# Programmation Système et Réseau

## Processus et Multiprocessing

Équipe pédagogique

CY Tech

# Bibliographie - Sitographie


- Slides de Juan Ángel Lorenzo del Castillo, Seytkamal Medetov et Son Vu
- Linux : Programmation système et réseau de Joëlle Delacroix Dunod
- Système d'exploitation de Andrew Tanenbaum, Pearson Education
- Linux Kernel Map : https://makelinux.github.io/kernel/map/
- https://cours.polymtl.ca/inf2610/documentation/notes/chap3.pdf

# Introduction

**Programmation Système**

Développement de programmes qui font partie du système d'exploitation d'un ordinateur, qui en réalisent les fonctions et qui utilisent les fonctions avancées de celui-ci.

# Introduction

**Programmation Système**

Développement de programmes qui font partie du système d'exploitation d'un ordinateur, qui en réalisent les fonctions et qui utilisent les fonctions avancées de celui-ci.

**Exemples**

- La gestion des processus,

# Introduction

**Programmation Système**

Développement de programmes qui font partie du système d'exploitation d'un ordinateur, qui en réalisent les fonctions et qui utilisent les fonctions avancées de celui-ci.

**Exemples**

- La gestion des processus,
- l'accès aux fichiers,

# Introduction

**Programmation Système**

Développement de programmes qui font partie du système d'exploitation d'un ordinateur, qui en réalisent les fonctions et qui utilisent les fonctions avancées de celui-ci.

**Exemples**

- La gestion des processus,
- l'accès aux fichiers,
- la programmation réseau, les entrées/sorties, la gestion de la mémoire...

# Objectif du cours

**Apprendre la programmation avec le système.**

# Objectif du cours

**Apprendre la programmation avec le système.**

- Comment créer des applications qui communiquent avec l'OS...

# Objectif du cours

**Apprendre la programmation avec le système.**

- Comment créer des applications qui communiquent avec l'OS...
- ...ou avec d'autres applications, en se synchronisant correctement.

# Objectif du cours

**Apprendre la programmation avec le système.**

- Comment créer des applications qui communiquent avec l'OS...
- ...ou avec d'autres applications, en se synchronisant correctement.
- Sous Linux.

# Objectif du cours

**Apprendre la programmation avec le système.**

- Comment créer des applications qui communiquent avec l'OS...
- ...ou avec d'autres applications, en se synchronisant correctement.
- Sous Linux.

**Pourquoi sous Linux ?**

# Objectif du cours

**Apprendre la programmation avec le système.**

- Comment créer des applications qui communiquent avec l'OS...
- ...ou avec d'autres applications, en se synchronisant correctement.
- Sous Linux.

**Pourquoi sous Linux ?**

- Les concepts sont généraux et se retrouvent dans tous les systèmes d'exploitation,

# Objectif du cours

**Apprendre la programmation avec le système.**

- Comment créer des applications qui communiquent avec l'OS...
- ...ou avec d'autres applications, en se synchronisant correctement.
- Sous Linux.

**Pourquoi sous Linux ?**

- Les concepts sont généraux et se retrouvent dans tous les systèmes d'exploitation,
- Les sources du noyau de linux sont accessibles et sont "open source",

# Objectif du cours

**Apprendre la programmation avec le système.**

- Comment créer des applications qui communiquent avec l'OS...
- ...ou avec d'autres applications, en se synchronisant correctement.
- Sous Linux.

**Pourquoi sous Linux ?**

- Les concepts sont généraux et se retrouvent dans tous les systèmes d'exploitation,
- Les sources du noyau de linux sont accessibles et sont "open source",
- Les interfaces sont normalisées (norme POSIX),

# Objectif du cours

**Apprendre la programmation avec le système.**

- Comment créer des applications qui communiquent avec l'OS...
- ...ou avec d'autres applications, en se synchronisant correctement.
- Sous Linux.

**Pourquoi sous Linux ?**

- Les concepts sont généraux et se retrouvent dans tous les systèmes d'exploitation,
- Les sources du noyau de linux sont accessibles et sont "open source",
- Les interfaces sont normalisées (norme POSIX),
- Le noyau Linux est écrit en C.

# Objectif du cours

**Apprendre la programmation avec le système.**

- Comment créer des applications qui communiquent avec l'OS...
- ...ou avec d'autres applications, en se synchronisant correctement.
- Sous Linux.

**Pourquoi sous Linux ?**

- Les concepts sont généraux et se retrouvent dans tous les systèmes d'exploitation,
- Les sources du noyau de linux sont accessibles et sont "open source",
- Les interfaces sont normalisées (norme POSIX),
- Le noyau Linux est écrit en C.

**Pré-requis du cours**

- Savoir coder en C sous Linux : processus, pointeurs, l'allocation dynamique, etc.

# Objectif du cours

**Apprendre la programmation avec le système.**

- Comment créer des applications qui communiquent avec l'OS...
- ...ou avec d'autres applications, en se synchronisant correctement.
- Sous Linux.

**Pourquoi sous Linux ?**

- Les concepts sont généraux et se retrouvent dans tous les systèmes d'exploitation,
- Les sources du noyau de linux sont accessibles et sont "open source",
- Les interfaces sont normalisées (norme POSIX),
- Le noyau Linux est écrit en C.

**Pré-requis du cours**

- Savoir coder en C sous Linux : processus, pointeurs, l'allocation dynamique, etc.
- Connaissances de la matière *ING1-Système d'exploitation*.

# Structure du cours

1. Processus et Multiprocessing
2. Recouvrement des processus
3. Signaux
4. Tubes, File de messages, Mémoire partagée
5. Multithreading
6. Semaphores
7. Sockets

3 cours magistraux = 4,5h
8 TD avec un peu de cours au début = 22,5h

# Évaluation

- Un DS papier.
- Deux heures.
- Aucun document autorisé.
- Remplacement par un Projet.

# Rappels

Système d'exploitation (SE ou OS)

# Rappel sur le SE/OS

Un SE doit pouvoir gérer :

# Rappel sur le SE/OS

Un SE doit pouvoir gérer :

- la mémoire

# Rappel sur le SE/OS

Un SE doit pouvoir gérer :

- la mémoire
- les E/S

# Rappel sur le SE/OS

Un SE doit pouvoir gérer :

- la mémoire
- les E/S
- les fichiers

# Rappel sur le SE/OS

Un SE doit pouvoir gérer :

- la mémoire
- les E/S
- les fichiers
- les utilisateurs

# Rappel sur le SE/OS

Un SE doit pouvoir gérer :

- la mémoire
- les E/S
- les fichiers
- les utilisateurs
- les processus

# Rappel sur le SE/OS

Les outils utilisés sont :

# Rappel sur le SE/OS

Les outils utilisés sont :

- la mémoire virtuelle pour la mémoire

# Rappel sur le SE/OS

Les outils utilisés sont :

- la mémoire virtuelle pour la mémoire
- les ITs pour les E/S et les processus (temps partagé)

# Rappel sur le SE/OS

Les outils utilisés sont :

- la mémoire virtuelle pour la mémoire
- les ITs pour les E/S et les processus (temps partagé)
- le contexte (registre, environnement) pour les processus

# Rappel sur le SE/OS

Les outils utilisés sont :

- la mémoire virtuelle pour la mémoire
- les ITs pour les E/S et les processus (temps partagé)
- le contexte (registre, environnement) pour les processus
- les droits pour les utilisateurs et les fichiers

# Rappel sur le SE/OS

Les outils utilisés sont :

- la mémoire virtuelle pour la mémoire
- les ITs pour les E/S et les processus (temps partagé)
- le contexte (registre, environnement) pour les processus
- les droits pour les utilisateurs et les fichiers

On y ajoute des mécanismes de :

# Rappel sur le SE/OS

Les outils utilisés sont :

- la mémoire virtuelle pour la mémoire
- les ITs pour les E/S et les processus (temps partagé)
- le contexte (registre, environnement) pour les processus
- les droits pour les utilisateurs et les fichiers

On y ajoute des mécanismes de :

- sécurité pour la mémoire, les fichiers et les utilisateurs

# Rappel sur le SE/OS

Les outils utilisés sont :

- la mémoire virtuelle pour la mémoire
- les ITs pour les E/S et les processus (temps partagé)
- le contexte (registre, environnement) pour les processus
- les droits pour les utilisateurs et les fichiers

On y ajoute des mécanismes de :

- sécurité pour la mémoire, les fichiers et les utilisateurs
- communication entre les processus

# Rappel sur le SE/OS

Les outils utilisés sont :

- la mémoire virtuelle pour la mémoire
- les ITs pour les E/S et les processus (temps partagé)
- le contexte (registre, environnement) pour les processus
- les droits pour les utilisateurs et les fichiers

On y ajoute des mécanismes de :

- sécurité pour la mémoire, les fichiers et les utilisateurs
- communication entre les processus
- synchronisation des processus

# Rappel sur le SE/OS

Le SE propose des primitives (fonctions) utilisables par des programmes et permettant l'utilisation des ressources :

# Rappel sur le SE/OS

Le SE propose des primitives (fonctions) utilisables par des programmes et permettant l'utilisation des ressources :

- gestion des fichiers (ouvrir, fermer, lire, écrire, ...)

# Rappel sur le SE/OS

Le SE propose des primitives (fonctions) utilisables par des programmes et permettant l'utilisation des ressources :

- gestion des fichiers (ouvrir, fermer, lire, écrire, ...)
- gestion du système de fichiers (créer/supprimer/parcourir les répertoires)

# Rappel sur le SE/OS

Le SE propose des primitives (fonctions) utilisables par des programmes et permettant l'utilisation des ressources :

- gestion des fichiers (ouvrir, fermer, lire, écrire, ...)
- gestion du système de fichiers (créer/supprimer/parcourir les répertoires)
- gestion de la mémoire (allouer, libérer, partager)

# Rappel sur le SE/OS

Le SE propose des primitives (fonctions) utilisables par des programmes et permettant l'utilisation des ressources :

- gestion des fichiers (ouvrir, fermer, lire, écrire, ...)
- gestion du système de fichiers (créer/supprimer/parcourir les répertoires)
- gestion de la mémoire (allouer, libérer, partager)
- gestion des processus (créer, terminer, arrêter, attendre, signaler, ...)

# Rappel sur le SE/OS

Le SE propose des primitives (fonctions) utilisables par des programmes et permettant l'utilisation des ressources :

- gestion des fichiers (ouvrir, fermer, lire, écrire, ...)
- gestion du système de fichiers (créer/supprimer/parcourir les répertoires)
- gestion de la mémoire (allouer, libérer, partager)
- gestion des processus (créer, terminer, arrêter, attendre, signaler, ...)
- gestion des communications entre processus (signaux, BAL, tubes, ...)

# Rappel sur le SE/OS

Le SE propose des primitives (fonctions) utilisables par des programmes et permettant l'utilisation des ressources :

- gestion des fichiers (ouvrir, fermer, lire, écrire, ...)
- gestion du système de fichiers (créer/supprimer/parcourir les répertoires)
- gestion de la mémoire (allouer, libérer, partager)
- gestion des processus (créer, terminer, arrêter, attendre, signaler, ...)
- gestion des communications entre processus (signaux, BAL, tubes, ...)
- gestion des E/S (écrans, souris, clavier, ...)

# Rappel sur le SE/OS

Le SE propose des primitives (fonctions) utilisables par des programmes et permettant l'utilisation des ressources :

- gestion des fichiers (ouvrir, fermer, lire, écrire, ...)
- gestion du système de fichiers (créer/supprimer/parcourir les répertoires)
- gestion de la mémoire (allouer, libérer, partager)
- gestion des processus (créer, terminer, arrêter, attendre, signaler, ...)
- gestion des communications entre processus (signaux, BAL, tubes, ...)
- gestion des E/S (écrans, souris, clavier, ...)
- gestion du réseau (sockets)

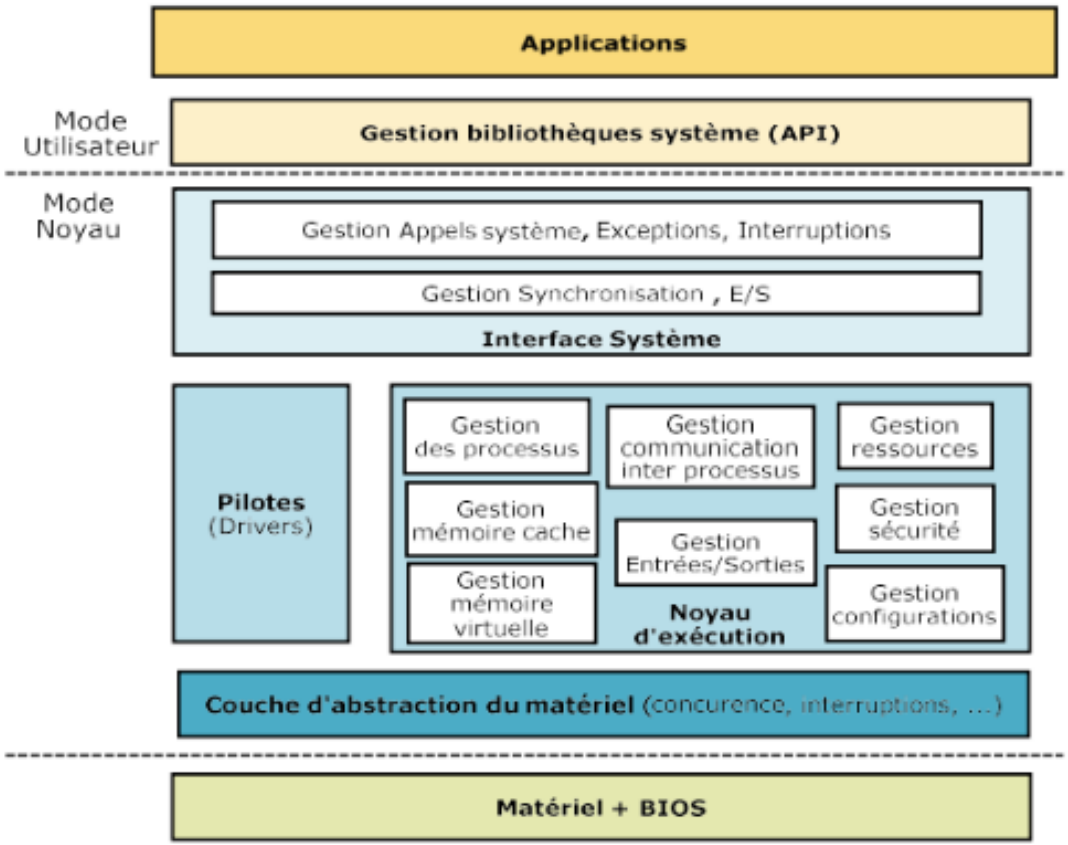
# Rappel sur le SE/OS

Le SE propose des primitives (fonctions) utilisables par des programmes et permettant l'utilisation des ressources :

- Pour UNIX il existe la norme POSIX (ISO 9945-1) qui définit les appels standards
- Pour Windows c'est l'API win32s et win32 pour 64 bits

# Rappel sur le SE/OS

Le SE a une structure en couches.



