# MI206 - Systèmes d'exploitation

## SEE04-2

M. Faverge
mfaverge@enseirb-matmeca.fr

ENSEIRB-Matmeca

2014 - 2015

# Objectifs du cours

**Qu'est ce qu'un système d'exploitation ?**

- Décrire les principes généraux
- Montrer des exemples pratiques
- Etudier et manipuler les concepts de bases

# Objectifs du cours

**Qu'est ce qu'un système d'exploitation ?**

- Décrire les principes généraux
- Montrer des exemples pratiques
- Etudier et manipuler les concepts de bases

**Ce que le cours n'est pas:**

- cours de programmation C
- apprenez le C ! Lisez le C, de B. Kernighan et D. Richie
- cours de Shell
- apprenez votre shell ! Utilisez `man`

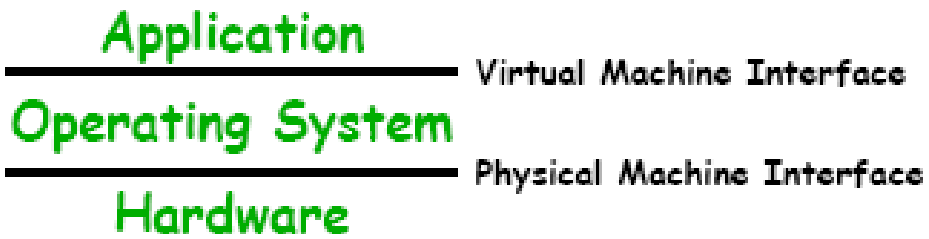
# Plan

# Plan

# Qu'est ce qu'un système d'exploitation?

**Un ordinateur est constitué de:**

1. du matériel
2. d'un système d'exploitation
3. de programmes



# Qu'est ce qu'un système d'exploitation?

## Le matériel

L'ensemble des dispositifs physiques qui constituent la couche la basse:

- le(s) processeur(s),
- la mémoire,
- le(s) disque(s),
- le clavier,
- l'imprimante,
- les interfaces réseaux,
- ...

# Qu'est ce qu'un système d'exploitation?

## Les programmes/applications

C'est en ensemble de logiciels écrits par des éditeurs de logiciels, ou par les utilisateurs eux-mêmes. Ils ont pour but de résoudre des problèmes plus ou moins spécifiques tel que:

- le traitement de données,
- les calculs scientifiques,
- divertir l'utilisateur,
- ...

# Qu'est ce qu'un système d'exploitation?

## Le système d'exploitation

C'est le programme **le plus important** du système. Il a deux fonctions importantes:

- Fonction de **machine étendue** (ou **machine virtuelle**): permet de faciliter l'utilisation de la machine.
- Fonction de **Gestionnaire de ressources**: permet l'accès concurrents aux ressources matérielles.

# Qu'est ce qu'un système d'exploitation?

## Le système d'exploitation

C'est le programme **le plus important** du système. Il a deux fonctions importantes:

- Fonction de **machine étendue** (ou **machine virtuelle**): permet de faciliter l'utilisation de la machine.
- Fonction de **Gestionnaire de ressources**: permet l'accès concurrents aux ressources matérielles.

Et il se compose:

- **d'un noyau**: partie critique d'un OS. Elle permet la communication entre les couches matérielles et logicielles affin de former un tout. Le noyau est le premier logiciel chargé en mémoire.
- **d'un ensemble d'appels systèmes**: bibliothèque qui regroupe les fonctions permettant à l'utilisateur d'utiliser l'OS et les périphériques et de les configurer.

# Fonction de **machine étendue**

- Rôle *d'interface* entre les applications et le matériel.
- Peut-être assimiler à une **machine étendue** ou **machine virtuelle**) qui facilite l'accès aux ressources et dont le but est de:
  - cacher la complexité du matériel
  - fournir un ensemble de bibliothèques standard (fenêtrage par ex.)
  - rendre la programmation plus facile, plus simple

# Fonction de **Gestionnaire de ressources**

Pour chacune des ressources, le système doit:

- connaître l'utilisateur de la ressource à tout moment,
- en accorder l'usage de manière équitable,
- gérer et éviter les conflits d'accès entre les différents programmes et utilisateurs.
- gérer les erreurs
- empêcher les usages impropres de la machine

Les deux tâches essentielles du système en tant que gestionnaire de ressources sont le **partage** et la **protection**.

# Fonction de **Gestionnaire de ressources**

**Quelles ressources, quels services?**

- Temps CPU:
  - Ordonnancement des tâches sur le processeur (scheduling)
- Mémoire:
  - Allocation
  - Protection mémoire
- Entrées/sorties:
  - Système de fichiers et droits,
  - Gestion des diverses cartes et protocoles réseau,
  - Fenêtrage
- Consommation d'énergie: réglage fréquence, ...

# Les fonctions de base d'un système d'exploitation

1. Les processus
   - gestion de l'exécution des programmes et du partage du temps CPU.
2. La mémoire
   - Gestion des transferts entre les différent niveaux de la mémoire
   - Protection des accès entre les processus/utilisateurs
   - Virtualisation
3. Le système de fichiers
   - Proposer une vision uniforme et structurée des données et ressources du système.
4. Les entrées-sorties
   - Proposer des mécanismes de communications entre processus et/ou processus/périphériques.

# Autres fonctions d'un système d'exploitation

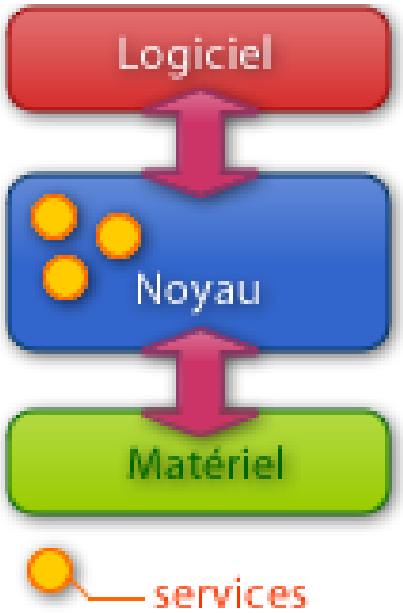
- Les réseaux d'ordinateurs
  - protocoles de communication, d'interconnexion.
- Les systèmes répartis
  - protocoles de d'appels de procédure à distance (RPC)
  - objets distribués
- Les systèmes de fenêtrage graphique

# Les différentes architectures de systèmes d'exploitation

- Monolithiques
- Modulaire / Multicouches
- Micronoyaux
- Exonoyaux
- Machine virtuelle

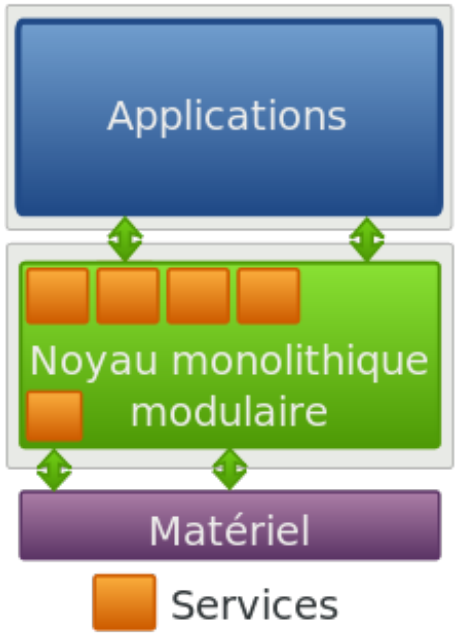
# OS monolithiques

- Un seul bloc contenant l'ensemble des services du système.
- Usine à gaz !!!
- Facilité de conception
- Performance peut être au RDV ...
- Code dur à maintenir
- exemple: Dos, très vieux UNIX et Linux, ...



