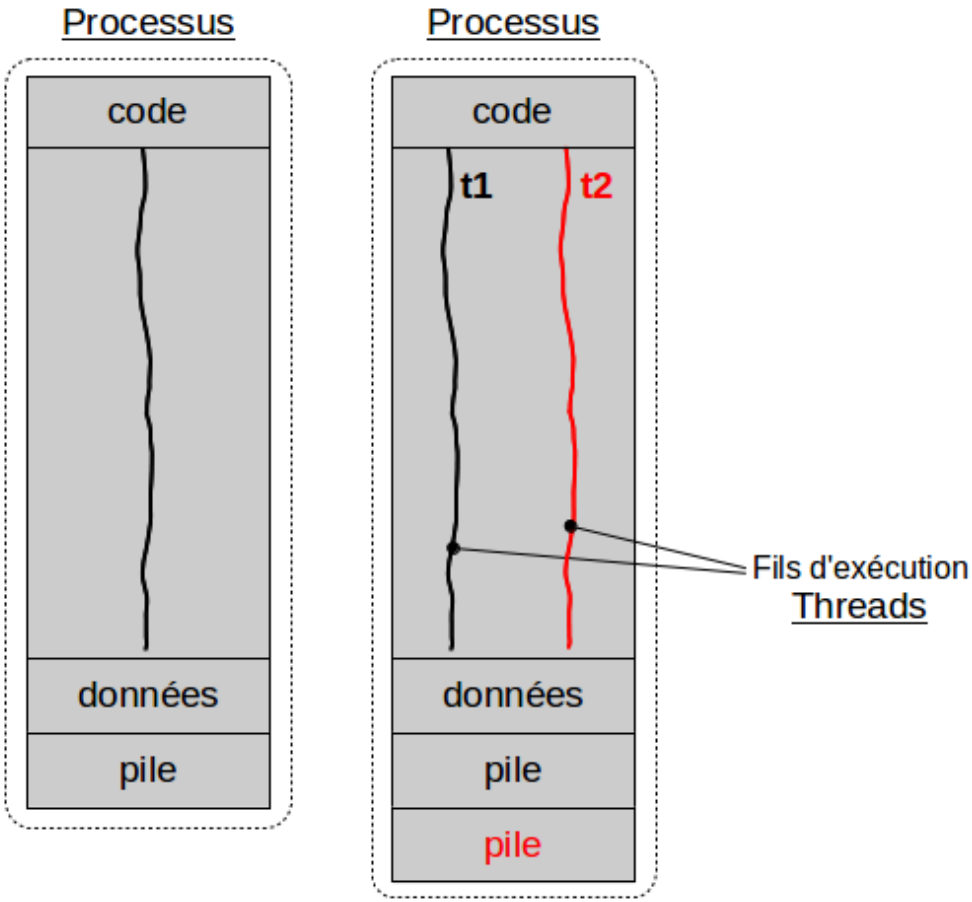
  - **« processus légers » (*threads*)**, qui partagent un contexte commun sauf la pile (les *threads* possèdent leur propre pile d'appel).



Frères des Ecoles Chrétiennes Saint-Jean-Baptiste de la Salle - Avignon

Processus lourd et léger (suite)

- Tous deux représentent l'exécution d'un ensemble d'instructions du langage machine d'un processeur (notion de fil).