# Rappel sur les processus

Processus ?

# Rappel sur les processus

Processus ?

- Un processus est une exécution de programme.

# Rappel sur les processus

Processus ?

- Un processus est une exécution de programme.
- On peut imaginer le fonctionnement général :

# Rappel sur les processus

Processus ?

- Un processus est une exécution de programme.
- On peut imaginer le fonctionnement général :
  - CPU = pâtissier

# Rappel sur les processus

Processus ?

- Un processus est une exécution de programme.
- On peut imaginer le fonctionnement général :
  - CPU = pâtissier
  - programme = recette de pâtisserie

# Rappel sur les processus

Processus ?

- Un processus est une exécution de programme.
- On peut imaginer le fonctionnement général :
  - CPU = pâtissier
  - programme = recette de pâtisserie
  - entrées = ingrédients

# Rappel sur les processus

Processus ?

- Un processus est une exécution de programme.
- On peut imaginer le fonctionnement général :
  - CPU = pâtissier
  - programme = recette de pâtisserie
  - entrées = ingrédients
  - sorties = la pâtisserie

# Rappel sur les processus

Processus ?

- Un processus est une exécution de programme.
- On peut imaginer le fonctionnement général :
  - CPU = pâtissier
  - programme = recette de pâtisserie
  - entrées = ingrédients
  - sorties = la pâtisserie
  - processus = l'action de lire la recette et de préparer la pâtisserie

# Rappel sur les processus

- Un ordinateur a la possibilité de réaliser plusieurs actions en parallèle comme par exemple des exécutions et des E/S.

# Rappel sur les processus

- Un ordinateur a la possibilité de réaliser plusieurs actions en parallèle comme par exemple des exécutions et des E/S.
- Si on a plusieurs processeurs, on peut exécuter plusieurs processus en parallèle sinon chaque processus s'exécute l'un après l'autre sur l'unique processeur.

# Rappel sur les processus

Objectifs des processus :

# Rappel sur les processus

Objectifs des processus :

- Séparer les différentes tâches du système

# Rappel sur les processus

Objectifs des processus :

- Séparer les différentes tâches du système
- Gestion des fichiers et des applications

# Rappel sur les processus

Objectifs des processus :

- Séparer les différentes tâches du système
- Gestion des fichiers et des applications
- Permettre le multitâche

# Rappel sur les processus

Objectifs des processus :

- Séparer les différentes tâches du système
- Gestion des fichiers et des applications
- Permettre le multitâche

Attention ! ! ! Chaque processus doit tenir comptes des autres.

# Rappel sur les processus

Principe :

# Rappel sur les processus

Principe :

- un processeur gère plusieurs processus en attente et donne du temps d'exécution à chacun d'entre eux

# Rappel sur les processus

Principe :

- un processeur gère plusieurs processus en attente et donne du temps d'exécution à chacun d'entre eux
- un processus a un contexte qui lui permet de reprendre son exécution lorsque le processeur lui donne du temps d'exécution.

# Rappel sur les processus

- La commutation de processus va sauvegarder le contexte d'un processus pour restaurer/reprendre (à la place) le contexte d'un processus précédemment interrompu dans le cadre d'un partage des ressources d'un processeur.

# Rappel sur les processus

- La commutation de processus va sauvegarder le contexte d'un processus pour restaurer/reprendre (à la place) le contexte d'un processus précédemment interrompu dans le cadre d'un partage des ressources d'un processeur.
  - P1 s'exécute, on stoppe P1 et on mémorise le contexte de P1,

# Rappel sur les processus

- La commutation de processus va sauvegarder le contexte d'un processus pour restaurer/reprendre (à la place) le contexte d'un processus précédemment interrompu dans le cadre d'un partage des ressources d'un processeur.
  - P1 s'exécute, on stoppe P1 et on mémorise le contexte de P1,
  - P2 s'exécute, on stoppe P2 et on mémorise le contexte de P2,

# Rappel sur les processus

- La commutation de processus va sauvegarder le contexte d'un processus pour restaurer/reprendre (à la place) le contexte d'un processus précédemment interrompu dans le cadre d'un partage des ressources d'un processeur.
  - P1 s'exécute, on stoppe P1 et on mémorise le contexte de P1,
  - P2 s'exécute, on stoppe P2 et on mémorise le contexte de P2,
  - On restaure le contexte de P1 et on refait 1) à partir de l'endroit où P1 a été stoppé

# Rappel sur les processus

- La commutation de processus va sauvegarder le contexte d'un processus pour restaurer/reprendre (à la place) le contexte d'un processus précédemment interrompu dans le cadre d'un partage des ressources d'un processeur.
  - P1 s'exécute, on stoppe P1 et on mémorise le contexte de P1,
  - P2 s'exécute, on stoppe P2 et on mémorise le contexte de P2,
  - On restaure le contexte de P1 et on refait 1) à partir de l'endroit où P1 a été stoppé
  - On restaure le contexte de P2 et on refait 2) à partir de l'endroit où P2 a été stoppé

# Rappel sur les processus

- Un **programme** construit lors d'une compilation est un objet inerte correspondant au contenu d'un fichier sur disque.

# Rappel sur les processus

- Un **programme** construit lors d'une compilation est un objet inerte correspondant au contenu d'un fichier sur disque.
- Un **processus** est un objet **dynamique** correspondant à l'exécution des instructions d'un programme. C'est l'entité d'exécution dans le système Linux.

# Rappel sur les processus

- Un **programme** construit lors d'une compilation est un objet inerte correspondant au contenu d'un fichier sur disque.
- Un **processus** est un objet **dynamique** correspondant à l'exécution des instructions d'un programme. C'est l'entité d'exécution dans le système Linux.
- Il a besoin de **ressources** pour s'exécuter (UC, mémoire, unités E/S...)

# Rappel sur les processus

- Un **programme** construit lors d'une compilation est un objet inerte correspondant au contenu d'un fichier sur disque.
- Un **processus** est un objet **dynamique** correspondant à l'exécution des instructions d'un programme. C'est l'entité d'exécution dans le système Linux.
- Il a besoin de **ressources** pour s'exécuter (UC, mémoire, unités E/S...)
- Dans linux il existe deux types de processus :

# Rappel sur les processus

- Un **programme** construit lors d'une compilation est un objet inerte correspondant au contenu d'un fichier sur disque.
- Un **processus** est un objet **dynamique** correspondant à l'exécution des instructions d'un programme. C'est l'entité d'exécution dans le système Linux.
- Il a besoin de **ressources** pour s'exécuter (UC, mémoire, unités E/S...)
- Dans linux il existe deux types de processus :
  - Processus *système*

# Rappel sur les processus

- Un **programme** construit lors d'une compilation est un objet inerte correspondant au contenu d'un fichier sur disque.
- Un **processus** est un objet **dynamique** correspondant à l'exécution des instructions d'un programme. C'est l'entité d'exécution dans le système Linux.
- Il a besoin de **ressources** pour s'exécuter (UC, mémoire, unités E/S...)
- Dans linux il existe deux types de processus :
  - Processus *système*
    - ★ swapper
    - ★ cron
    - ★ getty

# Rappel sur les processus

- Un **programme** construit lors d'une compilation est un objet inerte correspondant au contenu d'un fichier sur disque.
- Un **processus** est un objet **dynamique** correspondant à l'exécution des instructions d'un programme. C'est l'entité d'exécution dans le système Linux.
- Il a besoin de **ressources** pour s'exécuter (UC, mémoire, unités E/S...)
- Dans linux il existe deux types de processus :
  - Processus *système*
    - ★ swapper
    - ★ cron
    - ★ getty
  - Processus *utilisateur* qui correspondent à l'exécution :

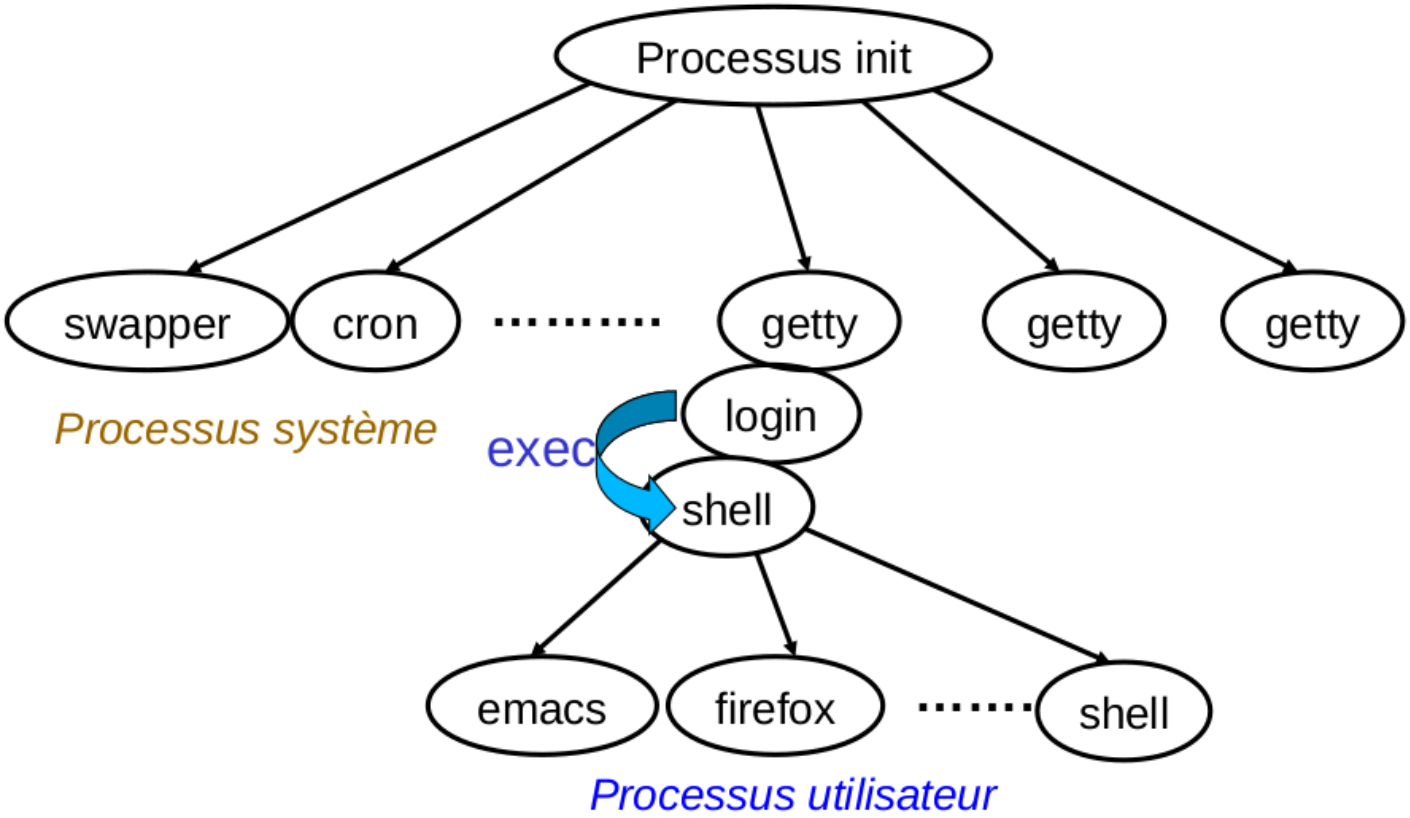
# Rappel sur les processus

- Un **programme** construit lors d'une compilation est un objet inerte correspondant au contenu d'un fichier sur disque.
- Un **processus** est un objet **dynamique** correspondant à l'exécution des instructions d'un programme. C'est l'entité d'exécution dans le système Linux.
- Il a besoin de **ressources** pour s'exécuter (UC, mémoire, unités E/S...)
- Dans linux il existe deux types de processus :
  - Processus *système*
    - ★ swapper
    - ★ cron
    - ★ getty
  - Processus *utilisateur* qui correspondent à l'exécution :
    - ★ d'une commande

# Rappel sur les processus

- Un **programme** construit lors d'une compilation est un objet inerte correspondant au contenu d'un fichier sur disque.
- Un **processus** est un objet **dynamique** correspondant à l'exécution des instructions d'un programme. C'est l'entité d'exécution dans le système Linux.
- Il a besoin de **ressources** pour s'exécuter (UC, mémoire, unités E/S...)
- Dans linux il existe deux types de processus :
  - Processus *système*
    - ★ swapper
    - ★ cron
    - ★ getty
  - Processus *utilisateur* qui correspondent à l'exécution :
    - ★ d'une commande
    - ★ d'une application

# Arborescence des processus dans Linux




Les commandes shell :

- `pstree` : affiche l'arborescence des processus
- `ps` (Processus Status) : donne la liste des processus actifs selon certains critères.