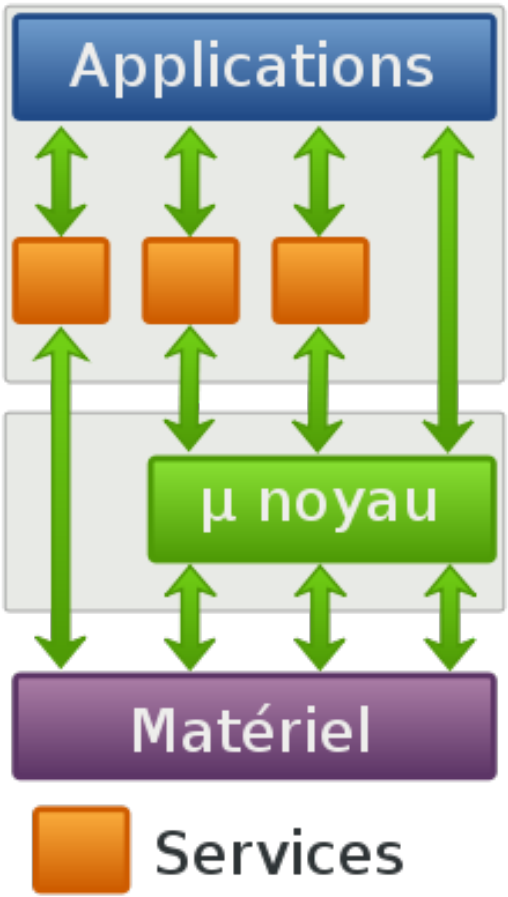
# OS multicouches

- OS organisé en hiérarchie de couches. Chacune construite sur la base des services offerts par la couche inférieure.
- Facilité de conception et de développement
- Code plus organisé et maintenable.
- Chargement des fonctionnalités à la demande
- exemple: Linux, BSD, Solaris



# OS Micronoyaux

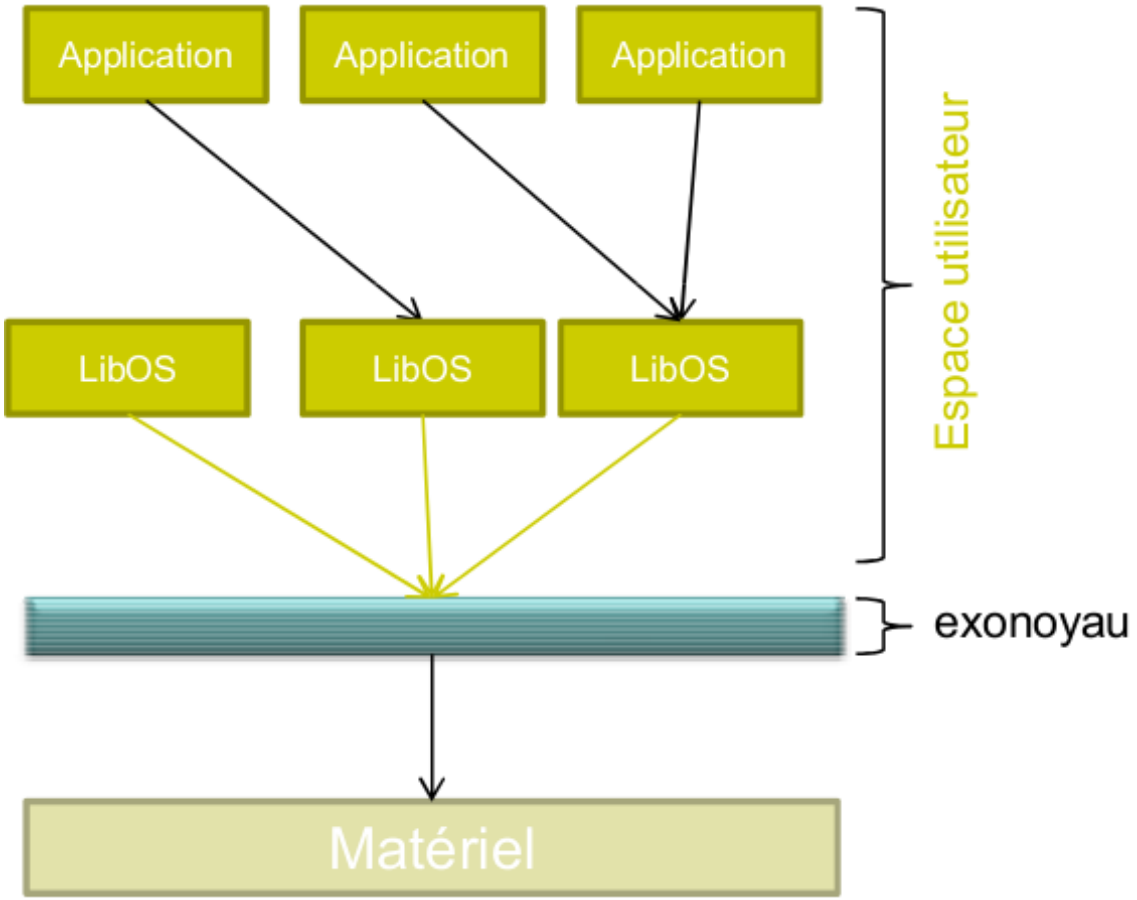
- Déplace plusieurs fonctions de l'OS vers des *processus serveur* s'exécutant en mode utilisateur $\Longrightarrow$ réduction au maximum de la taille du code privilégié.
- But: gérer les communications entre applications et serveurs pour:
  - Renforcer la politique de sécurité
  - Permettre l'exécution de fonctions système (accès aux registres d'E/S, etc.).
- Fiabilité augmentée: si un processus serveur *crash*, le système continue à fonctionner et il est possible de relancer le service sans redémarrer.
- Modèle facilement étendu à des systèmes distribués
- exemple: Mac OS-X, GNU HURD, WindowsNT