Synthèse

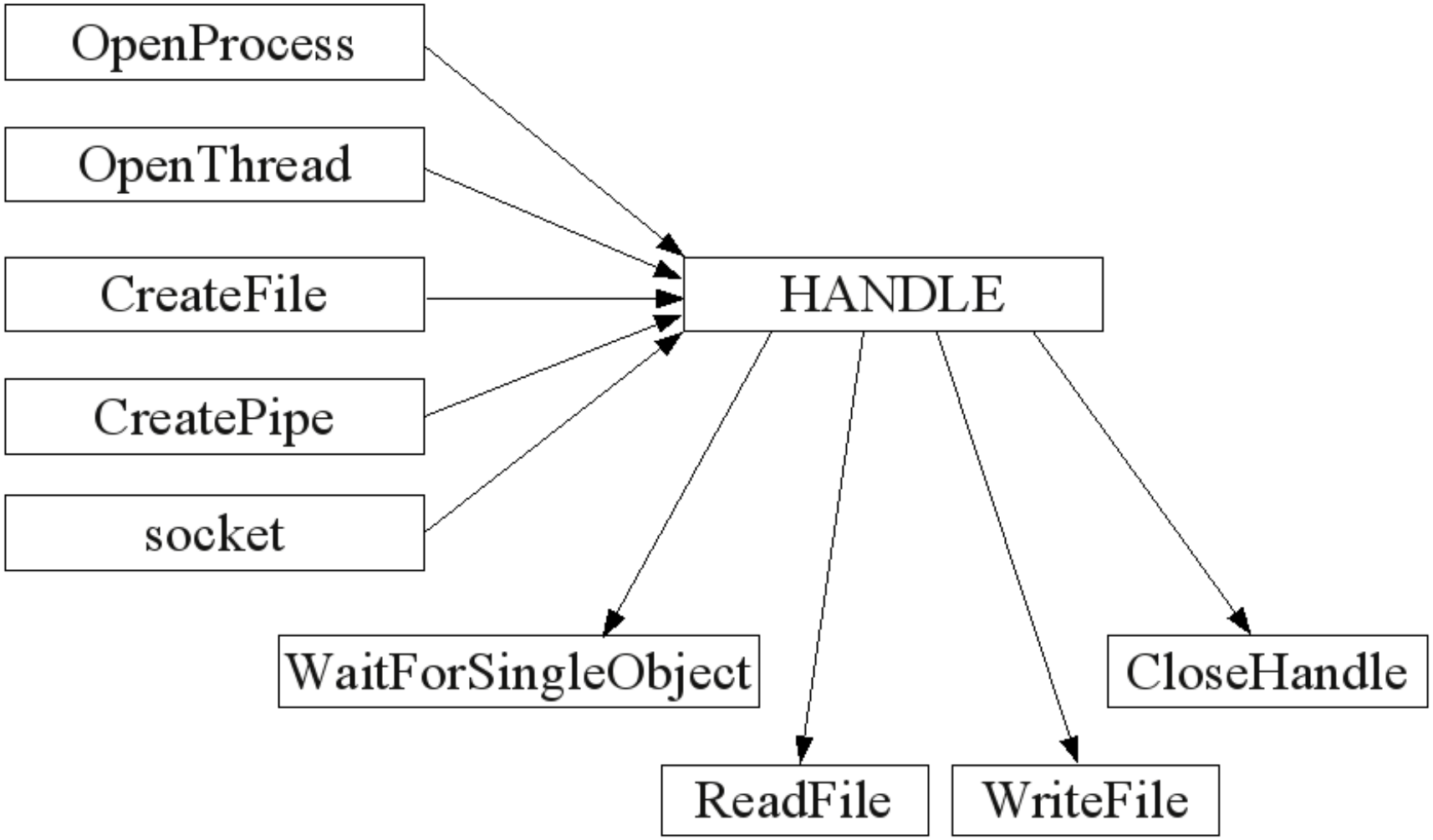
- Processus : programme en cours d'exécution. C'est l'exécution d'une image composée de code machine et de données mémoire.
- Contexte : image d'un processus en mémoire auquel s'ajoute son état (registres, ...).
- Processus lourd : c'est un processus « normal ». La création ou la commutation de contexte d'un processus a un coût pour l'OS. L'exécution d'un processus est réalisée de manière isolée par rapport auxtres processus.
- Processus léger : c'est un *thread*. Un processus lourd peut englober un ou plusieurs *threads* qui partagent alors le même contexte. C'est l'exécution d'une fonction (ou d'une méthode) au sein d'un processus et en parallèle avec les autres *threads* de ce processus. La commutation de *thread* est très rapide car elle ne nécessite pas de commutation de contexte.

Interface de programmation

- *Rappel : le noyau d'un système d'exploitation est vu comme un* ***ensemble de fonctions*** *qui forme l'****API****.*
- Chaque fonction ouvre l'**accès à un service offert par le noyau**.
- Ces fonctions sont regroupées au sein de la **bibliothèque** des **appels systèmes** (*system calls*) pour UNIX/Linux ou **WIN32** pour Windows.
- **POSIX** (*Portable Operating System Interface*) est une **norme relative à l'interface de programmation du système d'exploitation**. De nombreux systèmes d'exploitation sont conformes à cette norme, notamment les membres de la famille **Unix**.