# Duplication des processus

- Chaque processus est identifié par un numéro unique (PID).

# Duplication des processus

- Chaque processus est identifié par un numéro unique (PID).
- Lors de l'initialisation du système linux, un premier processus (*init*) est créé avec un PID = 1.

# Duplication des processus

- Chaque processus est identifié par un numéro unique (PID).
- Lors de l'initialisation du système linux, un premier processus (*init*) est créé avec un PID = 1.
- La création d'un processus se fait par **duplication**. Un processus peut demander au système sa duplication en utilisant la primitive `fork()`.

# Duplication des processus

- Chaque processus est identifié par un numéro unique (PID).
- Lors de l'initialisation du système linux, un premier processus (*init*) est créé avec un PID = 1.
- La création d'un processus se fait par **duplication**. Un processus peut demander au système sa duplication en utilisant la primitive `fork()`.
- Le système crée une copie exacte du processus original avec un PID différent.

# Duplication des processus

- Chaque processus est identifié par un numéro unique (PID).
- Lors de l'initialisation du système linux, un premier processus (*init*) est créé avec un PID = 1.
- La création d'un processus se fait par **duplication**. Un processus peut demander au système sa duplication en utilisant la primitive `fork()`.
- Le système crée une copie exacte du processus original avec un PID différent.
  - Père : processus créateur

# Duplication des processus

- Chaque processus est identifié par un numéro unique (PID).
- Lors de l'initialisation du système linux, un premier processus (*init*) est créé avec un PID = 1.
- La création d'un processus se fait par **duplication**. Un processus peut demander au système sa duplication en utilisant la primitive `fork()`.
- Le système crée une copie exacte du processus original avec un PID différent.
  - Père : processus créateur
  - Fils : processus créé.

# Duplication des processus

- Chaque processus est identifié par un numéro unique (PID).
- Lors de l'initialisation du système linux, un premier processus (*init*) est créé avec un PID = 1.
- La création d'un processus se fait par **duplication**. Un processus peut demander au système sa duplication en utilisant la primitive `fork()`.
- Le système crée une copie exacte du processus original avec un PID différent.
  - Père : processus créateur
  - Fils : processus créé.
- Le numéro du processus père est noté PPID.

# Duplication des processus

- Chaque processus est identifié par un numéro unique (PID).
- Lors de l'initialisation du système linux, un premier processus (*init*) est créé avec un PID = 1.
- La création d'un processus se fait par **duplication**. Un processus peut demander au système sa duplication en utilisant la primitive `fork()`.
- Le système crée une copie exacte du processus original avec un PID différent.
  - Père : processus créateur
  - Fils : processus créé.
- Le numéro du processus père est noté PPID.
- Le processus fils peut exécuter un nouveau code à l'aide des primitives EXEC (voir plus loin).

# Exécution d'une commande par un shell

- Le shell se duplique (`fork`) : il y aura 2 processus shell identiques.

# Exécution d'une commande par un shell

- Le shell se duplique (`fork`) : il y aura 2 processus shell identiques.
- Le père se met en attente de la fin du fils (`wait`).

# Exécution d'une commande par un shell

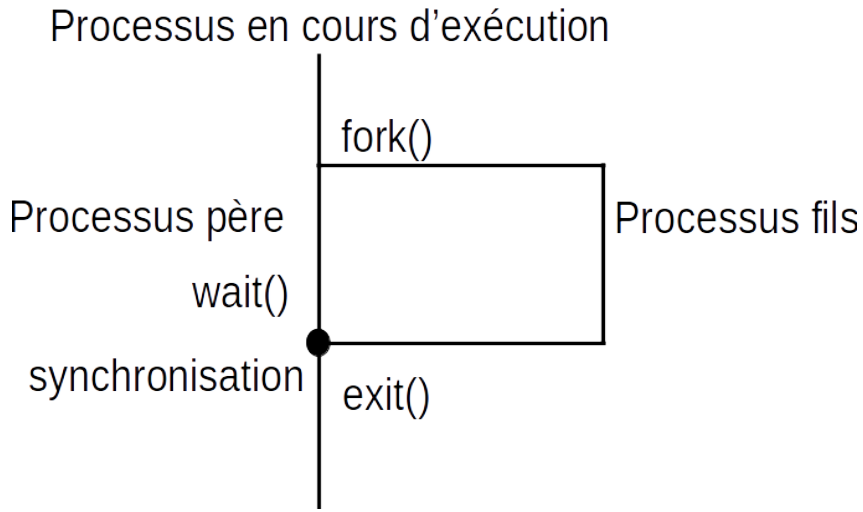
- Le shell se duplique (`fork`) : il y aura 2 processus shell identiques.
- Le père se met en attente de la fin du fils (`wait`).
- Le shell fils remplace son exécutable par celui de la commande à exécuter. Par exemple, `ls -l`.

# Exécution d'une commande par un shell

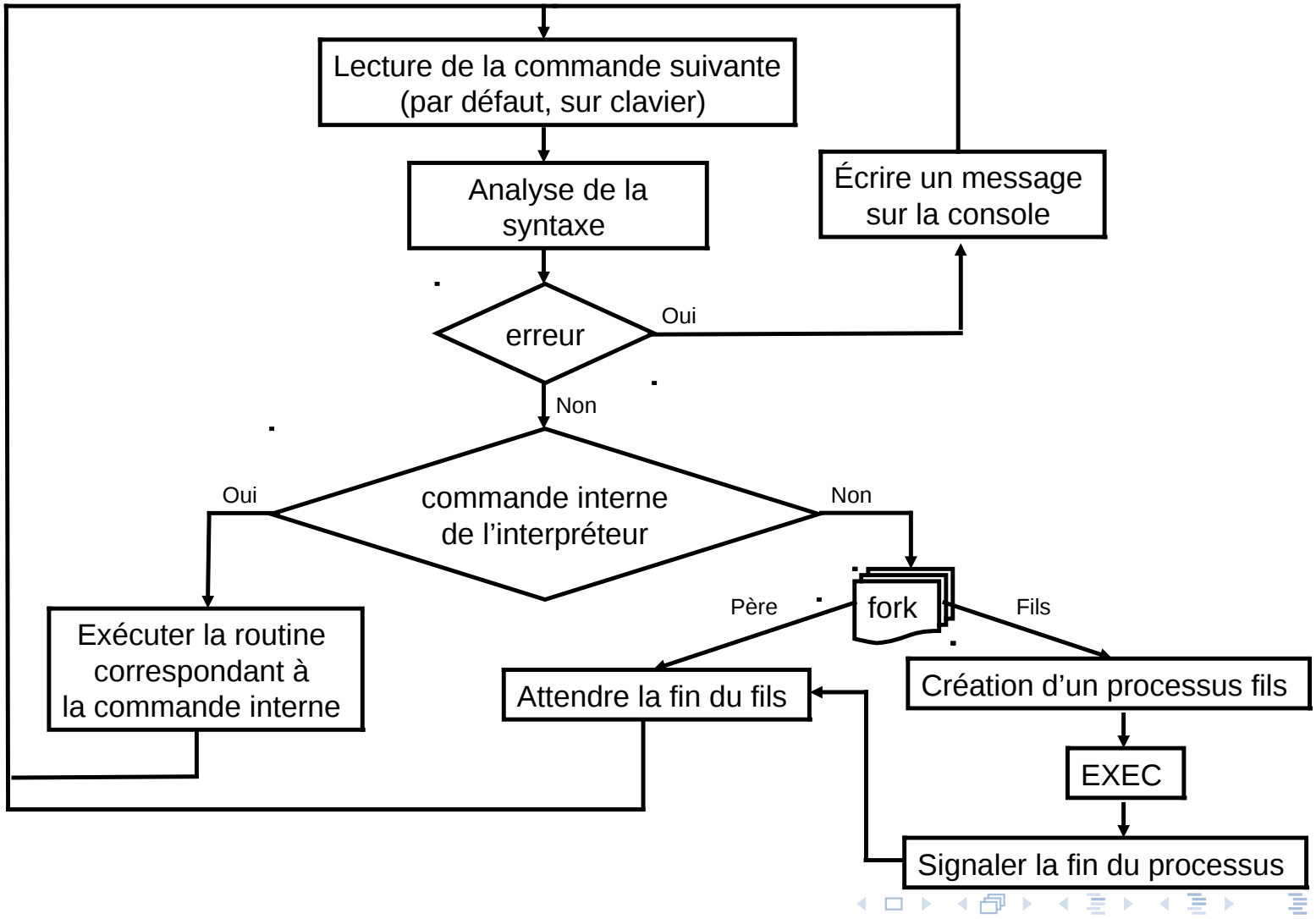
- Le shell se duplique (`fork`) : il y aura 2 processus shell identiques.
- Le père se met en attente de la fin du fils (`wait`).
- Le shell fils remplace son exécutable par celui de la commande à exécuter. Par exemple, `ls -l`
- La commande `ls -l` s'exécute. Lorsqu'elle termine, le processus fils disparaît.

# Exécution d'une commande par un shell

- Le shell se duplique (`fork`) : il y aura 2 processus shell identiques.
- Le père se met en attente de la fin du fils (`wait`).
- Le shell fils remplace son exécutable par celui de la commande à exécuter. Par exemple, `ls -l`
- La commande `ls -l` s'exécute. Lorsqu'elle termine, le processus fils disparaît.
- Le père est alors réactivé, et affiche le prompt suivant.

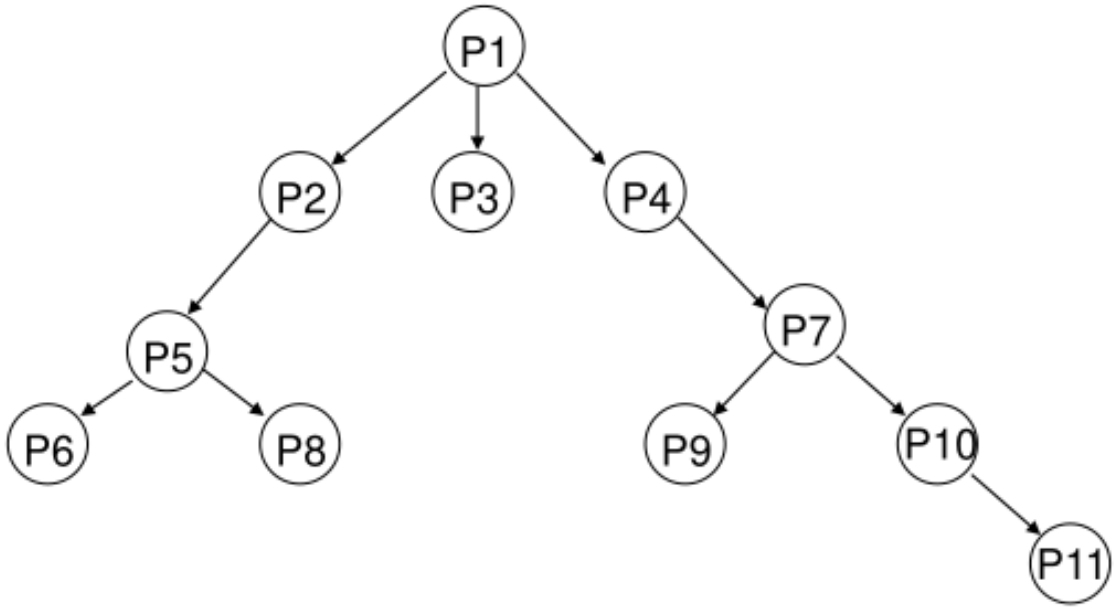


# Exécution d'une commande par un shell



# Hiérarchie des processus

Un processus avec tous ses descendants forment un groupe de processus représenté par un arbre de processus.

