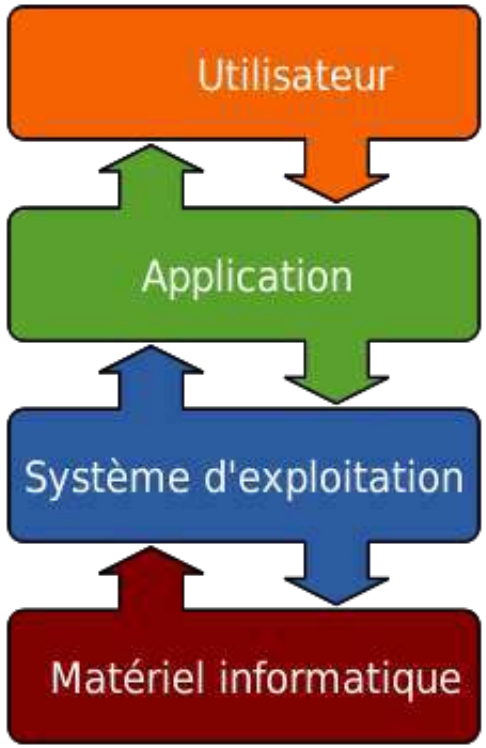
Programmation Système : les processus



« La véritable "idée" d'un système d'exploitation est d'utiliser les fonctionnalités du matériel, et de les placer derrière une couche d'appels de haut niveau. »

Linus Benedict Torvalds

Multiprogrammation - Définitions

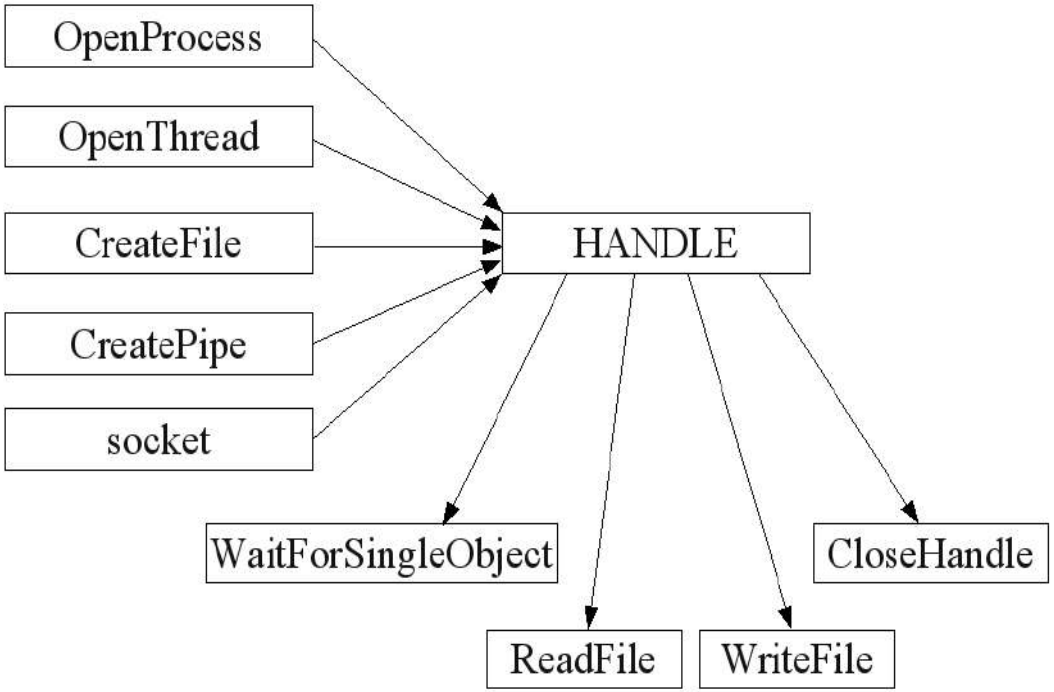
- La **multiprogrammation** se caractérise par le partage du processeur par plusieurs processus. Le temps est découpé en intervalles, chacun d'eux étant alloué successivement aux différents processus donnant l'illusion de simultanéité.
- Un **programme** est une suite d'instructions permettant de réaliser un traitement. Il revêt un caractère statique.
- Une **image** représente l'ensemble des objets et des informations qui peuvent donner lieu à une exécution dans l'ordinateur
- Un **processus** est l'exécution d'une image. Le processus est l'aspect dynamique d'une image.
- C'est un des rôles du système d'exploitation 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 œuvre par un algorithme de va et vient appelée aussi **swapping**.