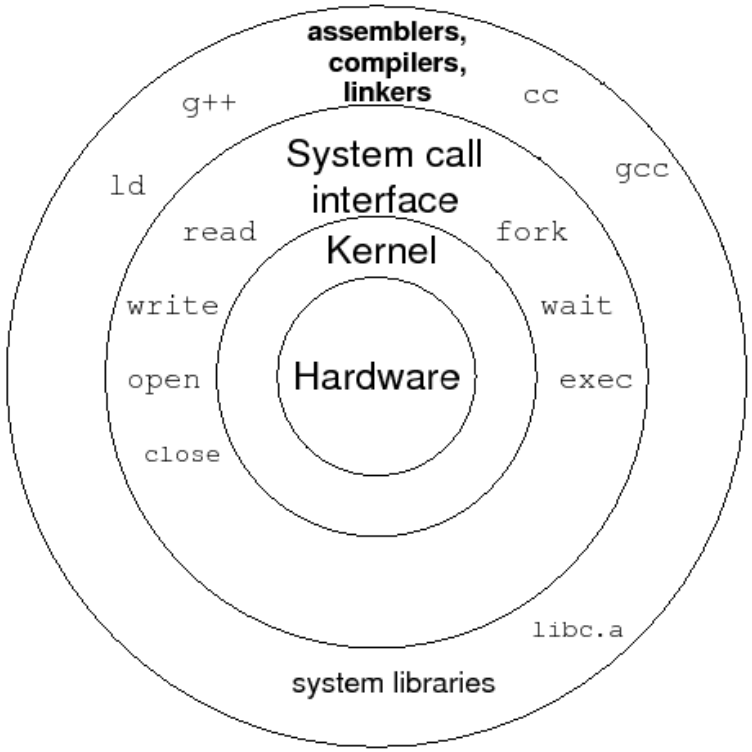
L'API Win32

- L'**API Windows** est orientée « **handle** » et non fichier.
- Un **handle** est un **identifiant d'objet système**.



L'API System Calls

- L'**API System Calls** d'UNIX est orientée « **fichier** » car dans ce système : TOUT est FICHIER !
- Un **descripteur de fichier** est **une clé abstraite** (c'est un entier) pour accéder à un fichier, c'est-à-dire le plus souvent une ressource du système.