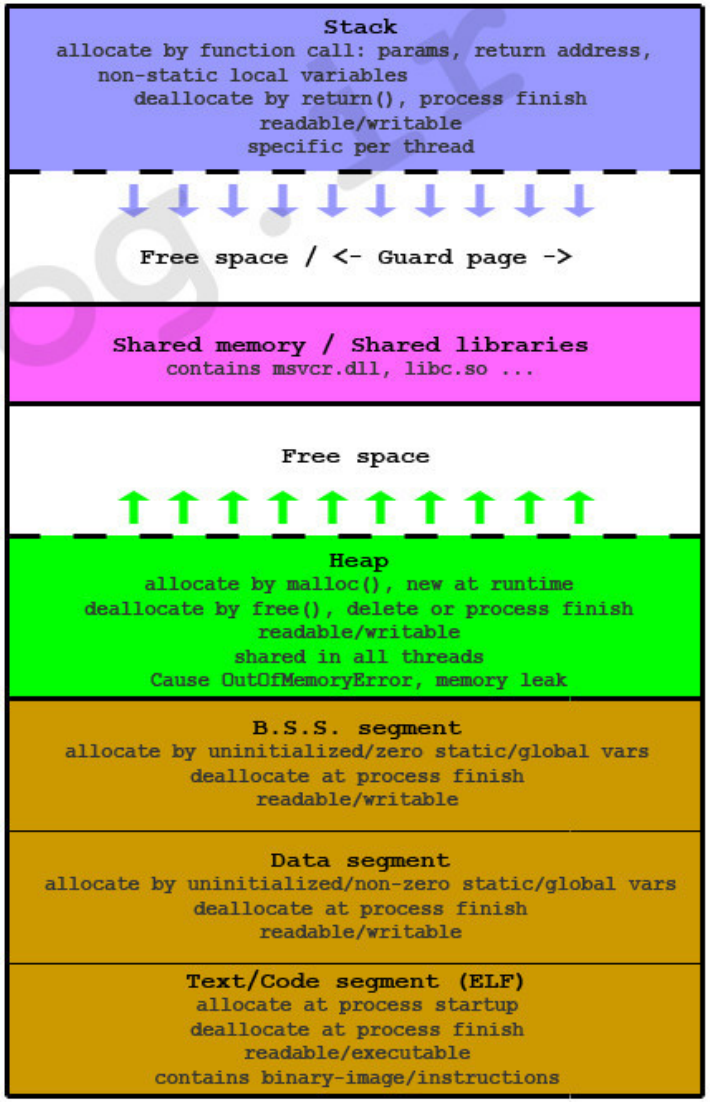


# Espace d'adressage d'un processus

Un processus Linux a un espace d'adressage constitué de 3 segments :

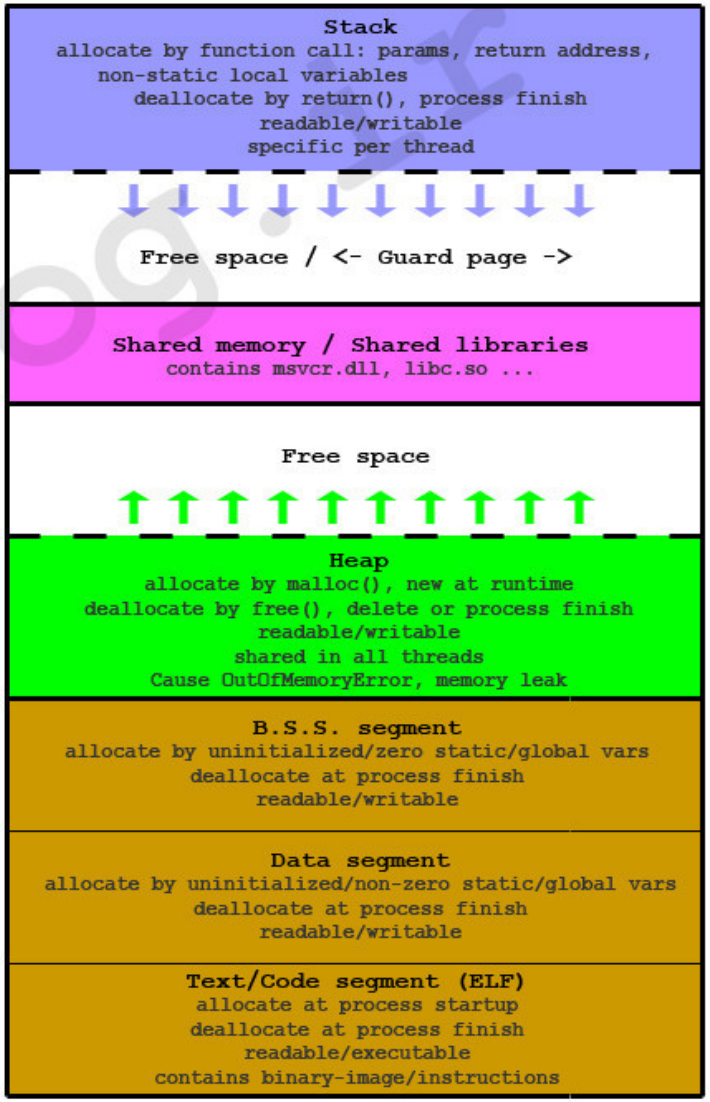
# Espace d'adressage d'un processus

Un processus Linux a un espace d'adressage constitué de 3 segments :



# Espace d'adressage d'un processus

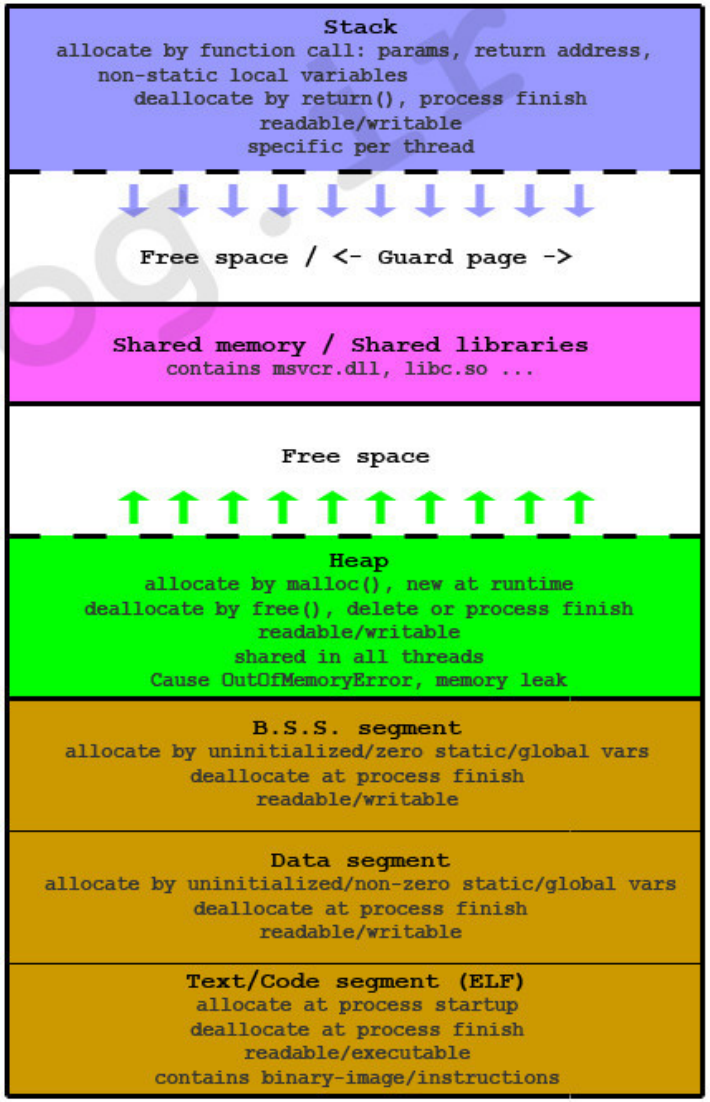
Un processus Linux a un espace d'adressage constitué de 3 segments :




- Le **code** (*Text/Code Segment*)correspond aux instructions, en langage d'assemblage, du programme à exécuter.

# Espace d'adressage d'un processus

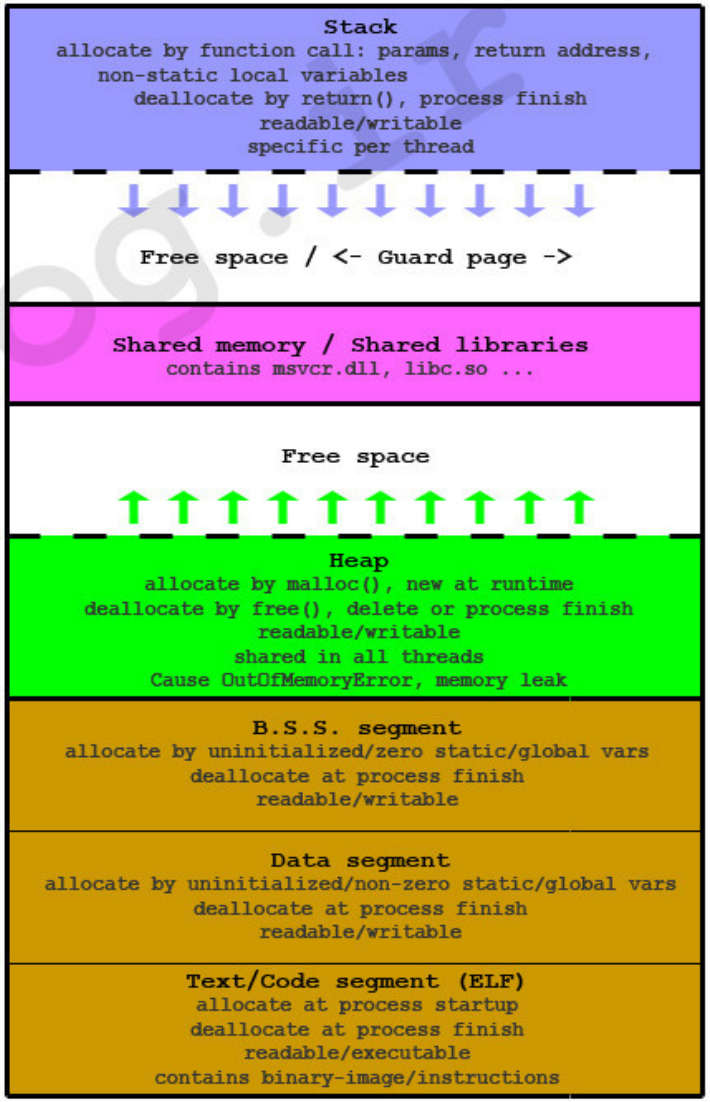
Un processus Linux a un espace d'adressage constitué de 3 segments :




- Le **code** (*Text/Code Segment*)correspond aux instructions, en langage d'assemblage, du programme à exécuter.
- La **zone de données** (*Data*, *B.S.S Segments* et *Heap*) contient les variables globales ou statiques du programme ainsi que les allocations dynamiques de mémoire.

# Espace d'adressage d'un processus

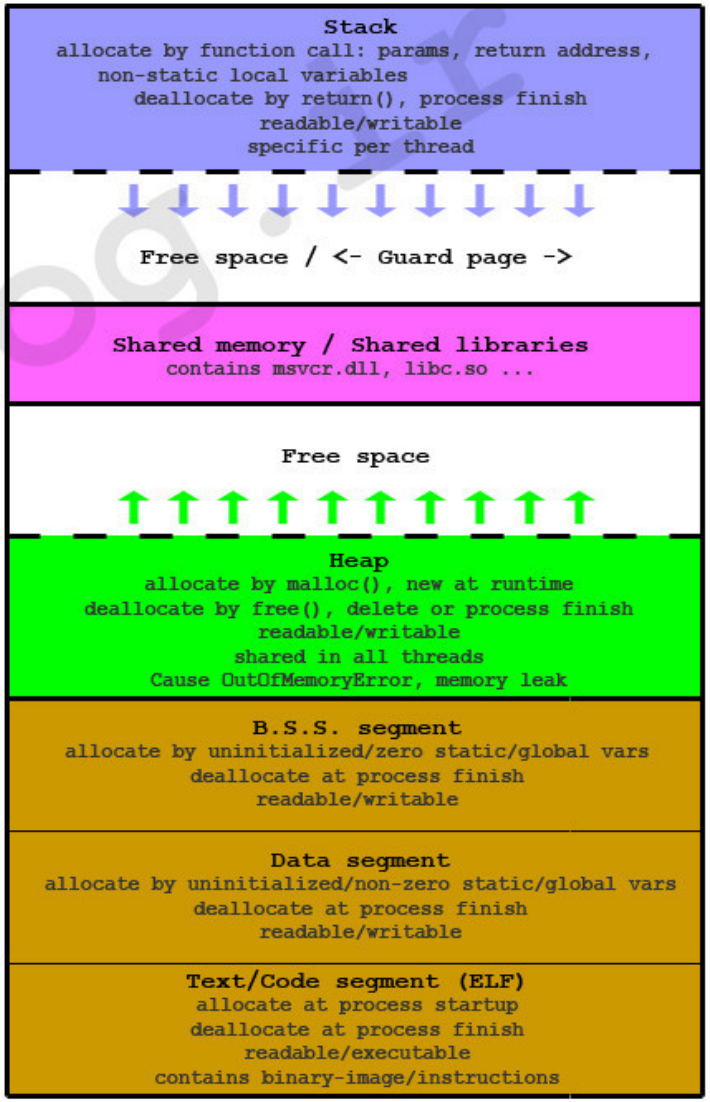
Un processus Linux a un espace d'adressage constitué de 3 segments :




- Le **code** (*Text/Code Segment*)correspond aux instructions, en langage d'assemblage, du programme à exécuter.
- La **zone de données** (*Data*, *B.S.S Segments* et *Heap*) contient les variables globales ou statiques du programme ainsi que les allocations dynamiques de mémoire.
- Enfin, les appels de fonctions, avec leurs paramètres et leurs variables locales, viennent s'empiler sur la **pile** (*Stack*).