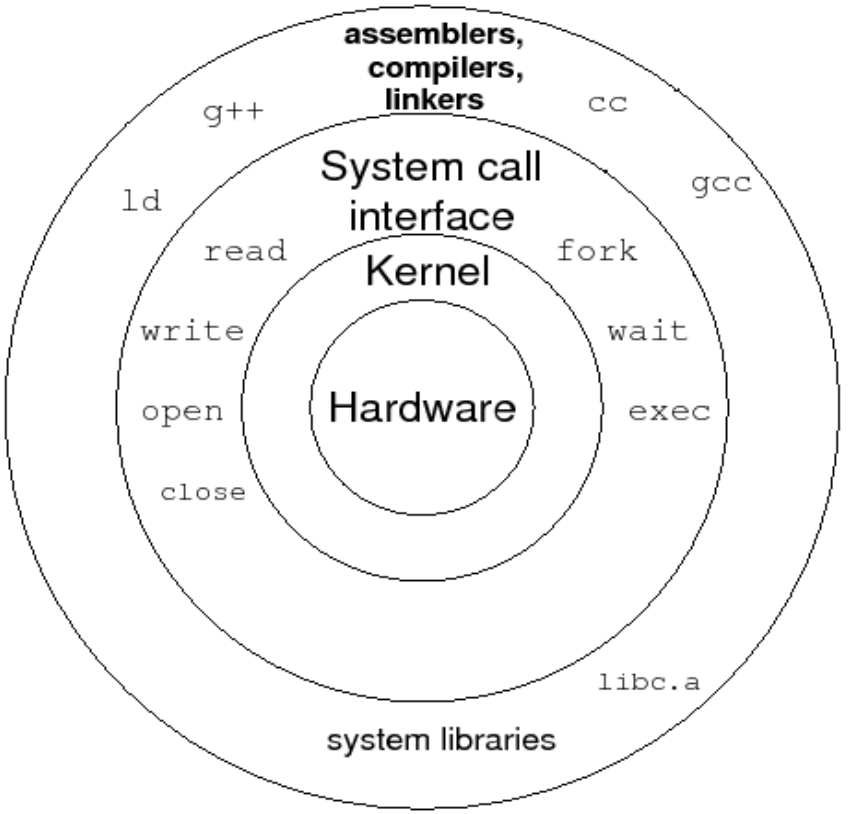
Interface de programmation (rappels)

- Le noyau est vu comme un ensemble de fonctions (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.

API WIN32 vs System Calls UNIX (1)





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

- L'API 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 (2)

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

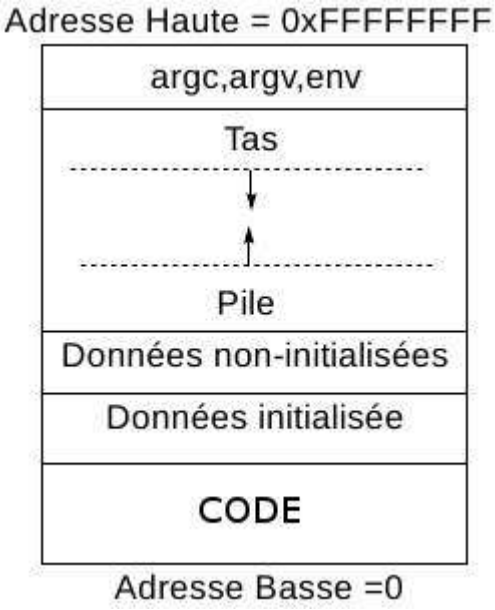
Processus

- Un processus comporte du code machine exécutable, une zone mémoire (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.

- Ce processus est une entité qui, de sa création à sa mort, est identifié par une valeur numérique : le PID (*Process IDentifier*).

- Tous les processus sont donc associés à une entrée dans la table des processus qui est interne au noyau.

- Chaque processus a un utilisateur propriétaire, qui est utilisé par le système pour déterminer ses permissions d'accès aux fichiers.