# OS Exonoyaux

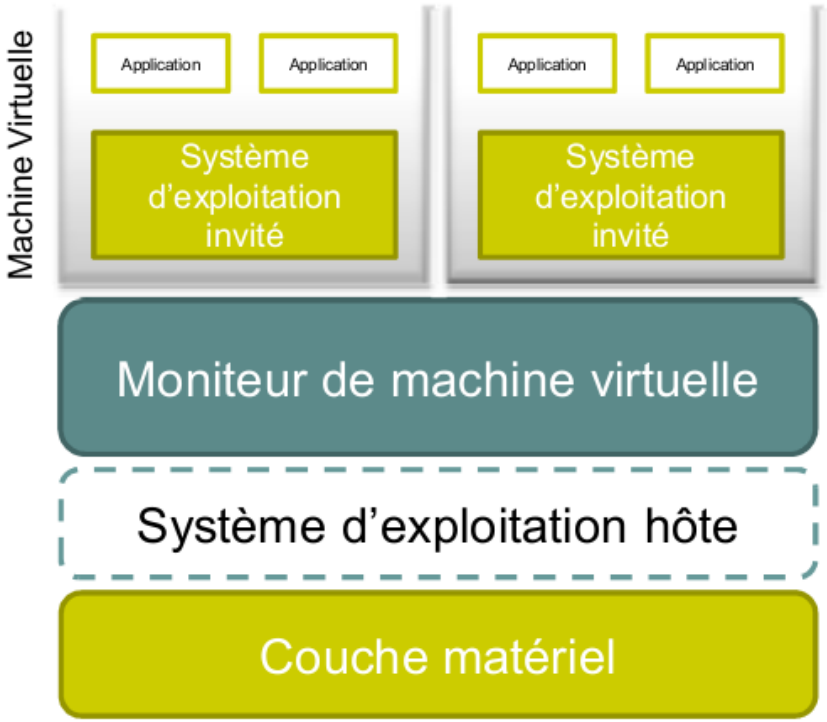
- Le noyau se contente de multiplexer et protéger l'accès aux ressources. Tout le traitement qu'on trouve habituellement dans un noyau (mémoire virtuelle, système de fichiers, ...) est délégué à l'espace utilisateur via des bibliothèques appelées *LibOS*. Plusieurs LibOS peuvent être utilisées en parallèle.
- Gain de performance
- Pb de sécurité
- Isolation de bugs
- Domaine encore balbutiant.
- exemple: XOK couplé à ExOS comme LibOS (MIT).

# OS Exonoyaux



# OS machine virtuelle

- Possibilité de mettre plusieurs OS sur une machine physique.
- Le moniteur de machine virtuelle (hyperviseur) intercepte les instructions privilégiées envoyées par l'OS invité, les vérifie (politique de sécurité) et les exécute.
- exemple: XEN, VMWare, QEMU



# Autre classification

- **OS temps partagé**: garantir le partage équitable du temps processeur et des ressource dans le but de maximiser le temps de traitement et de réduire le temps de réponse moyen.
- **OS temps réel**: garantir les temps de réponse
  - *Systèmes à contraintes souples/molles*: systèmes acceptant des variations minimes de temps de réponse (systèmes multimédias)
  - *Systèmes à contraintes dures*: gestion stricte du temps pour conserver l'intégrité du système (déterminisme logique et temporel et fiabilité)
- **OS embarqué**: OS prévus pour fonctionner sur des machines de petite taille, (PDA ou des appareils électroniques autonomes: sondes spatiales, robot, ordinateur de bord, etc.), possédant une autonomie réduite.
  $\Longrightarrow$ gestion avancée de l'énergie, ressources limitées.

# Références

- **Livre**
  - Operating System Concepts, Silbershatz, Galvin, Gagne ($8^{eme}$ édition)
- **Cours en ligne**
  - J. Kubiatowicz, CS 162, Berkeley University
  - D. Maziere, CS 140, Stanford University
  - A. Cohen, INF570, école Polytechnique
  - J. Boukhobza, Université de Brest
- **Autres ressources en ligne**
  - `http://www.top500.org`
  - `http://www.wikipedia.org`
  - `http://www.linuxmanpages.com`
    ou `http://linux.die.net`
  - `http://www.osdata.com`

# Plan

# Plan

# Notion de processus

- Un processus est un programme qui s'exécute.
- Un processus est **dynamique** par opposition à un programme qui lui est **statique**

# Notion de processus

## Processus

Un programme qui s'exécute ainsi que son contexte (mémoire, état des descripteurs de fichiers)

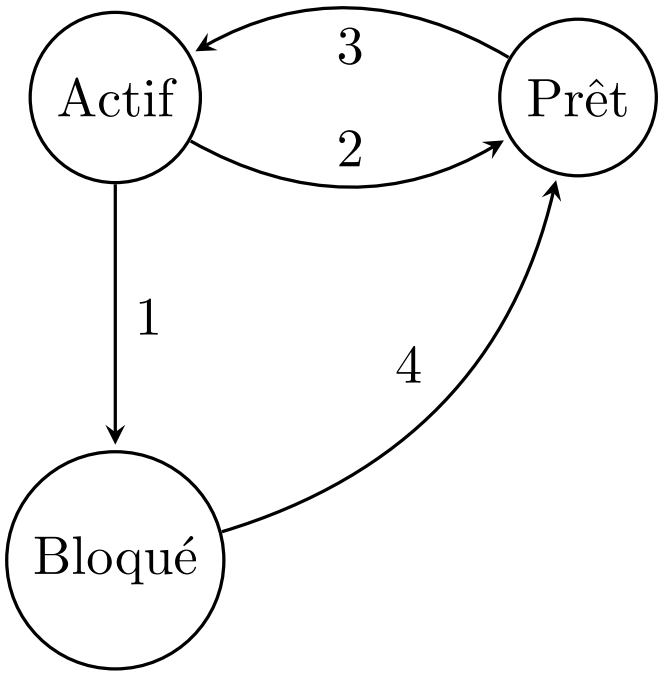
**Dans les systèmes d'exploitation:**

- Support pour de multiples processus
  - être capable de basculer d'un processus à un autre.
- Chaque processus a un espace d'adressage privé
  - Garantit le cloisonnement entre processus (sécurité et protection)
  - Abstraction de la mémoire (mémoire virtuelle)
  - Cache le noyau et les autres processus !

# Plan

# Ordonnancement des processus

## Diagramme simplifié d'état des processus




- A un instant donné, un seul processus est **Actif**.
- Les autres processus seront:
  - **Prêt**: en attente d'exécution,
  - **Bloqué**: en attente de ressource ou de données.

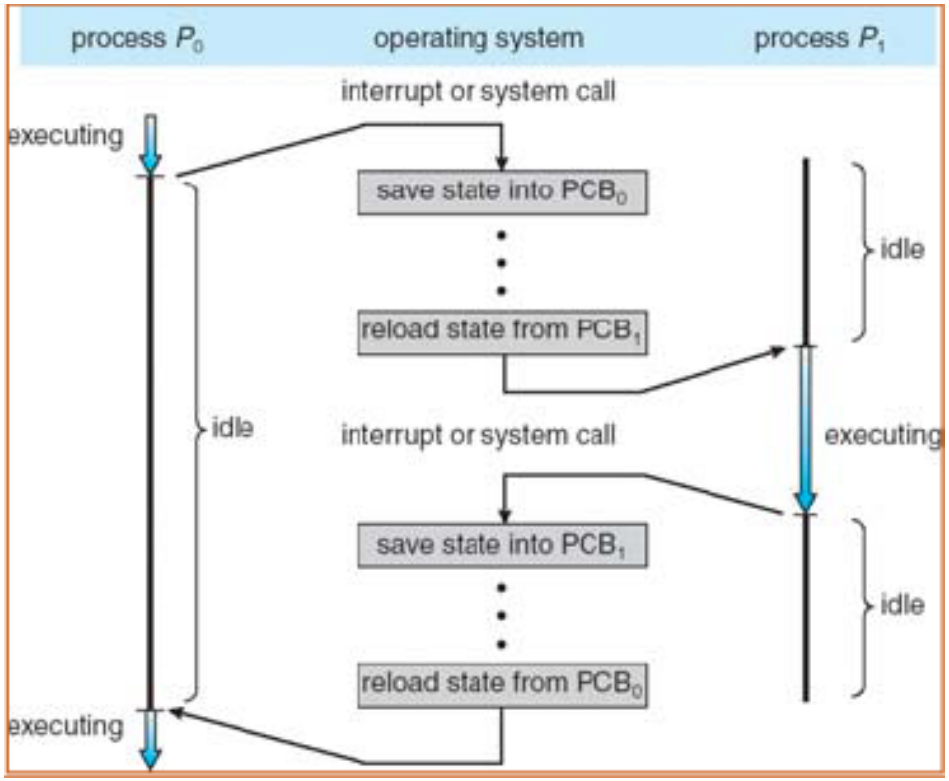
1. Le processus se bloque en attente de ressources/données;
2. L'ordonnanceur choisit d'exécuter un autre processus;
3. L'ordonnanceur choisit le processus;
4. Les ressources/données sont disponibles.