# Espace d'adressage d'un processus

Un processus Linux a un espace d'adressage constitué de 3 segments :




- Le **code** (*Text/Code Segment*)correspond aux instructions, en langage d'assemblage, du programme à exécuter.
- La **zone de données** (*Data, B.S.S Segments* et *Heap*) contient les variables globales ou statiques du programme ainsi que les allocations dynamiques de mémoire.
- Enfin, les appels de fonctions, avec leurs paramètres et leurs variables locales, viennent s'empiler sur la **pile** (*Stack*).

## Contexte d'un processus :

- Le contenu de son espace d'adressage
- Le contenu des registres matériels
- Le compteur ordinal
- Les variables
- Les structure de données du noyaux qui ont un rapport avec les processus

# Cycle de vie d'un processus

1. Appel système `fork` par le père. Le processus père continue son exécution en arrière plan.

# Cycle de vie d'un processus

1. Appel système `fork` par le père. Le processus père continue son exécution en arrière plan.
2. Appel système `exec` par le processus fils.

# Cycle de vie d'un processus

1. Appel système `fork` par le père. Le processus père continue son exécution en arrière plan.
2. Appel système `exec` par le processus fils.
3. Déroulement du programme.

# Cycle de vie d'un processus

1. Appel système `fork` par le père. Le processus père continue son exécution en arrière plan.
2. Appel système `exec` par le processus fils.
3. Déroulement du programme.
4. Fin du programme, envoi du code retour (`exit`).

# Cycle de vie d'un processus

1. Appel système `fork` par le père. Le processus père continue son exécution en arrière plan.
2. Appel système `exec` par le processus fils.
3. Déroulement du programme.
4. Fin du programme, envoi du code retour (`exit`).
5. Récupération du processus fils à l'état zombie par le processus père.

# Cycle de vie d'un processus

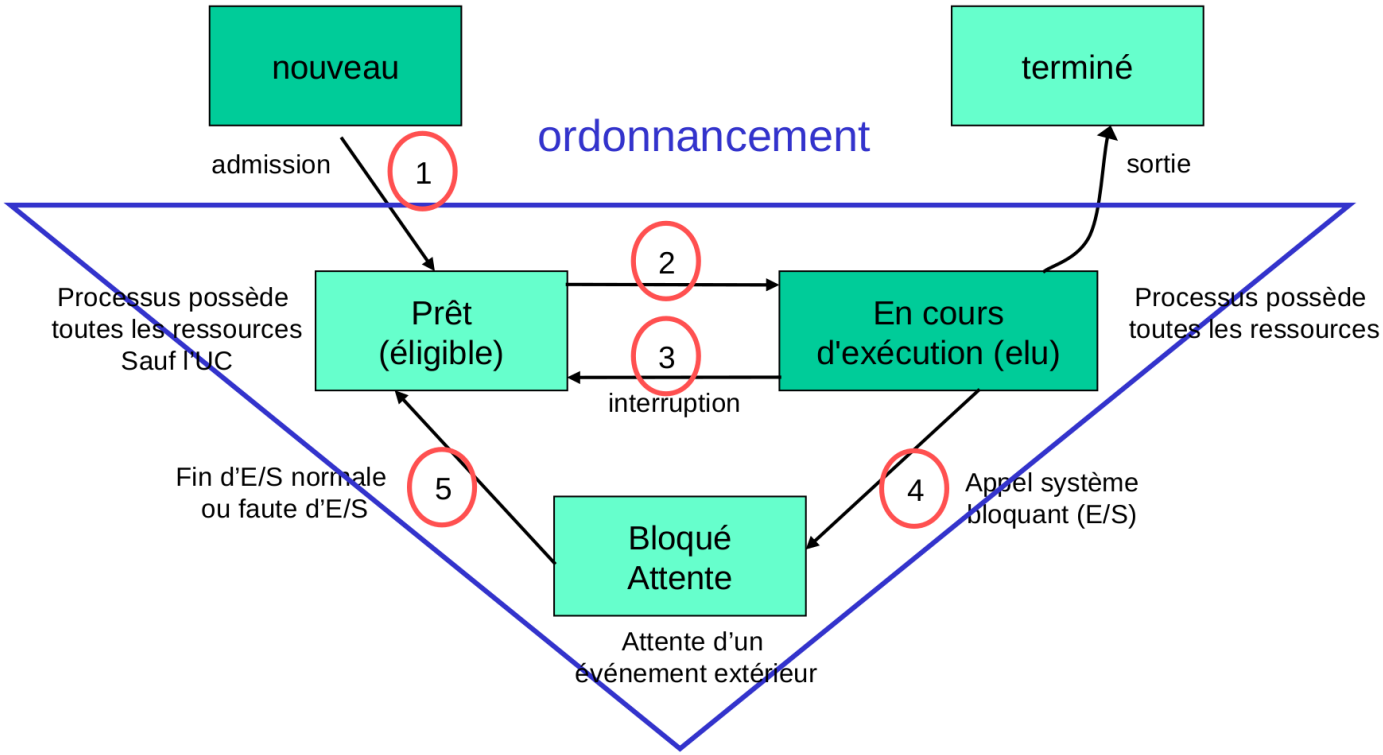
1. Appel système `fork` par le père. Le processus père continue son exécution en arrière plan.
2. Appel système `exec` par le processus fils.
3. Déroulement du programme.
4. Fin du programme, envoi du code retour (`exit`).
5. Récupération du processus fils à l'état zombie par le processus père.
6. Si le processus père a fini son exécution avant le processus fils, le processus fils à l'état zombie est "rattaché" au processus originel. Son PPID est alors le PID de `init`.

# Cycle de vie d'un processus

1. Appel système `fork` par le père. Le processus père continue son exécution en arrière plan.
2. Appel système `exec` par le processus fils.
3. Déroulement du programme.
4. Fin du programme, envoi du code retour (`exit`).
5. Récupération du processus fils à l'état zombie par le processus père.
6. Si le processus père a fini son exécution avant le processus fils, le processus fils à l'état zombie est "rattaché" au processus originel. Son PPID est alors le PID de `init`.
7. Tous processus Linux qui se terminent possèdent une valeur retour appelée *code de retour* (`exit status`) à laquelle le père peut accéder.

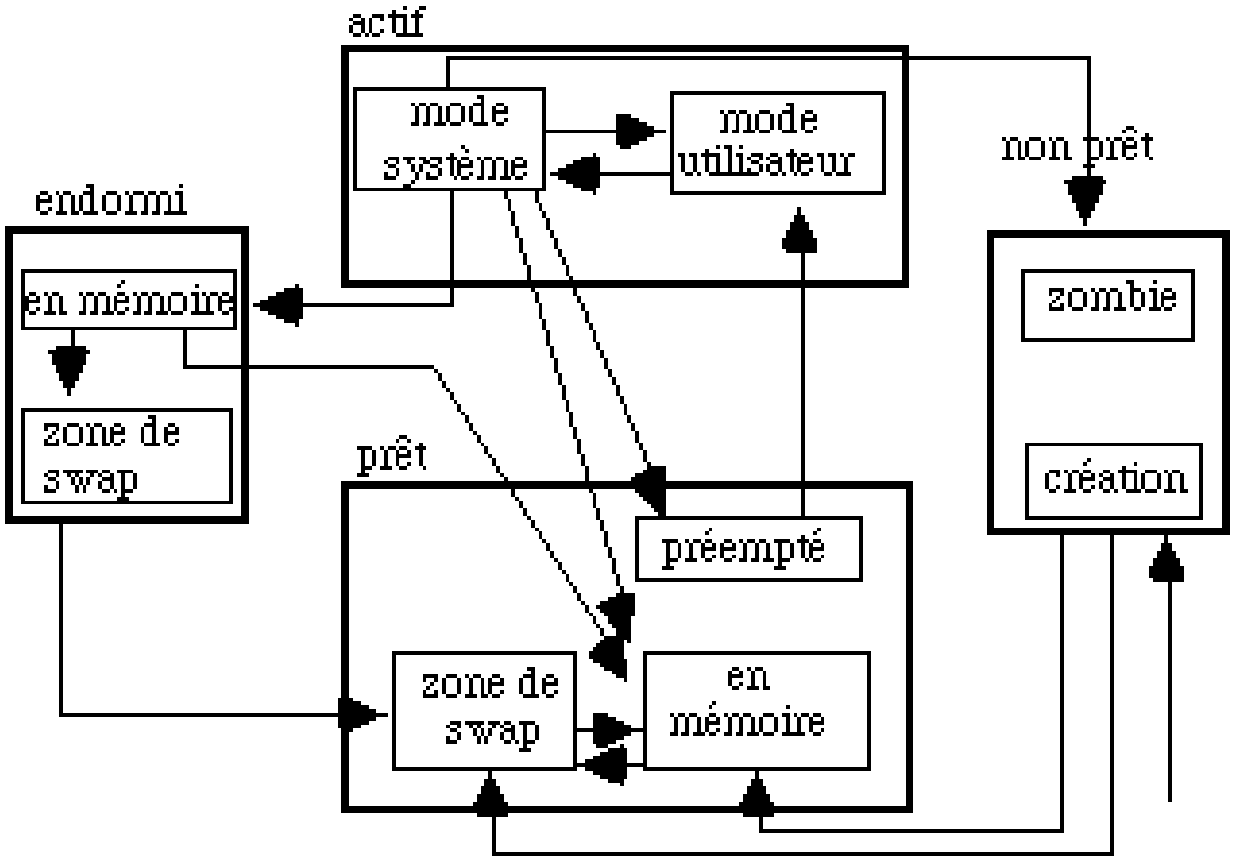
# États des processus

- Le processus peut être dans un certain nombre d'états connus : c'est un automate.
- L'ordonnanceur gère l'allocation du temps CPU aux processus.



# États des processus

- Le processus peut être dans un certain nombre d'états connus : c'est un automate.
- L'ordonnanceur gère l'allocation du temps CPU aux processus.



# Caractéristiques d'un processus

Cf. Table de processus (`ps`) et `/proc`

- PID et PPID
- Etat (*O/S/R/Z/T*)
- Priorité
- Zone de code exécutable
- Zone de données manipulées
- Répertoire courant
- Table des descripteurs (fichiers ouverts)
- Masque de création des fichiers(*umask*)
- Nombre maximal des fichiers que ce processus peut ouvrir par login (*ulimit*)
- Un état des registres
- Répertoire courant
- Pile d'exécution
- Propriétaire (*UID*) et son groupe (*GID*)
- Terminal d'attachement
- Le temps UC consommé

Multiprocessing

# Système Multi-taches (Multitasking)

**Multitasking :**

- Assurer de plusieurs programmes en même temps (c-à-d. plusieurs processus).

# Système Multi-taches (Multitasking)

**Multitasking :**

- Assurer de plusieurs programmes en même temps (c-à-d. plusieurs processus).
- Concentré sur des processus gérés par le même système d'exploitation, s'exécutant en parallèle, s'échangeant, partageant des données et en se synchronisant.