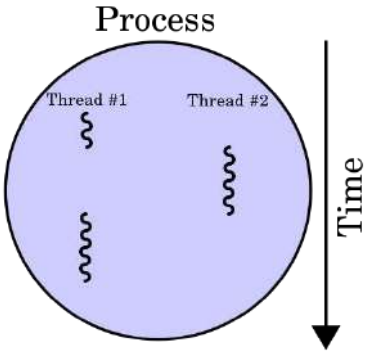
- *Remarques : Les commandes* **`ps`** *et* **`top`** *listent les processus sous UNIX/Linux et, sous Windows on utilisera le gestionnaire de tâches (*`taskmgr.exe`*).*

Contexte d'un processus

- Le **contexte** d'un processus (*Process Control Block*) est l'ensemble de :
  1. son état
  2. son mot d'état : en particulier la valeur des registres actifs et le compteur ordinal
  3. les valeurs des variables globales statiques ou dynamiques
  4. son entrée dans la table des processus
  5. les données privées du processus (*zone u*)
  6. Les piles *user* et *system*
  7. les zones de code et de données.

- 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**. Le noyau s'exécute alors dans le nouveau contexte.

- En raison de ce contexte, on parle de **processus lourd**, en opposition aux **processus légers** que sont les ***threads***
(car ceux-ci partagent une grande partie du contexte où ils s'exécutent).



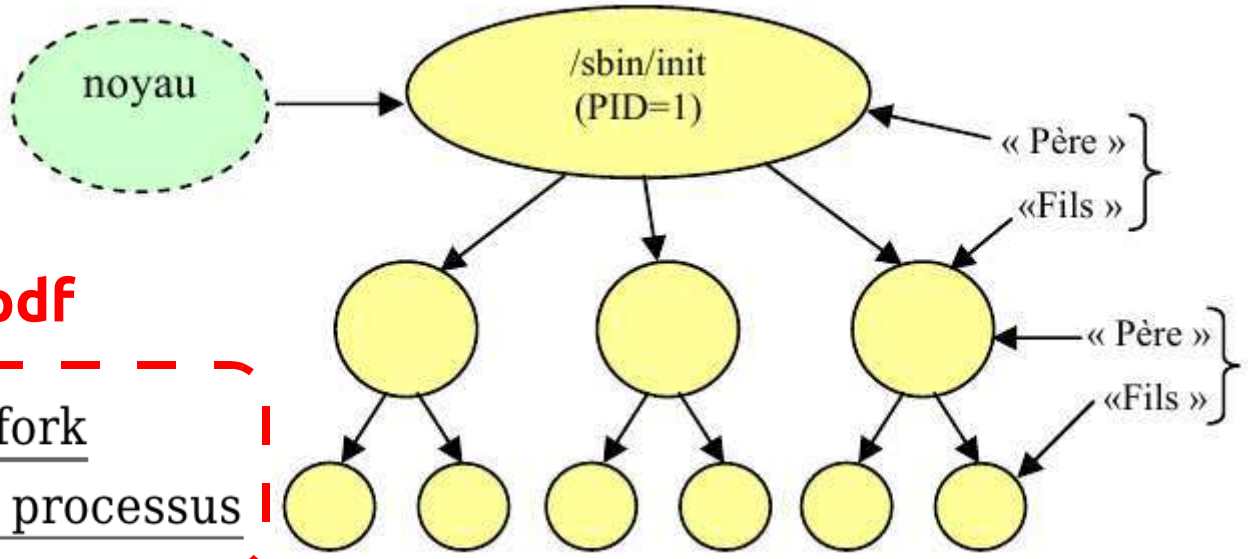
Généalogie des processus

- La création d'un processus étant réalisée par un **appel système** (***fork*** sous UNIX/Linux). Chaque processus est identifié par un numéro unique, le **PID** (*Processus IDentification*).
- Un processus est donc forcément 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)
- Exemple sous UNIX/Linux :

tp-sys-processus.pdf

Séquence 1 : l'appel fork

Séquence 2 : généalogie de processus



Opérations réalisables sur un processus

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

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.

- Lors de l'initialisation du système (**boot**), le noyau crée plusieurs processus spontanés dans l'espace de l'utilisateur. Ces processus sont dits spontanés parce qu'ils ne sont pas créés par le mécanisme *fork* traditionnel.

- Le nom et la nature des processus spontanés varient d'un système à l'autre. **`init`** est le seul processus à part entière. Le PID d'*init* vaut toujours **1** et c'est l'ancêtre de tous les processus utilisateurs et de presque tous les processus système.