# Ordonnancement des processus

## Changement de contexte

**Comment le processeur passe d'un processus à un autre?**

- Cela s'appelle un *context switch*.
- Le temps du *context switch* doit être réduit au minimum (cf commande `time`)
- L'OS définit une politique d'ordonnancement (scheduling)



# Ordonnancement des processus

## Changement de contexte

**Mécanismes classiques pour le scheduling**

- Mécanisme de préemption:
  - Un processus est interrompu au bout d'un quota de temps (10ms par ex.)
  - Certains OS (temps réel) ne sont pas préemptifs, les processus doivent laisser la main à l'OS volontairement.
- Algorithme d'ordonnancement:
  - Utilise des priorités
  - Tourniquet (round robin): tous les processus ont droit à un quota de temps à tour de rôle

# Ordonnancement des processus

## États possible d'un processus

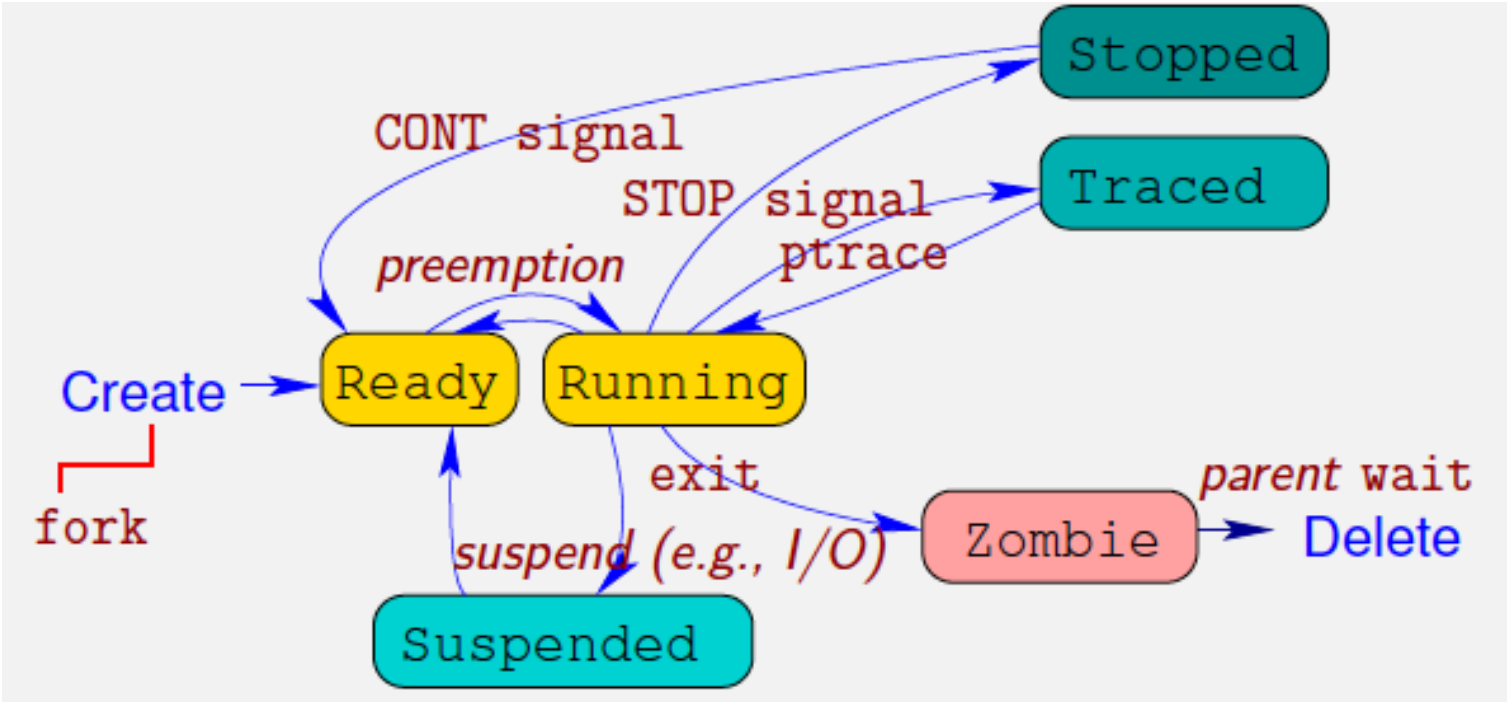
- **Prêt**: est en attente du processeur, dépend de l'ordonnancement
- **En exécution**: occupe le processeur
- **Suspendu**: en attente d'une entrée sortie
- **Stoppé, tracé**: en attente d'un signal, en position d'attente (pour synchronisation avec d'autres processus par ex.)
- **Zombie puis arrêté**: attend que le processus parent recoive le signal de terminaison du fils puis termine (PCB détruit)

La commande `ps` donne la liste des processus en cours et leur état. `pstree` donne la filiation entre processus.

# Ordonnancement des processus

## États possible d'un processus

- **Création**: créer par clonage avec `fork`
- **Prêt**: prêt à être exécuté, en attente du processeur



# Plan

# Composition du contexte d'un processus
## Introduction

- Soit les deux commandes suivantes:
  - `ls -t` (Ordre chronologique)
  - `ls -l` (Liste détaillée)
- L'exécution de chacune de ces commandes va engendrer la création d'un processus:
  - Un identifiant (`pid`) différent
  - des données différentes
  - un état différent
- et ce, malgré un même programme.

# Composition du contexte d'un processus

## Introduction (3)

Le contexte d'un processus se décompose en deux parties:

1. Le **bloc de contrôle (PCB)** qui comporte toutes les données nécessaires au système pour l'exécution d'un processus.
2. La **structure interne** qui sera contenue dans la mémoire principale sous forme de plusieurs segments et qui contient les données internes au processus et indépendantes du système.