# Système Multi-taches (Multitasking)

**Multitasking :**

- Assurer de plusieurs programmes en même temps (c-à-d. plusieurs processus).
- Concentré sur des processus gérés par le même système d'exploitation, s'exécutant en parallèle, s'échangeant, partageant des données et en se synchronisant.
- Chaque processus a besoin du processeur

# Système Multi-taches (Multitasking)

**Multitasking :**

- Assurer de plusieurs programmes en même temps (c-à-d. plusieurs processus).
- Concentré sur des processus gérés par le même système d'exploitation, s'exécutant en parallèle, s'échangeant, partageant des données et en se synchronisant.
- Chaque processus a besoin du processeur
  - situation concurrente
  - solution : partage des ressources dans le temps = ordonnancement (*scheduling*)

# Système mono-processeur

Système avec un seul processeur

- quasi parallèle (multiplexage temporel)

# Système mono-processeur

Système avec un seul processeur

- quasi parallèle (multiplexage temporel)
- arrêter et reprendre les différents processus
  $\rightarrow$ gestion avec le *scheduler* (ordonnancement des processus)

# Système mono-processeur

**Exemple :**

# Système mono-processeur

**Exemple :**

- Multiprogrammation de quatre programmes

# Système mono-processeur

**Exemple :**

- Multiprogrammation de quatre programmes
- Concept : 4 processus séquentiels indépendants

# Système mono-processeur

**Exemple :**

- Multiprogrammation de quatre programmes
- Concept : 4 processus séquentiels indépendants
- À chaque instant, un seul processus est actif

# Système mono-processeur

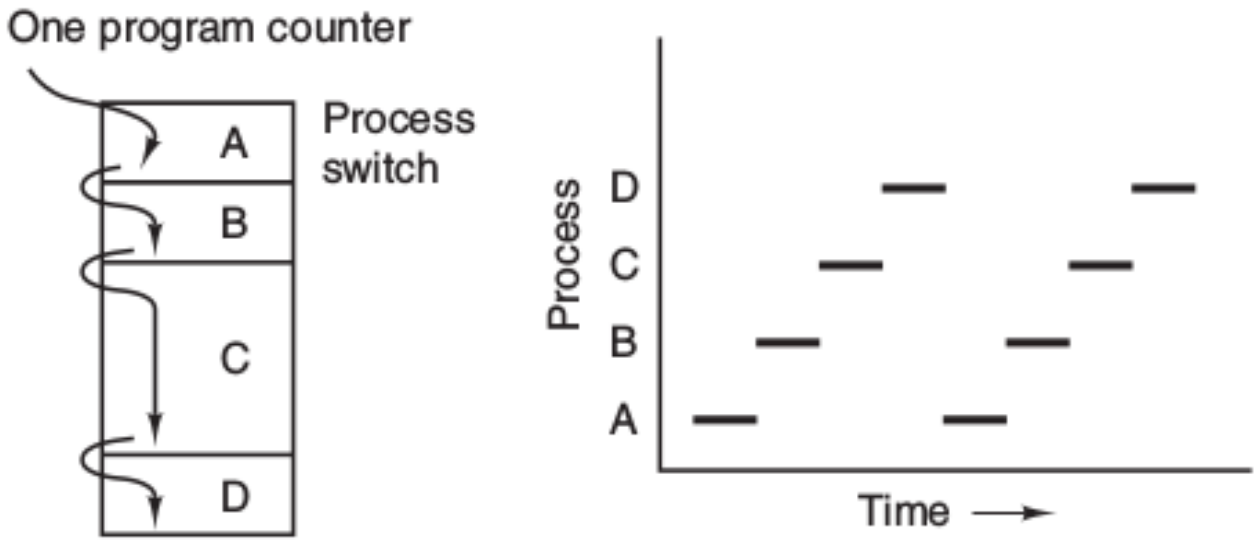
**Exemple :**

- Multiprogrammation de quatre programmes
- Concept : 4 processus séquentiels indépendants
- À chaque instant, un seul processus est actif
- Un compteur ordinal qui bascule entre processus

# Système mono-processeur

**Exemple :**

- Multiprogrammation de quatre programmes
- Concept : 4 processus séquentiels indépendants
- À chaque instant, un seul processus est actif
- Un compteur ordinal qui bascule entre processus



# Système multi-processeurs

- Système avec plusieurs processeurs

# Système multi-processeurs

- Système avec plusieurs processeurs
  - parallèle

# Système multi-processeurs

- Système avec plusieurs processeurs
  - parallèle
  - vrai multi-tâches

# Système multi-processeurs

- Système avec plusieurs processeurs
  - parallèle
  - vrai multi-tâches
  - doit assurer l'exécution d'autant de processus que de processeurs en même temps.

# Système multi-processeurs

- Système avec plusieurs processeurs
  - parallèle
  - vrai multi-tâches
  - doit assurer l'exécution d'autant de processus que de processeurs en même temps.
- Communications entre les processus et les processeurs

# Système multi-processeurs

**Exemple :**

# Système multi-processeurs

**Exemple :**

- Multiprogrammation de quatre programmes

# Système multi-processeurs

**Exemple :**

- Multiprogrammation de quatre programmes
- Concept : 4 processus séquentiels indépendants

# Système multi-processeurs

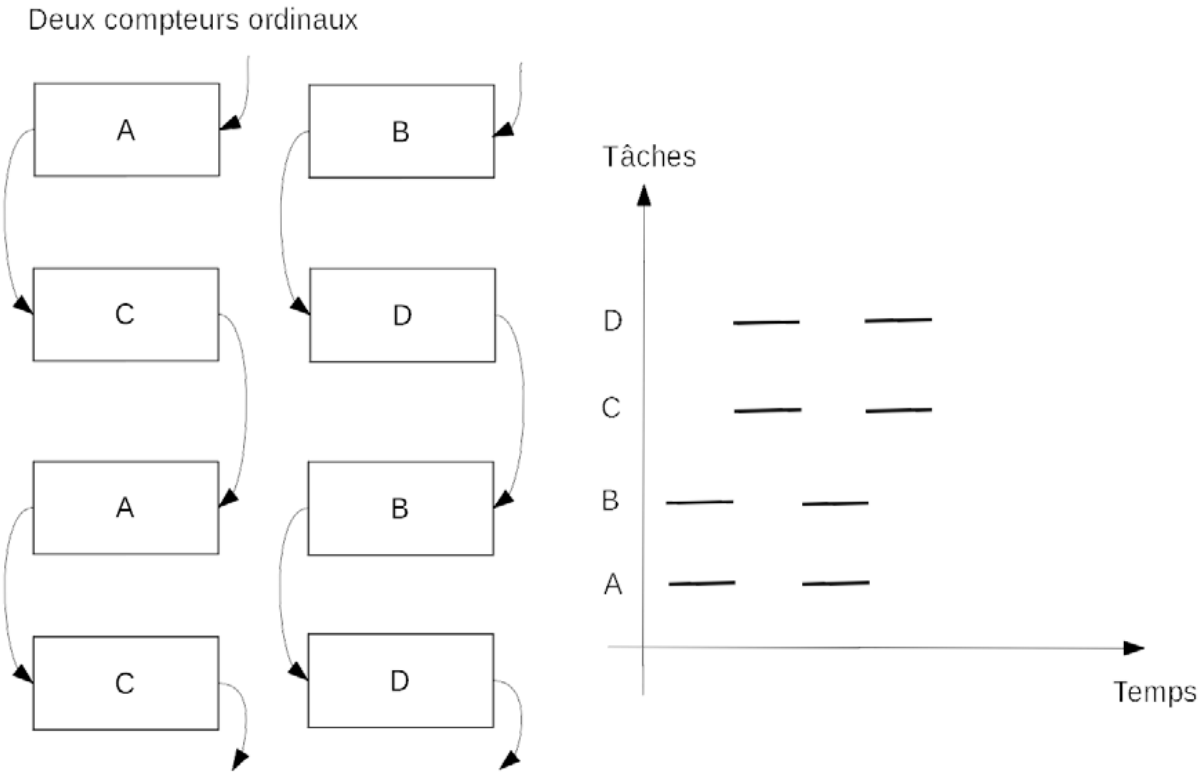
**Exemple :**

- Multiprogrammation de quatre programmes
- Concept : 4 processus séquentiels indépendants
- deux processus actifs (exemple double-cpu)

# Système multi-processeurs

**Exemple :**

- Multiprogrammation de quatre programmes
- Concept : 4 processus séquentiels indépendants
- deux processus actifs (exemple double-cpu)



# Avantages / Inconvénients du Multitasking

- Gain de temps de traitement

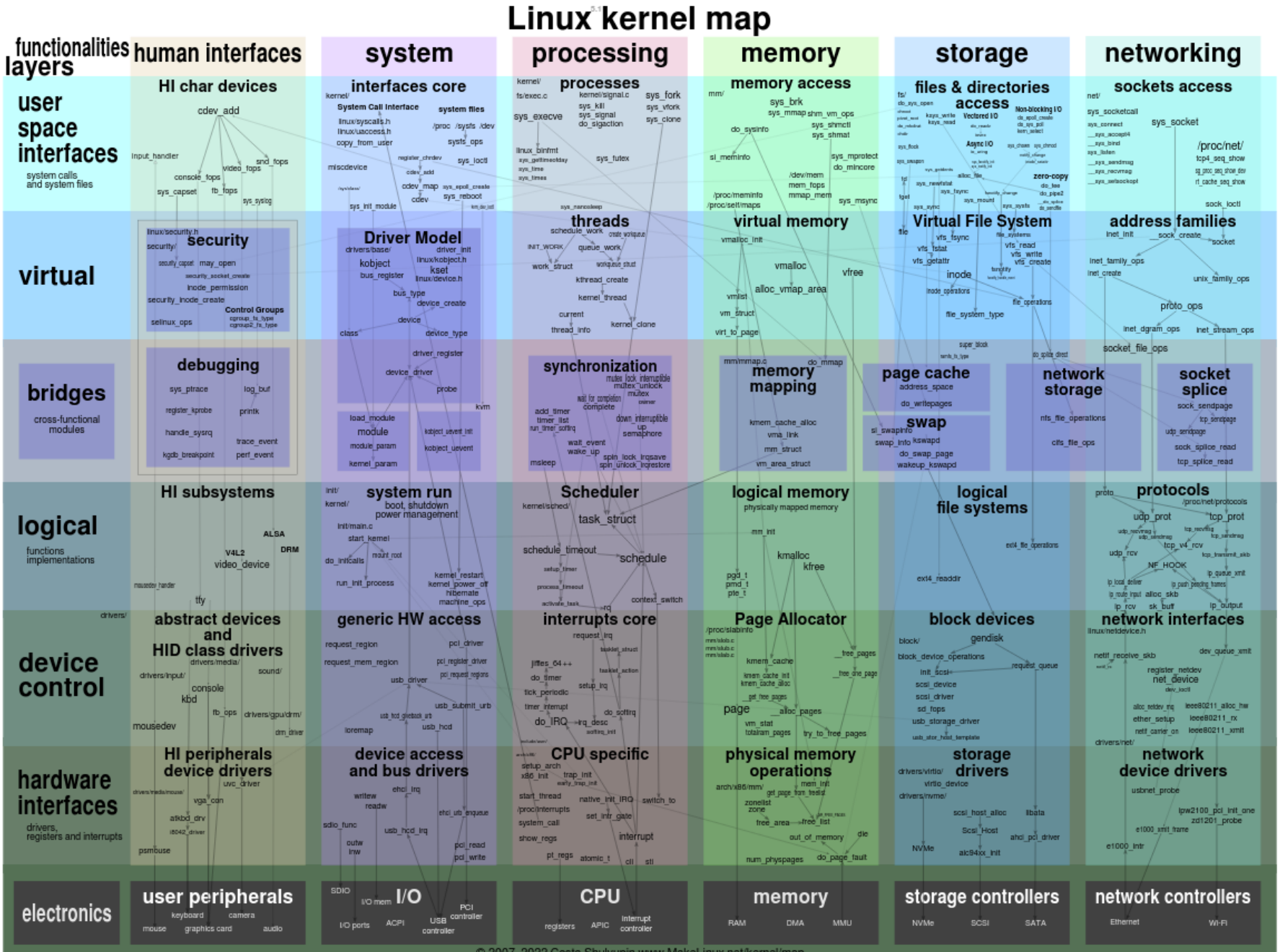
# Avantages / Inconvénients du Multitasking

- Gain de temps de traitement
- Meilleure utilisation d'une machine multiprocesseurs

# Avantages / Inconvénients du Multitasking

- Gain de temps de traitement
- Meilleure utilisation d'une machine multiprocesseurs
- Problèmes d'accès concurrents (synchronisation, interblocages, verrous...)

# Linux Kernel Map



# La primitive `fork()`

- L'appel de la primitive `fork()` demande au système d'effectuer une copie exacte du processus en cours d'exécution.

# La primitive `fork()`

- L'appel de la primitive `fork()` demande au système d'effectuer une copie exacte du processus en cours d'exécution.
- Si cette primitive a réussie, un nouveau processus est créé et exécute le même programme.

# La primitive `fork()`

- L'appel de la primitive `fork()` demande au système d'effectuer une copie exacte du processus en cours d'exécution.
- Si cette primitive a réussie, un nouveau processus est créé et exécute le même programme.
- Il hérite de la pile d'exécution et donc ce nouveau processus débute par le retour de la primitive `fork()`.

# La primitive `fork()`

- L'appel de la primitive `fork()` demande au système d'effectuer une copie exacte du processus en cours d'exécution.
- Si cette primitive a réussie, un nouveau processus est créé et exécute le même programme.
- Il hérite de la pile d'exécution et donc ce nouveau processus débute par le retour de la primitive `fork()`.
- Dans le processus père, `fork()` retourne le PID du processus fils crée.

# La primitive `fork()`

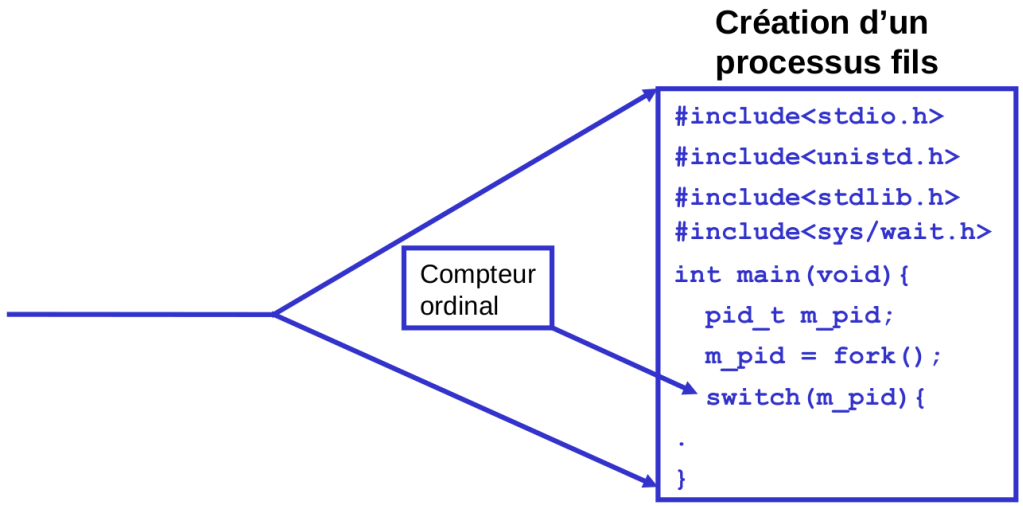
- L'appel de la primitive `fork()` demande au système d'effectuer une copie exacte du processus en cours d'exécution.
- Si cette primitive a réussie, un nouveau processus est créé et exécute le même programme.
- Il hérite de la pile d'exécution et donc ce nouveau processus débute par le retour de la primitive `fork()`.
- Dans le processus père, `fork()` retourne le PID du processus fils crée.
- Dans le processus fils, `fork()` retourne la valeur 0.