Attributs d'un processus

- **PID** : chaque processus est identifié par un numéro unique, le PID (Processus IDentification). Les PID sont attribués au fur et à mesure de la création des processus.
- **PPID** : le PPID correspond au PID du processus qui l'a créé (son père).
- L'UID d'un processus est un numéro d'identification d'utilisateur qui correspond à la personne qui l'a créé, ou plus précisément à l'EUID du processus parent.
- L'EUID est l'utilisateur effectif de l'utilisateur, un UID supplémentaire qui permet de déterminer les ressources et les fichiers auxquels un processus a le droit d'accéder à n'importe quel moment.
- Le GID est le numéro d'identification de groupe d'un processus. L'EGID est au GID ce que le GPID est au PID.

Exécution d'un nouveau code

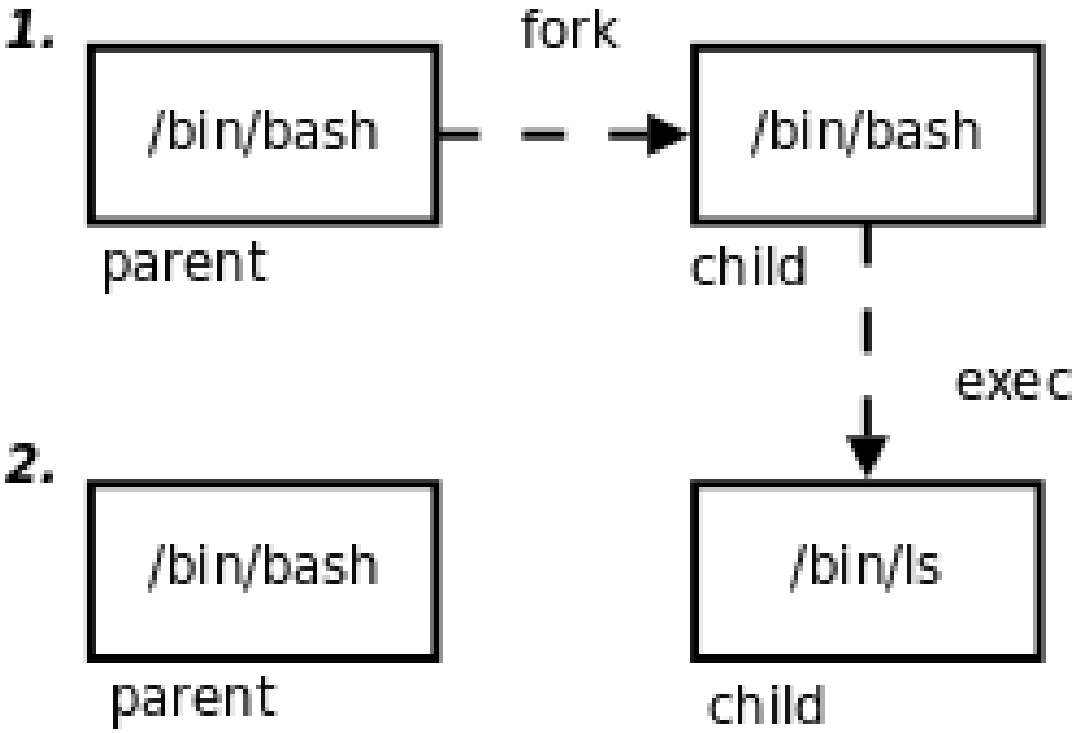
- Rappel : après une opération **`fork`**, le noyau Unix a créé un nouveau processus qui est une copie conforme du processus qui a réalisé l'appel.

- 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.

Exécution d'une commande

- C'est ce principe qui est par exemple utilisé lorsque l'on exécute une commande (**`ls`**) à partir du *shell* (**`bash`**) :