# Composition du contexte d'un processus

## PCB: bloc de contrôle

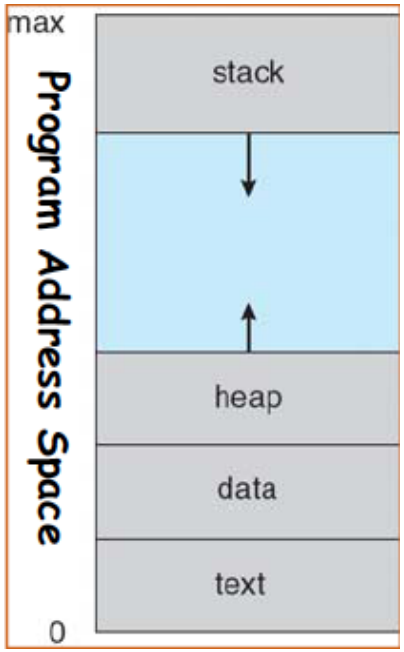
- Le contexte d'un processus contient tous les éléments nécessaire au système d'exploitation pour l'exécution d'un processus à un instant donné:
  - **Etat**: en cours d'exécution, en attente, prêt, zombie, ...
  - **Etat de la machine**: registres, PC
  - **ID, parent**: pointeur vers le processus qui l'a lancé (PPID)
  - **Fichiers**: liste des descripteurs de fichiers ouverts
  - **Limites**: définit une limite sur le nombre de processus fils, le nombre de descripteurs de fichiers ouverts, ... (`man getrlimit`
  - **Signaux**: quels comportements sont prévus, pour quels signaux (sera vu après)
  - **Table des pages mémoire**: quelles pages mémoires sont allouées par le processus
  - **Threads**: information sur les threads
  - ...
- Cet ensemble de données doit permettre au système de reprendre l'exécution d'un processus qui a été préalablement interrompu.

# Structure interne
## Description

- Chaque processus à un espace d'adressage propre pour ses données
- Cet espace d'adressage est *virtualisé*, pas d'adresses physiques pour plus de sécurité
- Chaque processus ne voit que sa propre structure interne (son espace d'adressage virtuel)
- La mémoire est initialisée pour chaque processus en différentes zones ou segments
- Chaque processus possède quatres zones: la pile, le tas, les données et le texte.
- Espace de taille $2^n$ adresses avec des adresses sur $n$ bits

# Les 4 zones

- Stack/Pile: permet d'empiler les appels de fonctions avec leurs paramètres et variables locales. Lors du retour de fonction, l'ensemble est dépilé.
- Heap/Tas: contient l'ensemble des variables allouées dynamiquement (malloc)
- Data/Données: zone de stockage des données globales et/ou statiques
- Text/Texte: zone de stockage du code, des instructions du programme en langage machine.

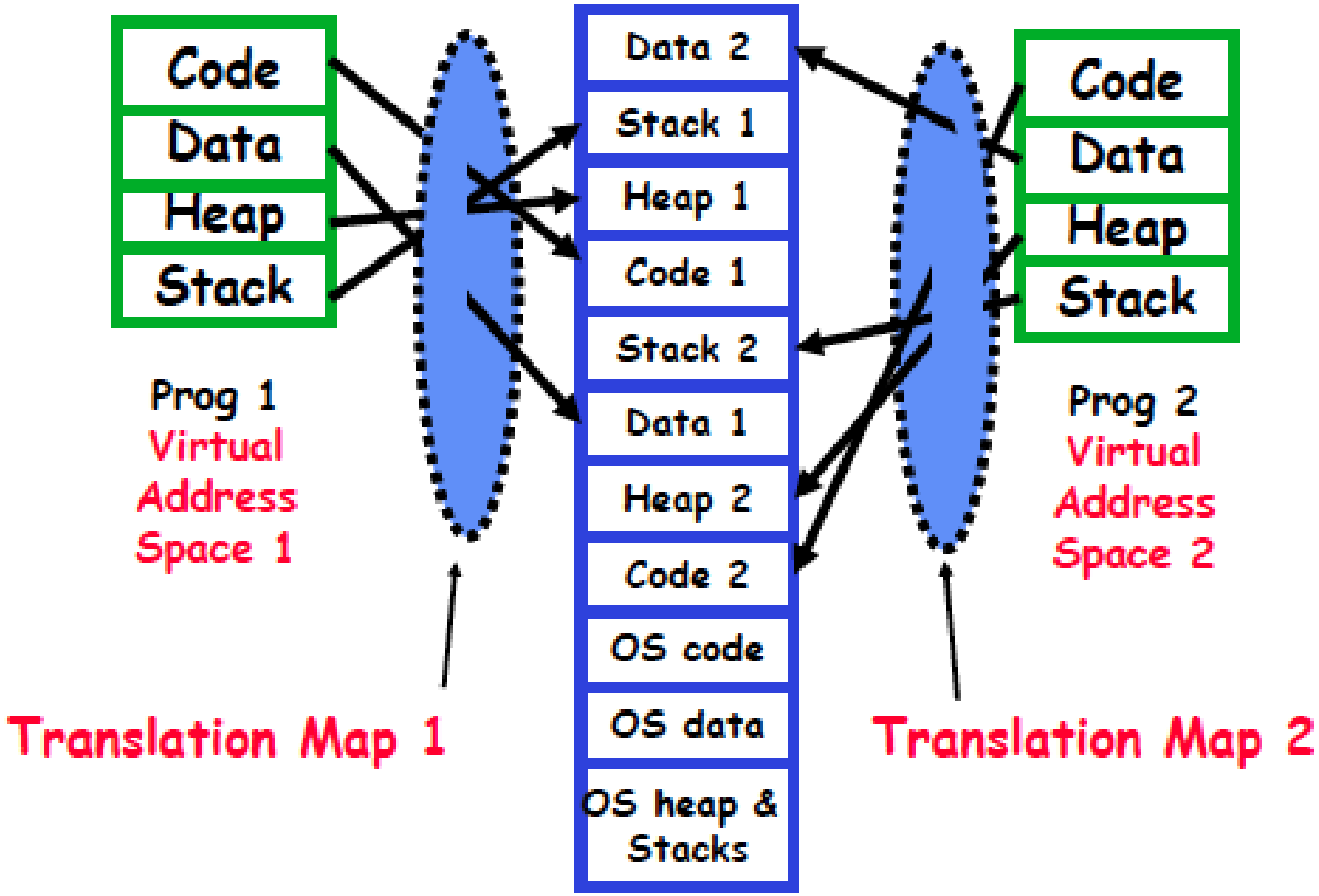


# Structure interne
## Gestion mémoire par l'OS

- Traduction mémoire virtuelle → mémoire réelle fait par le CPU (MMU/TLB)
- Mécanisme de pages:
  - La mémoire est gérée par blocs de 2k cellules mémoire (4Ko ou 2Mo sur x86), des pages
  - Chaque page virtuelle correspond à une page réelle
- La traduction s'appuie sur un mapping des pages (traduction par dictionnaire)
- L'adresse virtuelle est composée de différents champs permettant de construire l'adresse réelle

# Structure interne

## Gestion mémoire par l'OS



# Structure interne

## Gestion mémoire par l'OS (Quelques bases)

- Un ensemble de pages physiques sont allouée à un processus (pour les différentes zones)
- Allocation paresseuse
  - L'allocation de la page est réellement faite que lorsqu'elle est réellement accédée
  - Permet gestion plus économique de la mémoire (plutôt qu'une allocation gloutonne)
- Politique de gestion mémoire:
  - Que faire si plus de pages physiques disponibles ?
  - Rôle du swap:
    - Si plus de pages physique, stocke sur disque les pages de processus en attente

# Plan

# Création de processus

## Hiérarchie de processus UNIX

- Au lancement, il n'existe qu'un seul processus qui est l'ancêtre de tous les autres.
- Chaque processus peut lancer de nouveaux processus.
- Le processus créateur est le père et les processus créés les fils.
- Structure arborescente de processus.
- Création de processus par clonage.