# La primitive `fork()`

- L'appel de la primitive `fork()` demande au système d'effectuer une copie exacte du processus en cours d'exécution.
- Si cette primitive a réussie, un nouveau processus est créé et exécute le même programme.
- Il hérite de la pile d'exécution et donc ce nouveau processus débute par le retour de la primitive `fork()`.
- Dans le processus père, `fork()` retourne le PID du processus fils crée.
- Dans le processus fils, `fork()` retourne la valeur 0.
- Si échec de fork(), alors pas de fils créé et `fork()` retourne -1.

# Création d'un nouveau processus

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

int main(void){
    int status;
    pid_t m_pid;
    m_pid = fork();

    switch(m_pid){
    case -1 :
            printf("Erreur: echec de fork
                \n");
    case 0 :
            printf("Processus fils : pid =
                % d\n", getpid())
            exit(0) // fin du processus fils
            break;
    default :
          printf("Pere : le fils a un PID :
              % d\n", m_pid);
          wait(&status); // attend fin du
              fils
}
```



# Héritage

Caractéristiques héritées par le processus fils :

- UID, identifiant du ou des propriétaires
- GID, identifiant du groupe
- Toutes les valeurs des variables
- Les descripteurs des fichiers ouverts

Caractéristiques non héritées :

- PID
- PPID
- Temps d'exécution (initialisés à zéro)
- Les signaux pendants
- La priorité d'exécution si elle a été modifiée
- Les verrous sur fichiers

**Obtention des informations d'un processus :**

`#include <unistd.h>`

`getpid();` retourne le PID du processus appelant

`getppid();` retourne le PID du père de processus

`char * getcwd(char * buf, size_t taille);` La référence absolue du repertoire de travail d'un processus peut être obtenue dans la chaine `buf` de taille `taille`

# Héritage

**Cas spécifique des descripteurs de fichiers :**

- Après un fork, les entrées de la table des descripteurs pointent sur la même entrée de la table des fichiers ouverts.
- Le nombre de descripteurs de la table des fichiers ouverts est incrémenté.
- L'offset est commun aux deux processus.

# Fin de vie d'un processus

- Rappel : tout processus passe en état zombie lorsque son exécution est terminée.

# Fin de vie d'un processus

- Rappel : tout processus passe en état zombie lorsque son exécution est terminée.
- Un processus zombie occupe une entrée dans la table des processus (le nombre est limité !).

# Fin de vie d'un processus

- Rappel : tout processus passe en état zombie lorsque son exécution est terminée.
- Un processus zombie occupe une entrée dans la table des processus (le nombre est limité !).
- Le processus père peut alors accéder aux informations relatives à cette terminaison.

# Fin de vie d'un processus

- Sortie (a)normale (volontaire)
  - Unix / Linux : `exit()`
    ```
    #include <stdlib.h>
    void exit(int status)
    ```
    L'argument *status* est un entier qui indique au shell (ou au père de façon générale) qu'une erreur s'est produite. On laisse à zéro pour indiquer une fin normale.
- Erreur fatale (involontaire)
- Bugs : Division par 0, accès mémoire illégal etc..
- Possibilité de paramétrer le comportement (signaux UNIX).
- Tué par un autre processus (involontaire).
  - Terminaison normale involontaire UNIX : kill.

# Processus zombie

- L'exécution asynchrone entre processus parent et fils a certaines conséquences.

# Processus zombie

- L'exécution asynchrone entre processus parent et fils a certaines conséquences.
- Souvent, le fils d'un processus se termine, mais son père ne l'attend pas.

# Processus zombie

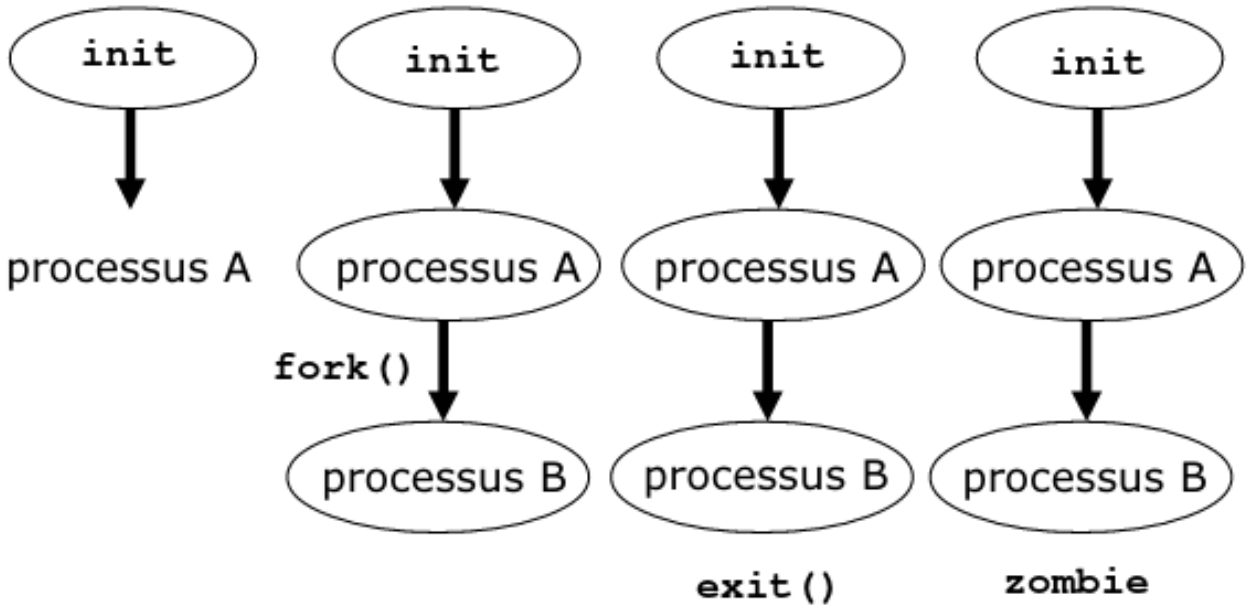
- L'exécution asynchrone entre processus parent et fils a certaines conséquences.
- Souvent, le fils d'un processus se termine, mais son père ne l'attend pas.
- Le processus fils devient alors un processus zombie.

# Processus zombie

- L'exécution asynchrone entre processus parent et fils a certaines conséquences.
- Souvent, le fils d'un processus se termine, mais son père ne l'attend pas.
- Le processus fils devient alors un processus zombie.
- Le processus fils existe toujours dans la table des processus, mais il n'utilise plus les ressources du kernel.

# Processus zombie

- L'exécution asynchrone entre processus parent et fils a certaines conséquences.
- Souvent, le fils d'un processus se termine, mais son père ne l'attend pas.
- Le processus fils devient alors un processus zombie.
- Le processus fils existe toujours dans la table des processus, mais il n'utilise plus les ressources du kernel.



# Processus zombie

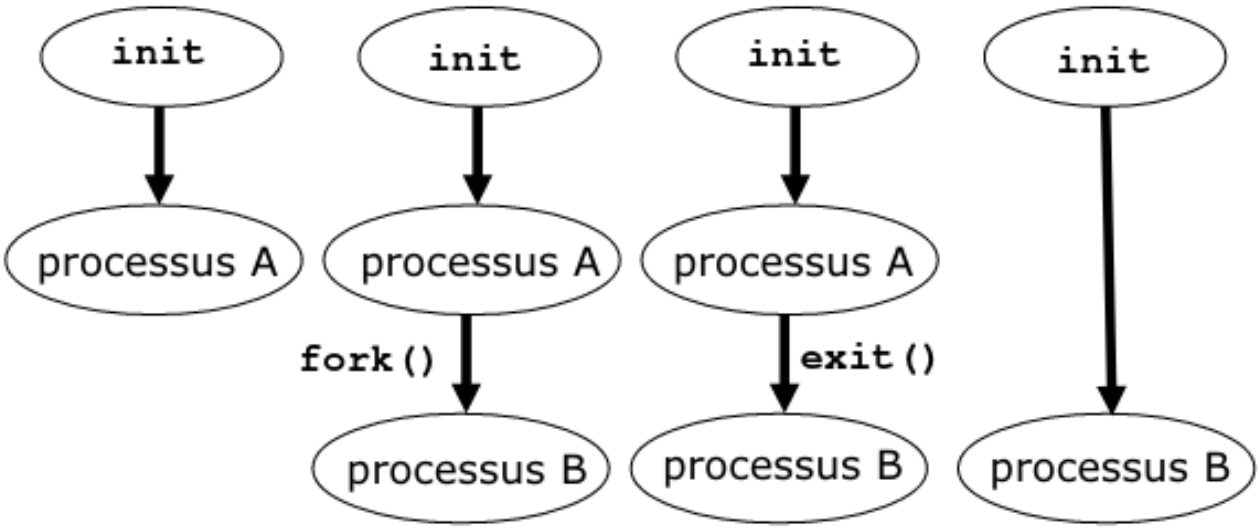
- Il peut y avoir aussi des terminaisons prématurées, où le processus parent se termine ses processus fils.

# Processus zombie

- Il peut y avoir aussi des terminaisons prématurées, où le processus parent se termine ses processus fils.
- Dans cette situation, ses processus fils sont adoptés par le processus init, dont le pid = 1.

# Processus zombie

- Il peut y avoir aussi des terminaisons prématurées, où le processus parent se termine ses processus fils.
- Dans cette situation, ses processus fils sont adoptés par le processus init, dont le pid = 1.



**Primitive wait :**

- La primitive `wait()` permet de récupérer les informations de terminaison et de supprimer les processus zombie.
  - Si l'appelant possède au moins un fils non zombie l'appel est bloquant
  - Si l'appelant ne possède aucun fils (ni en exécution ni zombie) le retour est immédiat et vaut -1

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int * status);
```

- Le retour est le PID du processus fils qui a été récupéré.
- *status* contient la valeur de retour retourné par le fils.

# Synchronisation

**Primitive waitpid :**

- Permet de sélectionner un processus fils de pid particulier.

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int * status, int option);
```

- Le paramètre **pid** permet de sélectionner le processus attendu de la manière suivante :
  - $< -1$ père suspendu jusqu'à la fin de n'importe lequel de ses fils dont = GID
  - -1 père suspendu jusqu'à la fin de n'importe lequel de ses fils
  - 0 père suspendu jusqu'à la fin de n'importe lequel de ses fils de son groupe
  - $> 0$ père suspendu jusqu'à la fin d'un fils d'identité pid

# Synchronisation

**Primitive waitpid :**

- Le paramètre **option** permet entre autre de choisir entre les modes
  - 0 (WUNTRACED) bloquant : recevoir l'information concernant également les fils bloqués
  - 1 (WNOHANG) non-bloquant : ne pas bloquer si aucun fils ne s'est terminé
- La fonction renvoie :
  - -1 en cas d'erreur (en mode bloquant ou non bloquant) ;
  - 0 en cas d'échec (processus demandé existant mais ni terminé ni stoppé) en mode non bloquant ;
  - le pid du processus fils zombie pris en compte sinon.

## Information sur la terminaison

- Si *status* est non NULL, `wait` et `waitpid` y stockent l'information sur la terminaison du fils. Cette information peut être analysée avec les macros suivantes :
  - WIFEXITED (status) : vrai si terminaison normale du fils
  - WEXITSTATUS (status) : renvoie la valeur du fils, si WIFEXITED(status)
  - WIFSIGNALED (status) : vrai si terminaison du fils à cause d'un signal
  - WIFSTOPPED (status) : indique si le fils est actuellement arrêté.
  - WTERMSIG (status) : renvoie le numéro du signal qui a causé l'arrêt du fils. Ne peut être évaluée que si WIFSTOPPED renvoie vrai.

  Plus d'info : `man 2 wait`

# Synchronisation

## Exemple du `wait` :

```
#include <stdio.h> #include <sys/wait.h>
#include <unistd.h> #include <stdlib.h>

int main (void){
 int status ;
 pid_t pid = fork();
 switch (pid){
   case -1 :
     perror("fork") ;
     exit (1) ;
   case 0 : /* le fils */
     printf("processus fils\n") ;
     exit (2) ;
  default : /* le pere */
     printf("pere: vient de créer le processus %d\n",
         pid);
     wait (&status);
     if (WIFEXITED (status))
       printf("fils termine normalement: status = %
           d\n", WEXITSTATUS(status));
     else
       printf("fils termine anormalement\n");
 }
```

# Synchronisation

**Exemple du `waitpid` :**

```
pid_t code= fork();
if (code == -1) {... /* traitement de l'erreur */}
if(code == 0) {/* Code du fils */
   ...
   exit(23);
}
else{
   /* Code du père */
   int status;
   ...
   waitpid(-1,&status, 0); /* Attente d'un fils */
   if (WIFEXITED(status)) {
      fprintf(stdout, "Le fils a retourné %d\n",
                        WEXITSTATUS(status));
   }
}
```

# Autres primitives

**Primitive `sleep()`**

- La primitive `sleep()` est similaire à la commande shell `sleep`. Le processus est bloqué durant le nombre de secondes spécifié, sauf s'il reçoit entre temps un signal.