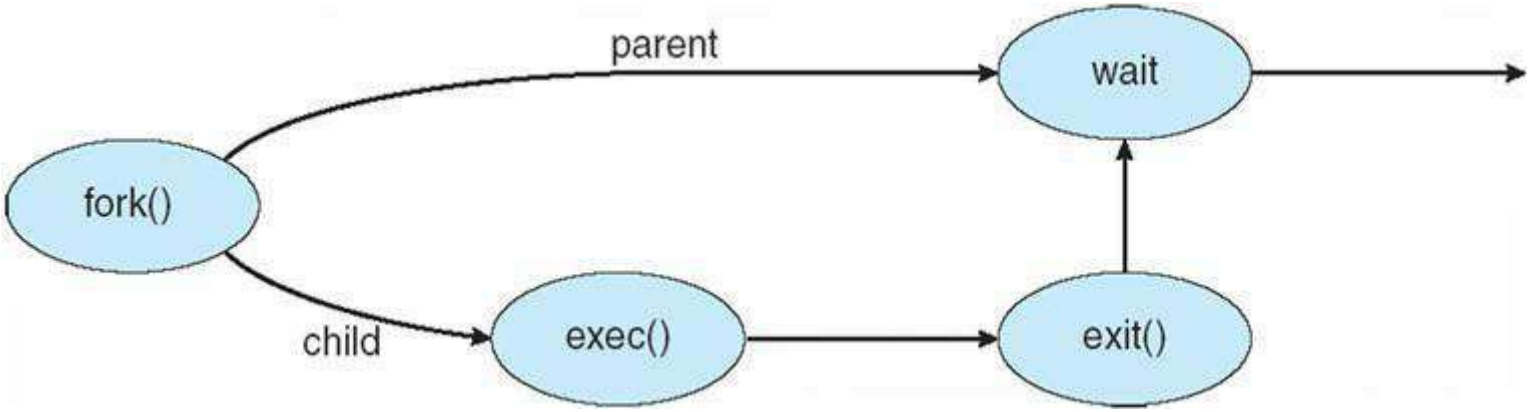


tp-sys-processus.pdf

Séquence 3 : les appels exec

Synchronisation des terminaisons

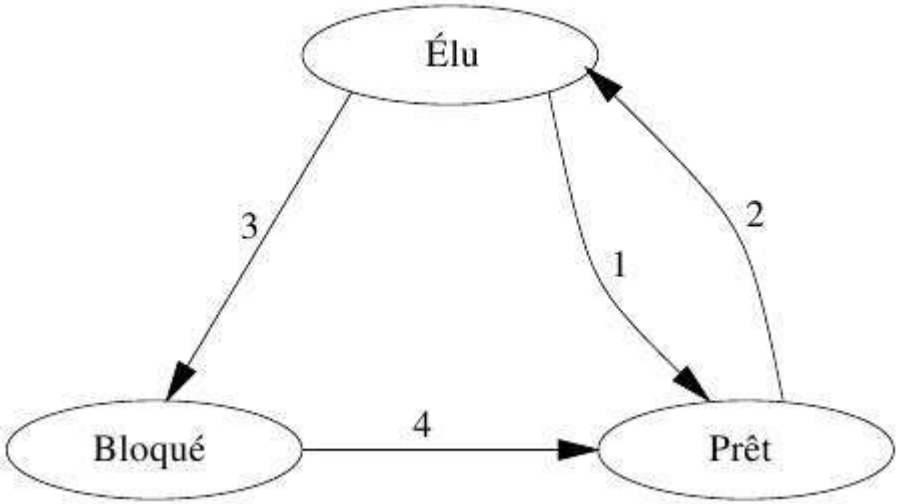
- Rappel : les processus créés par des **`fork`** s'exécutent de façon concurrente avec leur père. On ne peut présumer l'ordre d'exécution de ces processus (cf. politique de l'ordonnanceur).
- Il sera donc impossible de déterminer quels processus se termineront avant tels autres (y compris leur père). 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.



États d'un processus

- Le processus est une activité dynamique et possède un **état** qui évolue au cours du temps. Ce processus transitera par différents états selon que :
  - il s'exécute (ACTIF ou Élu)
  - il attend que le noyau lui alloue le processeur (PRET)
  - il attend qu'un événement se produise (ATTENTE ou Bloqué)

- C'est l'**ordonnanceur** (*scheduler*) qui contrôle l'exécution et les états des processus.