# Création de processus

## Processus 0

**Quel est le premier processus?**

# Création de processus

## Processus 0

**Quel est le premier processus?**
**Processus 0**

# Création de processus

## Processus 0

**Quel est le premier processus?**
**Processus 0**:

- **Un par CPU**
- Construit complètement par le noyau et tourne en mode noyau
- **N'utilise que de la mémoire statique**
- Construit et initialise les structures pour la gestion de la mémoire virtuelle
- Crée des threads noyaux (swap, log, ...)
- Crée le processus 1

# Création de processus

## Processus 1

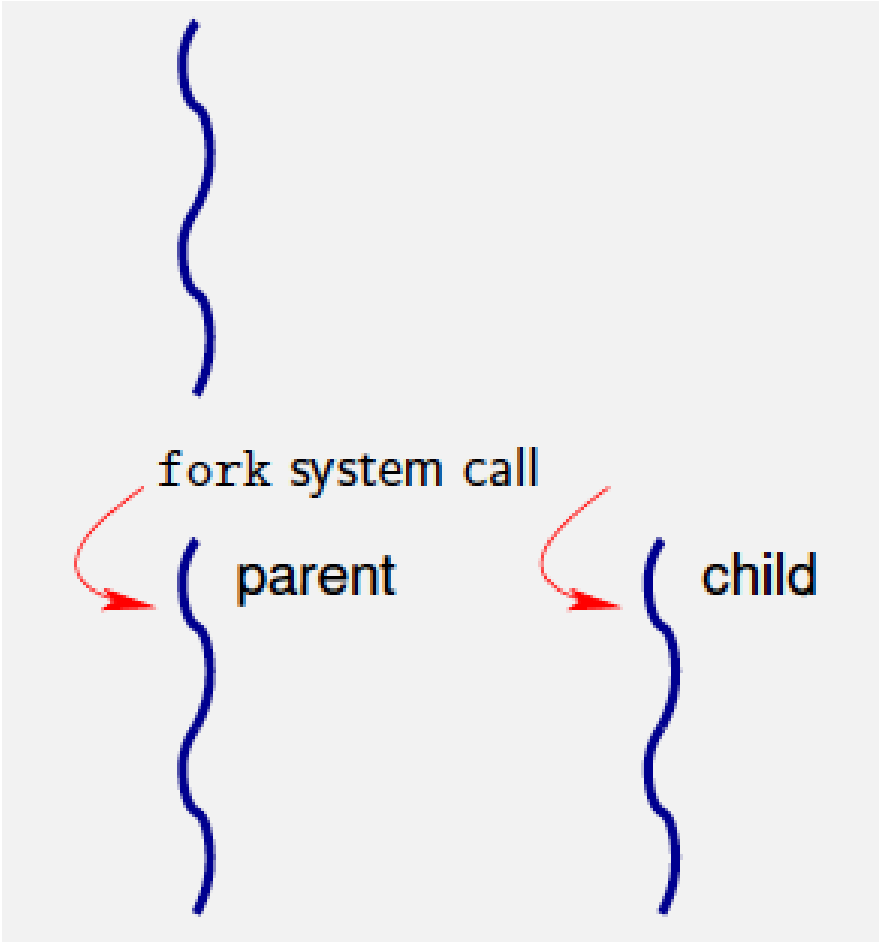
**Quel est le premier processus?**
**Processus 1**:

- **Un par CPU**
- Partage ses données avec le processus 0
- Finit l'initialisation du système
- Passe en mode utilisateur, devient un processus visible
- Exécute **/sbin/init**
- Adopte tous les processus orphelins
- Tous les autres processus sont crées à partir d'init

# Création de processus

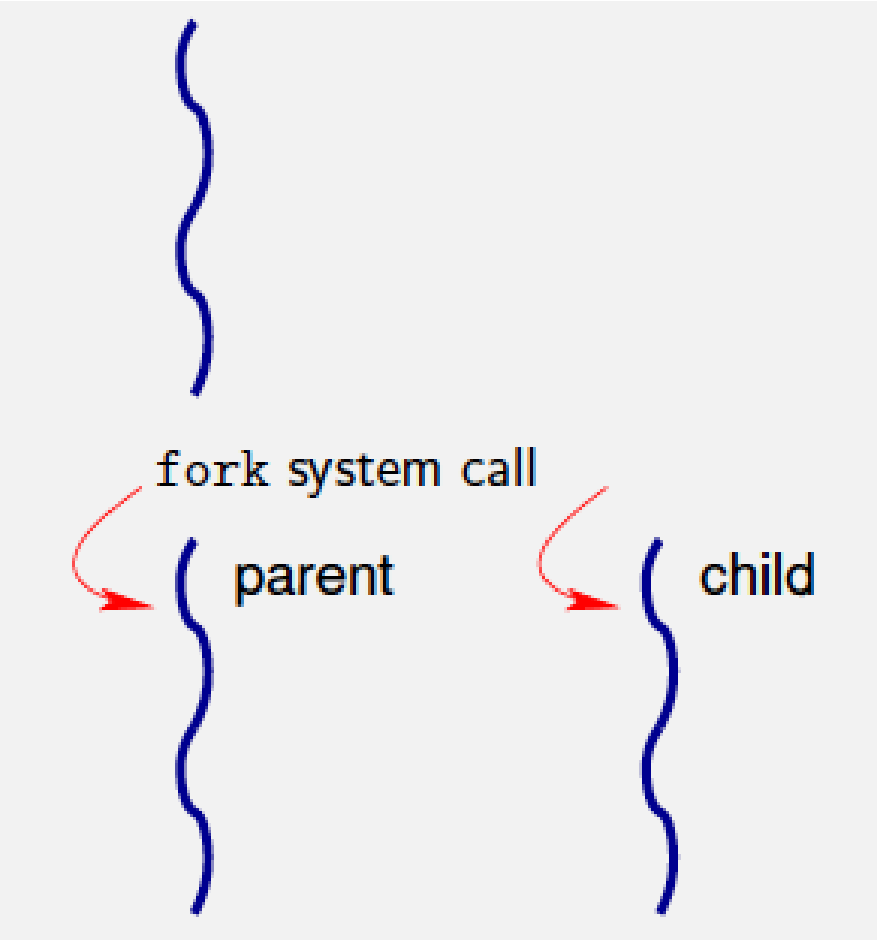
## Création par clonage

- Fonctions `fork` ou `clone`
- Le processus fils:
  - exécute le même programme
  - possède une copie de l'espace virtuel du père
- Sous linux, seule méthode de création !