```
#include <unistd.h>

int sleep (int secondes)
```

- L'effet de `sleep` est très différent de celui de `wait` : `wait` bloque jusqu'à ce qu'une condition précise soit vérifiée (par exemple, la mort d'un fils), alors que `sleep` attend pendant une durée fixe.

sleep ne doit jamais être utilisé pour tenter de synchroniser deux processus

Recouvrement

# Recouvrement

**Principe du recouvrement :**

- Il s'agit d'un ensemble de primitives permettant à un processus de charger en mémoire un nouveau code exécutable.
- Cela se fait à aide des primitives de la famille `EXEC` qui permettent de faire exécuter par un processus un autre programme que celui d'origine.
- Lorsqu'un processus exécute un appel EXEC, il charge donc un autre programme exécutable en conservant le même environnement système :
  - Le processus « repart à 0 » avec le texte d'un autre programme.
  - Il garde son pid, ppid, son propriétaire et groupe réel, son répertoire courant, son umask, ses signaux pendants, ...

# Recouvrement

## Primitives de recouvrement `exec` :

- Il existe plusieurs types de fonctions EXEC. Les différents noms de ces fonctions sont des mnémoniques :
  - l **(list)** : paramètres données dans une liste terminée par NULL
  - v **(vector)** : arguments sont forme d'un tableau
  - p **(path)** : recherche du fichier avec la variable d'environnement PATH.
  - e **(environment)** : transmission d'un environnement en dernier paramètre, en remplacement de l'environnement courant.
- Ils sont mélangés pour fournir les fonctions suivantes :
  - `int execl(const char *path, const char *arg, ...);`
  - `int execlp(const char *file, const char *arg, ...);`
  - `int execle(const char *path, const char *arg, ..., char * const envp[]);`
  - `int execv(const char *path, char *const argv[]);`
  - `int execvp(const char *file, char *const argv[]);`
  - `int execvpe(const char *file, char *const argv[], char *const envp[]);`
- Plus d'information : `man 3 exec`

# Recouvrement

## Primitive `execl`

```
#include <stdio.h>
void execl(char *nom, char *arg0, ...., char *argN, NULL);
```

Les arguments de la commande sont fournis sous la forme d'une liste finissant par un pointeur NULL.

- **nom** est une chaîne de caractères donnant le chemin absolu du nouveau programme à substituer et à exécuter.
- **arg0,...,argN** sont les arguments du programme.
- Le premier argument, **arg0**, reprend en fait le nom du programme.
- **NULL** : car on ne connaît pas la taille de la liste a priori.

**Exemple :**

```
#include <stdio.h>
int main()
{
   execl("/bin/ls","ls","-l",NULL);
   printf("Erreur lors de l'appel a ls \n");
}
```

# Recouvrement

## Primitive execv

```
#include <stdio.h>
void execv(char *nom, const char *argv[]);
```

Les arguments de la commande sont sous la forme d'un tableau de pointeurs dont chaque élément pointe sur un argument, le tableau étant terminé par un pointeur NULL.

- **nom** est une chaîne de caractères donnant l'adresse du nouveau programme à substituer et à exécuter.
- `argv[]` contient la liste des arguments.

**Exemple :**

```
#include <stdio.h>
#define NMAX 5
int main()
{
   char *argv[NMAX];
   argv[0] = "ls";
   argv[1] = "-l";
   argv[2] = NULL;
   execv("/bin/ls",argv);
   printf("Erreur lors de l'appel a ls \n");
}
```

## Primitive `system`

La fonction `system()` de la bibliothèque standard propose une manière simple d'exécuter une commande depuis un programme, comme si la commande avait été tapée dans un shell.

```
#include <stdlib.h>
int system (const char *commande);
```

**Exemple :**

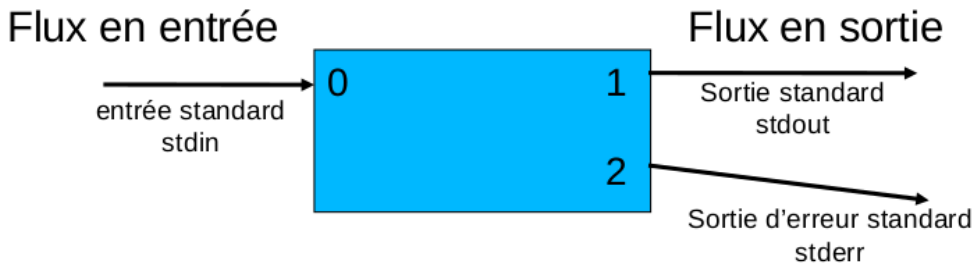
```
#include <stdio.h>
#include <stdlib.h>

int main()
{
  int return_value;
  return_value = system ("ls -l");
  return return_value;
}
```

Cette méthode est relativement simple mais doit être utilisée avec modération car elle est peu performante et présente des risques de sécurité.

Entrée / Sortie

# Descripteurs de fichiers d'un processus

- Tout processus possède à sa création 3 descripteurs de fichiers :
  - 0 : stdin (entrée standard)
  - 1 : stdout (sortie standard)
  - 2 : stderr (sortie erreur)




- Par défaut ces trois descripteurs sont reliés au terminal correspondant à la session

# Redirection

- Redirection de l'entrée standard
  - `commande < nom_de_fichier`
- Redirection de la sortie standard
  - `commande > nom_de_fichier`
  - `commande » nom_de_fichier` (redirection sans écrasement)
- Redirection de la sortie d'erreur standard
  - `commande 2> nom_de_fichier`
- Redirection de la sortie d'erreur vers la sortie standard, qui a été redirigé vers le fichier nom_de_fichier.
  - `commande > nom_de_fichier 2>&1`

# Paramètres sous Linux

```
int main (int argc, char *argv[]);
```

- argc : nombre de paramètres passés plus le nom de l'exécutable
- argv : tableau contentant les paramètres passés

**Exemple :** Exécution d'un programme en C **prog** sous un shell en récupérant les paramètres :

- $**prog** param1 param2
  - `argc` 3
  - `argv[0]` **prog**
  - `argv[1]` param1
  - `argv[2]` param2

# Variables d'environnement

- Variables d'environnement sont des affectations de type *Nom = Valeur* qui sont disponibles pour tous les processus du système, y compris le shell.
- Nous pouvons accéder aux variables d'environnement d'un processus par l'intermédiaire de deux fonctions :

```
#include <stdlib.h>
```

```
char *getenv(const *variables_environnement)
```

  - Permet d'acceder à une variable d'environnement particulière à partir de son nom. La fonction récupère un pointeur sur la valeur de cette variable ou retourne NULL.

```
int putenv(const char *chaine)
```

  - Permet d'assigner une variable d'environnement. Modifie ou ajoute une variable d'environnement *variable=valeur*. Retourne 0 dans le cas correct.

# Manipulation des fichiers en C (open)

Les fichiers seront toujours manipulés avec des primitives bas niveau (non bufferisées) :

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int desc;
int open (const char *nomfich, int mode_ouverture,
    mode_t droits);
  // demande une nouvelle entrée dans la table des
      fichiers ouverts du systeme
  // mode_t droits est optionnel, n'est utilisé que
      lorsque open() réalise la création du fichier.

desc = open(nomfich, mode_ouverture, droits);
```

# Manipulation des fichiers en C (open)

Paramètre **mode_overture :**

- construit par disjonction bit à bit des constantes définies dans `fcntl.h`

| | |
|---|---|
| O_RDONLY | En lecture |
| O_WRONLY | En écriture |
| O_RDWR | En lecture et en écriture |
| O_NONBLOCK | Ouverture dans un mode non bloquant |
| O_APPEND | Ajout à la fin du fichier |
| O_CREAT | Crée le fichier s'il n'existe pas |
| O_TRUNC | Remet le fichier à 0 s'il existe |
| O_EXCL | Echoue si le fichier existe |

# Manipulation des fichiers en C (open)

Paramètre **droits :**

| | |
|---|---|
| S_IRUSR | Lecture pour le propriétaire |
| S_IWUSR | Écriture pour le propriétaire |
| S_IXUSR | Exécution pour le propriétaire |
| S_IRGRP | Lecture pour le groupe |
| S_IWGRP | Écriture pour le groupe |
| S_IXGRP | Exécution pour le groupe |
| S_IROTH | Lecture pour les autres |
| S_IWOTH | Écriture pour les autres |
| S_IXOTH | Exécution pour les autres |

# Manipulation des fichiers en C (close)

Les fichiers seront toujours manipulés avec des primitives bas niveau (non bufferisées) :

```
#include <unistd.h>
int close(int desc);
  // Cette primitive permet de libérer un descripteur
       dans la table des descripteurs
```

# Manipulation des fichiers en C (read et write)

Ces primitives permettent la lecture et l'ecriture dans un fichier de descripteur *desc* :

```
#include <unistd.h>
ssize_t read (int desc, void *ptr, size_t nb_octets)
```

- Le fichier doit être ouvert en lecture (options `O_RDONLY`, `O_RDWR`).
- `nb_octets` : le nombre (int) d'octets que l'utilisateur voudrait lire.
- `ssize_t` : reçoit le nombre d'octets réellement lus.
- La fonction read détecte la fin de fichier et renvoie -1.

```
#include <unistd.h>
ssize_t write (int desc, void *ptr, size_t nb_octets)
```

- Écrit jusqu'à `nb_octets` octets dans le fichier associé au descripteur `desc` depuis le tampon pointé par `ptr`.
- Le nombre d'octets écrits peut être inférieur à `nb_octets`.
- Renvoie le nombre d'octets écrits (0 signifiant aucune écriture), ou -1 s'il échoue, auquel cas `errno` contient le code d'erreur.

# Manipulation des fichiers en C (lseek)

```
#include <sys/types.h>
#include <unistd.h>

off_t lseek (int desc, off_t offset, int origine)
```

- Modifie l'offset par rapport à une origine dans le fichier de descripteur desc

# Manipulation des fichiers en C

## Fonction **stat** :

Les fonctions `stat`, `fstat` et `lstat` obtiennent l'état d'un fichier (*file status*).

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

int stat(const char *path, struct stat *buf);
int fstat(int fd, struct stat *buf);
int lstat(const char *path, struct stat *buf);
```

- `stat()` récupère l'état du fichier pointé par `path` et remplit le tampon `buf`.
- `lstat()` est identique à `stat()`, sauf que si path est un lien symbolique, il donne l'état du lien lui-même plutôt que celui du fichier visé.
- `fstat()` est identique à `stat()`, sauf que le fichier ouvert est pointé par le descripteur `fd`, obtenu avec `open`.

Plus d'information : http://manpagesfr.free.fr/man/man2/stat.2.html

# Manipulation des fichiers en C

Structure **stat** :

Les trois fonctions précédentes retournent une structure stat contenant les champs suivants :

```
#include <sys/stat.h>
struct stat {
 dev_t     st_dev;     // ID periph. ou est le fichier
 ino_t     st_ino;     // Numero inoeud
 mode_t    st_mode;    // Protection
 nlink_t   st_nlink;   // Nb liens materiels
 uid_t     st_uid;     // UID proprietaire
 gid_t     st_gid;     // GID proprietaire
 dev_t     st_rdev;    // ID periph. si fichier special
 off_t     st_size;    // Taille totale en octets
 blksize_t st_blksize; // Taille de bloc pour E/S
 blkcnt_t  st_blocks;  // Nombre de blocs alloues
 time_t    st_atime;   // Heure dernier acces
 time_t    st_mtime;   // Heure derniere modification
 time_t    st_ctime;   // Heure dernier changement etat
};
```

# Manipulation des fichiers en C

## Structure **stat** :

La valeur du champ `st_mode` est une combinaison logique par l'opérateur de disjonction | des constantes suivantes :

| **Nom symbolique** | **Interprétation du bit** |
|---|---|
| S_ISUID, S_ISGID | le set_uid et le set_gid<br>un autre bit pour le sticky bit |
| S_IRUSR | lecture par le propriétaire |
| S_IWUSR | écriture par le propriétaire |
| S_IXUSR | exécution par le propriétaire |
| S_IRWXU | lecture, écriture et exécution par le propriétaire |
| S_IRGRP | lecture par les membres du groupe propriétaire |
| S_IWGRP | écriture par les membres du groupe propriétaire |
| S_IXGRP | exécution par les membres du groupe propriétaire |
| S_IRWXG | lecture, écriture et exécution par le groupe |
| S_IROTH | lecture par les autres |
| S_IWOTH | écriture par les autres |
| S_IXOTH | exécution par les autres |
| S_IRWXO | lecture, écriture et exécution par les autres |