tp-sys-processus.pdf

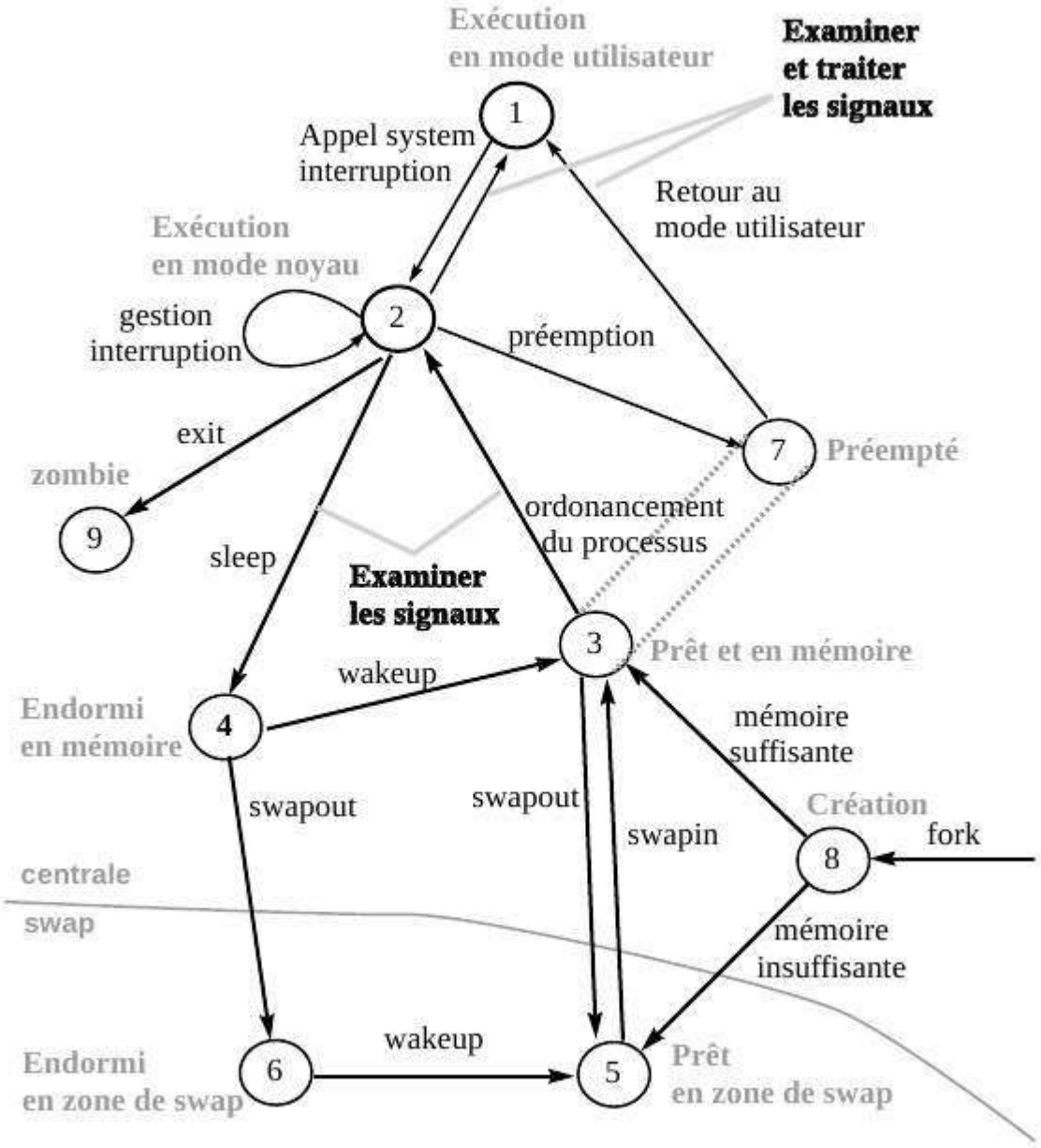
Séquence 4 : synchronisation des terminaisons

Séquence 5 : état d'un processus

Diagramme d'états sous Linux

La **préemption** se définit comme la réquisition du processeur pour l'exécution d'une tâche et d'une seule pendant un temps déterminé.

Dans un ordonnancement (statique à base de priorités) avec préemption : lorsque le processeur est inactif, la tâche prête de plus haute priorité sera choisie pour être exécutée. A chaque instant cette tâche peut être préemptée (remplacée) par n'importe quelle tâche plus prioritaire qui serait devenue prête.



Conclusion

- Il est conseillé avant de continuer de revoir :
  - Les définitions (programme, multiprogrammation, processus, image, préemption, ...)
  - L'exemple fourni pour l'environnement Windows

- Il reste à voir entre autres :
  - Les threads
  - L'ordonnancement des processus
  - La communication entre processus