# Création de processus
## Coût de création

- Création d'un Process Control Block (pas cher)
- Copie des pages (**Très cher** si on les copies toutes au lancement!)
- Utilisation de copie à l'écriture (copy on write):
  on marque la page comme devant être copiée, mais elle n'est copiée qu'à la prochaine écriture (du père ou du fils)



# Plan

# Interface de gestion des processus

## Duplication de processus: fork

```
pid_t fork();
```

Duplique le processus courant. Les 2 processus sont **parfaitement** identiques sauf pour leur identifiants respectifs: PID pour le fils, PPID pour le parent (voir `getpid()` et `getppid()`).
Le processus fils:

- NE récupère PAS les signaux en cours
- récupère les handlers et les signaux bloquant du parent

**Retourne:**

- 0 si c'est le fils
- Le `PID` du fils ($> 0$) si c'est le père
- -1 si erreur

Autres variantes: `clone` (sert aussi aux threads), `vfork`.

# Interface de gestion des processus
## fork (exemple)

```
pid_t d=fork();
switch(id) {
   case 0: // code du fils
      ...
      break;
   case -1: // erreur
      break;
   default: // code du pere
}
```

# Interface de création des processus

## Comment créer un processus différent?

La seule solution pour créer un processus exécutant un programme différent est la suivante:

1. Créer un processus par clonage
2. Remplacer les zones textes et data de la structure interne par celles du programme que l'on souhaite exécuter

La famille des fonctions systèmes `exec` permet de réaliser l'étape 2.

# Interface de création des processus

## Lancement d'un exécutable (execve)

```
int execve(const char *name, char *const argv[],
           char *const envp[]);
```

**Paramètres:**

- *name*: le chemin et nom de l'exécutable
- *argv*: tableau des arguments qui sera passé au main du programme
- *envp*: pointeur sur environnement, obtenu par `getenv()` et manipulé par `setenv()`. Définit les variables d'environnements:
  CC, PATH, LD_LIBRARY_PATH, ...

# Interface de création des processus

## Lancement d'un exécutable (execve)

```
int execve(const char *name, const *const argv[],
           char *const envp[]);
```

**Si réussi:**

- Remplace les segments heap, data (statique), texte (le code) avec celles du programme chargé
- Garde les PID, PPID, descripteurs de fichiers ouverts
- Si le fichier a le bit SUID, change l'ID effective à celle du fichier
- **L'appel ne retourne pas**
- Sinon retourne -1

**Variantes**: `execl`, `execv`, `execvp`, `execlp`, `execle`, `execve`

# Interface de gestion des processus

## Résumé des principes de bases

- La création des processus par duplication se fait avec la fonction `fork()`.
- le recouvrement du code du processus fils par un nouveau programme se fait à l'aide des fonction `exec`.
- La terminaison d'un processus avec `exit()`.
- La destruction du PCB d'un processus terminé par le son père à l'aide de `wait()`
- Si le fils termine, alors que son père ne l'as toujours pas attendu, il passe dans l'état *zombie*.

# Plan

# Fichiers

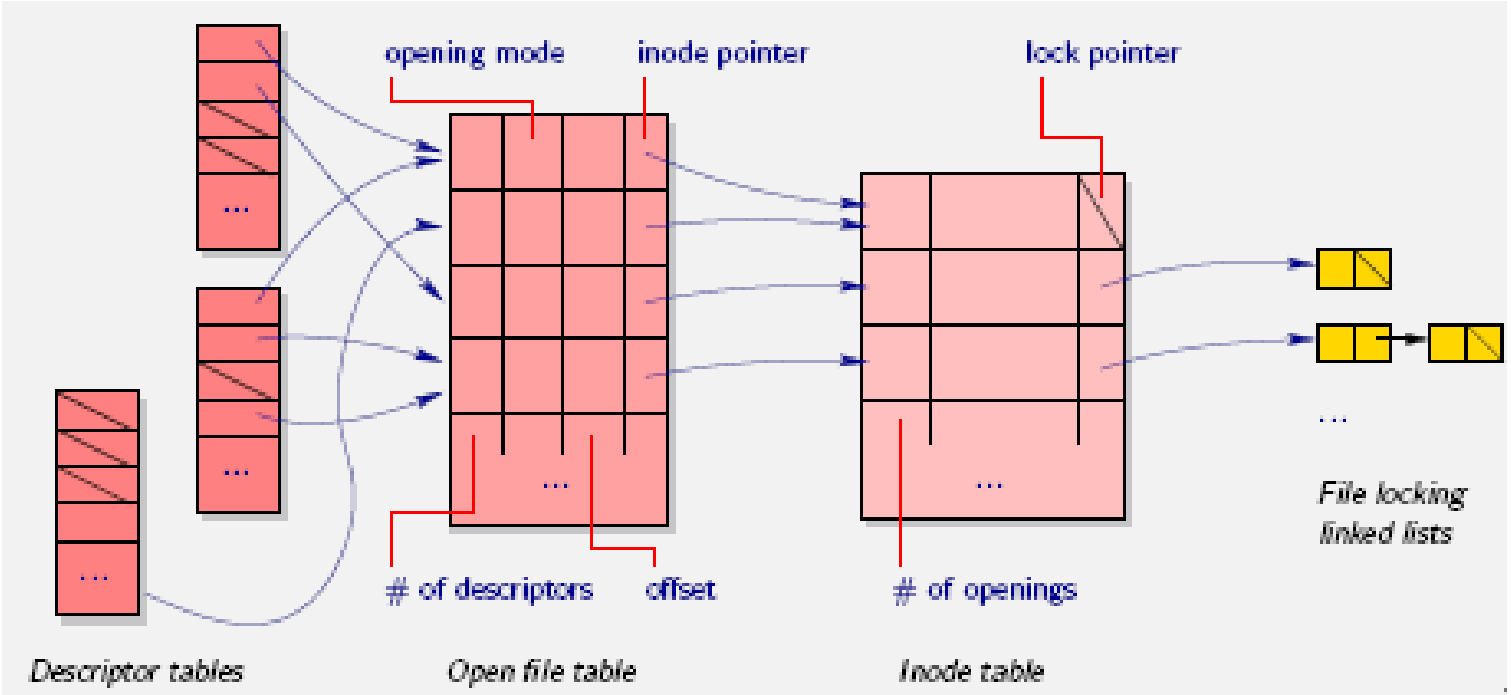
Une donnée importante et propre à un processus qui est copiée lors du clonage d'un processus est la **table des descripteurs de fichier**.

- Regroupe l'ensemble des descripteurs de fichiers ouverts par le processus
- Identifié par un entier.
- Trois descripteurs par défaut:
  0. Entrée standard (clavier)
  1. Sortie standard
  2. Sortie d'erreur
- Création de nouvelles entrées par l'intermédiaire de: `open`, `create`, `dup`, ...
- Suppression d'entrées avec `close`.