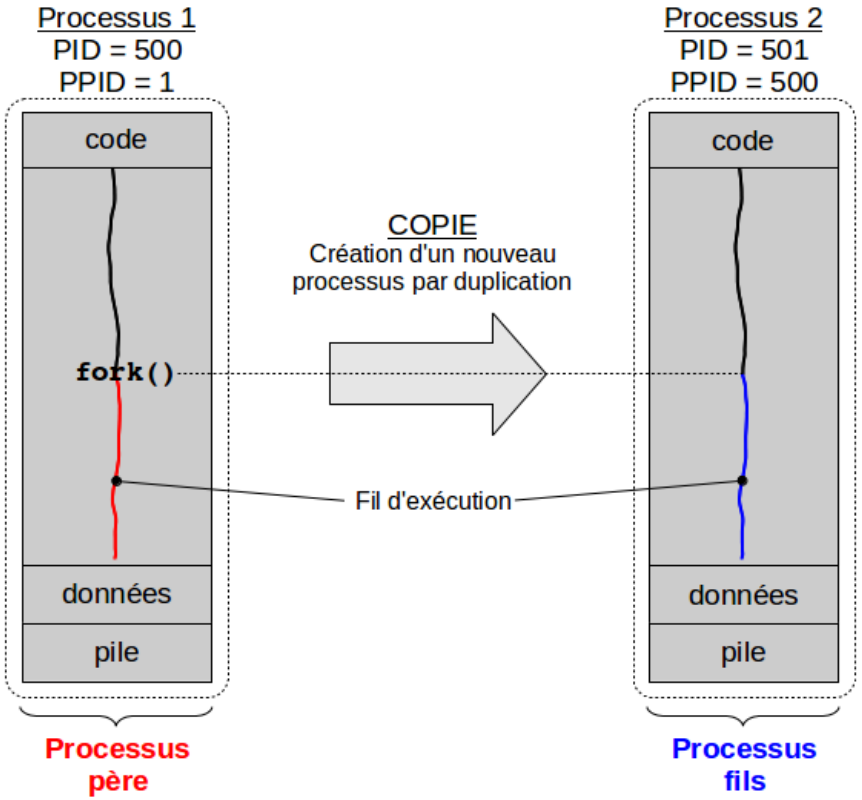


API WIN32 vs System Calls

UNIX	Win32	Description
fork	CreateProcess	Create a new process
waitpid	WaitForSingleObject	Can wait for a process to exit
execve	(none)	CreateProcess = fork + execve
exit	ExitProcess	Terminate execution
open	CreateFile	Create a file or open an existing file
close	CloseHandle	Close a file
read	ReadFile	Read data from a file
write	WriteFile	Write data to a file
lseek	SetFilePointer	Move the file pointer
stat	GetFileAttributesEx	Get various file attributes
mkdir	CreateDirectory	Create a new directory
rmdir	RemoveDirectory	Remove an empty directory
link	(none)	Win32 does not support links
unlink	DeleteFile	Destroy an existing file
mount	(none)	Win32 does not support mount
umount	(none)	Win32 does not support mount
chdir	SetCurrentDirectory	Change the current working directory
chmod	(none)	Win32 does not support security (although NT does)
kill	(none)	Win32 does not support signals
time	GetLocalTime	Get the current time

Création dynamique de processus

- Lors d'une opération `fork`, le noyau Unix crée un nouveau processus qui est une **copie conforme du processus père**. Le code, les données et la pile sont copiés et tous les fichiers ouverts par le père sont ainsi hérités par le processus fils.

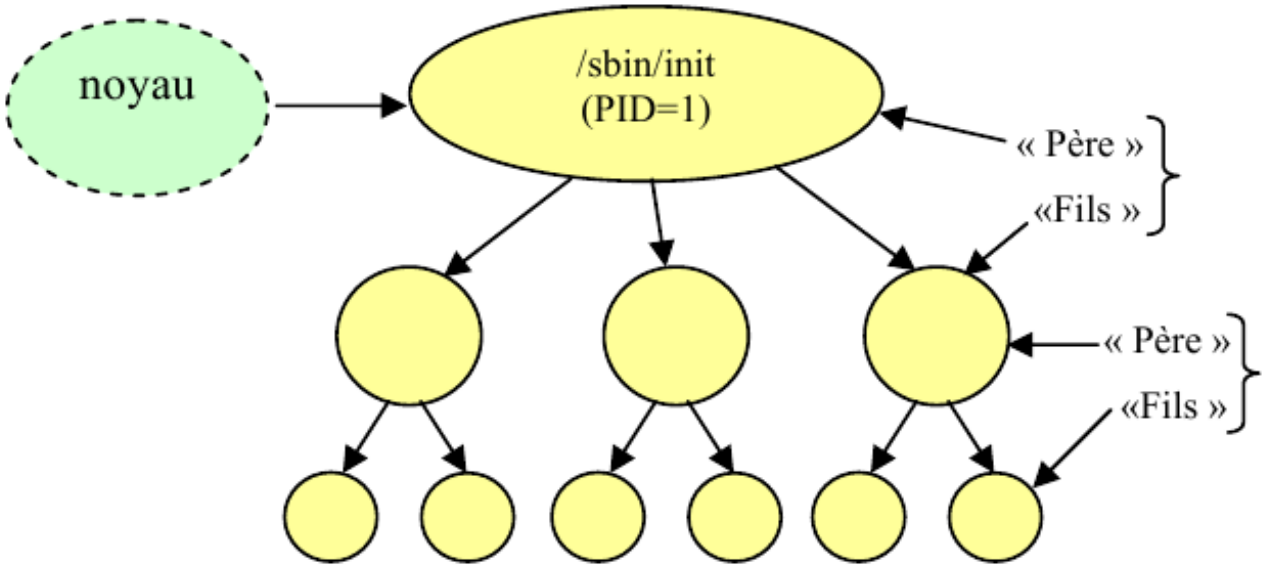


Exécution d'un nouveau processus

- *Rappel : après une opération fork, le noyau Unix a créé un nouveau processus qui est une copie conforme du processus qui a réalisé l'appel.*
- Si l'on désire exécuter du code à l'intérieur de ce nouveau processus, on utilisera un appel de type `exec` : `execl`, `execlp`, `execle`, `execv` ou `execvp`.
- La famille de fonctions `exec` remplace l'**image mémoire du processus en cours par un nouveau processus**.

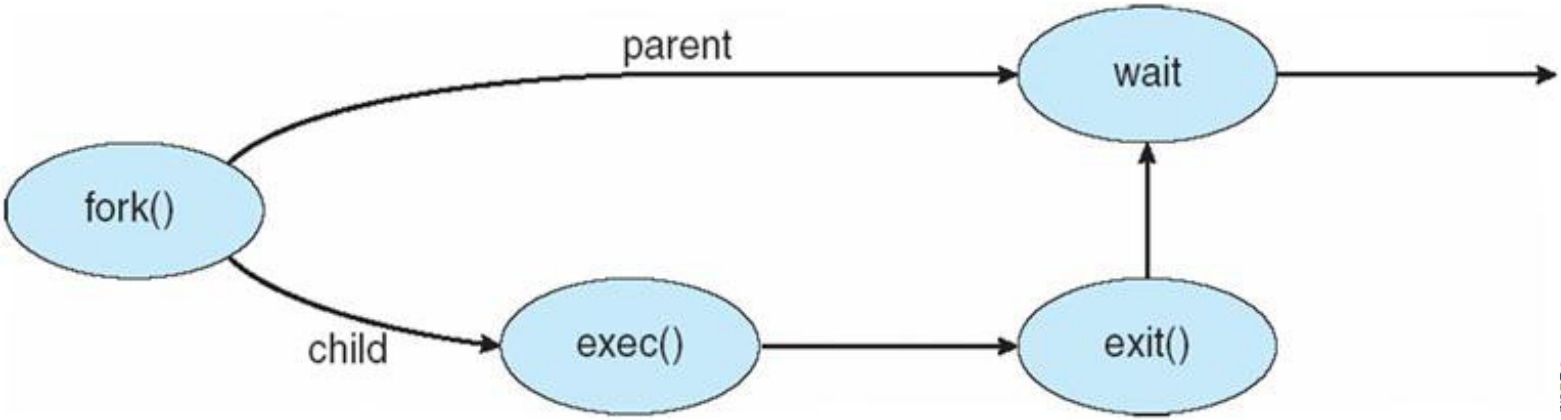
Généalogie des processus

- Chaque processus est identifié par un numéro unique, le **PID** (*Processus IDentification*). Un processus est créé par un autre processus (notion père-fils). Le **PPID** (*Parent PID*) d'un processus correspond au PID du processus qui l'a créé (son père).



Synchronisation des processus

- On ne peut présumer l'ordre d'exécution de ces processus (cf. politique de l'ordonnanceur).
- Il sera donc impossible de déterminer quels processus se termineront avant tels autres (y compris leur père). D'où l'existence, dans certains cas, d'un problème de synchronisation.
- La primitive `wait` permet l'élimination de ce problème en provoquant la suspension du processus appelant jusqu'à ce que l'un de ses processus fils se termine.



Opérations réalisables sur un processus

- Création : `fork`
- Exécution : `exec`
- Destruction : terminaison normale, auto-destruction avec `exit`, meurtre avec `kill` ou `Ctrl-C`
- Mise en attente/réveil : `sleep`, `wait`, `kill`
- Suspension et reprise : `Ctrl-Z` ou `fg`, `bg`
- Changement de priorité : `nice`

Les threads

- Les systèmes d'exploitation mettent en oeuvre généralement les *threads* :
  - Le standard des processus légers **POSIX** est connu sous le nom de `pthread`. Le standard POSIX est largement mis en oeuvre sur les systèmes **UNIX/Linux**.
  - **Microsoft** fournit aussi une API pour les processus légers : `WIN32 threads` (Microsoft Win32 API threads).
- En **C++**, il sera conseillé d'utiliser un *framework* (**Qt**, **Builder**, commoncpp, boost, ...) qui fournira le support des *threads* sous forme de classes prêtes à l'emploi. La version `C++11` intégrera le support de *threads* en standard.
- **Java** propose l'interface `Runnable` et une classe abstraite `Thread` de base.

Frères des Ecoles Chrétiennes
Saint-Jean-Baptiste de la Salle - Avignon

Synthèse

- API : c'est une interface de programmation (*Application Programming Interface*) qui est un ensemble de classes, de méthodes ou de fonctions qui offre des services pour d'autres logiciels. Elle est offerte par une bibliothèque logicielle ou un *framework*.
- WIN32 : c'est l'API de Windows qui permet la programmtion système (création de processus ou de thread, communication inter-processus, ...). Elle est orientée *handle*.
- *System calls* : c'est l'API Unix/Linux composée d'appels systèmes pour la création de processus ou de thread, la communication inter-processus, ...). Elle est orientée *fichier*.
- POSIX : c'est une norme d'API que l'on retrouve notamment sous Unix.