# Fichiers

## Table des descripteurs de fichier d'un processus

**Chaque processus:** Liste de descripteurs de fichiers ouverts/accédés, Pointe vers les inodes des fichiers

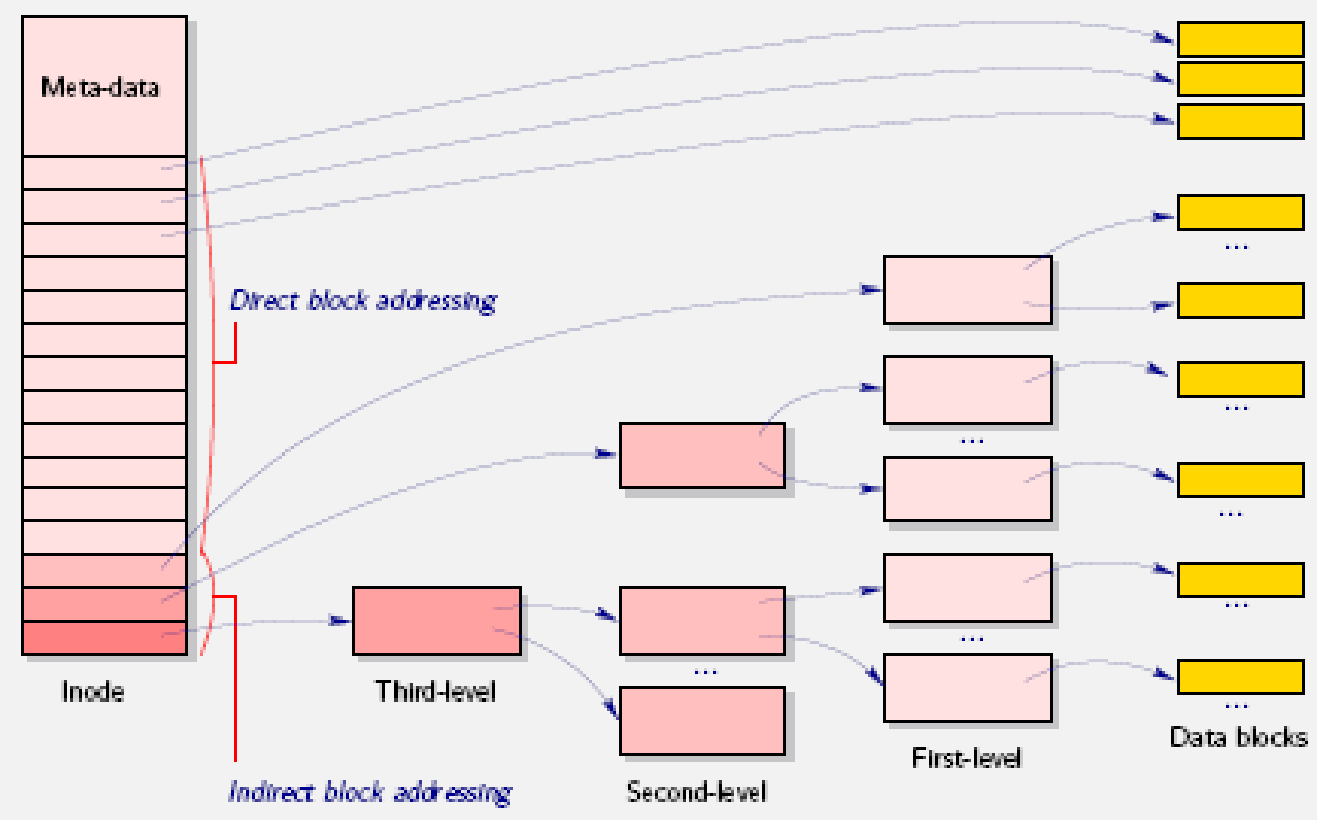


# Fichiers
## Description d'un fichier sous UNIX

- La description d'un fichier est décomposé en deux parties:
  - le contenu fichier
  - les informations sur le fichier (metadata): inode
- **Informations d'un inode:**
  - Type de fichier
  - Nombre de liens durs (hard links) partageant l'inode
  - Longueur du fichier en octets
  - Identifiant de device
  - Identifiant de l'utilisateur (UID) propriétaire, de son groupe (GID)
  - Date de création, modification
  - Droits d'accès

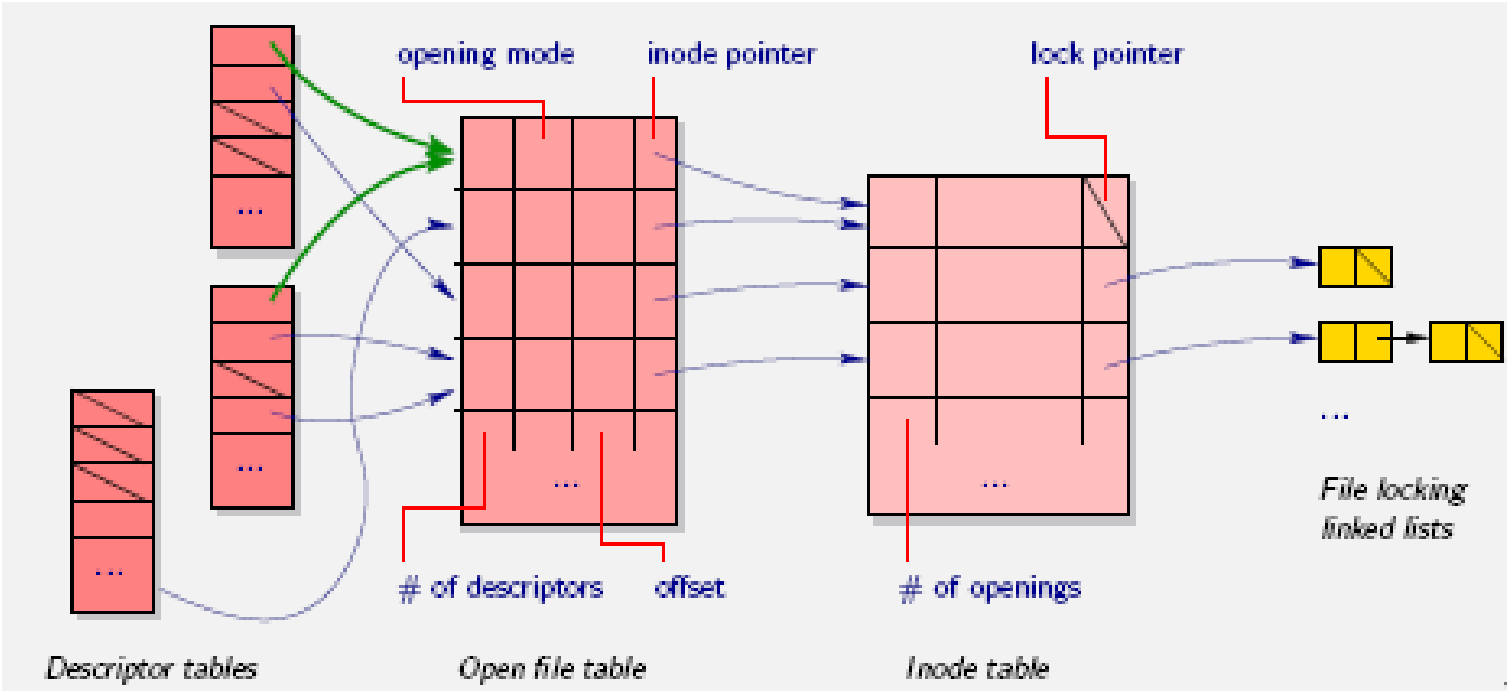
# Fichiers: inodes

**inode:** regroupe informations sur le fichier
table des blocs de données, avec accès directs ou indirects



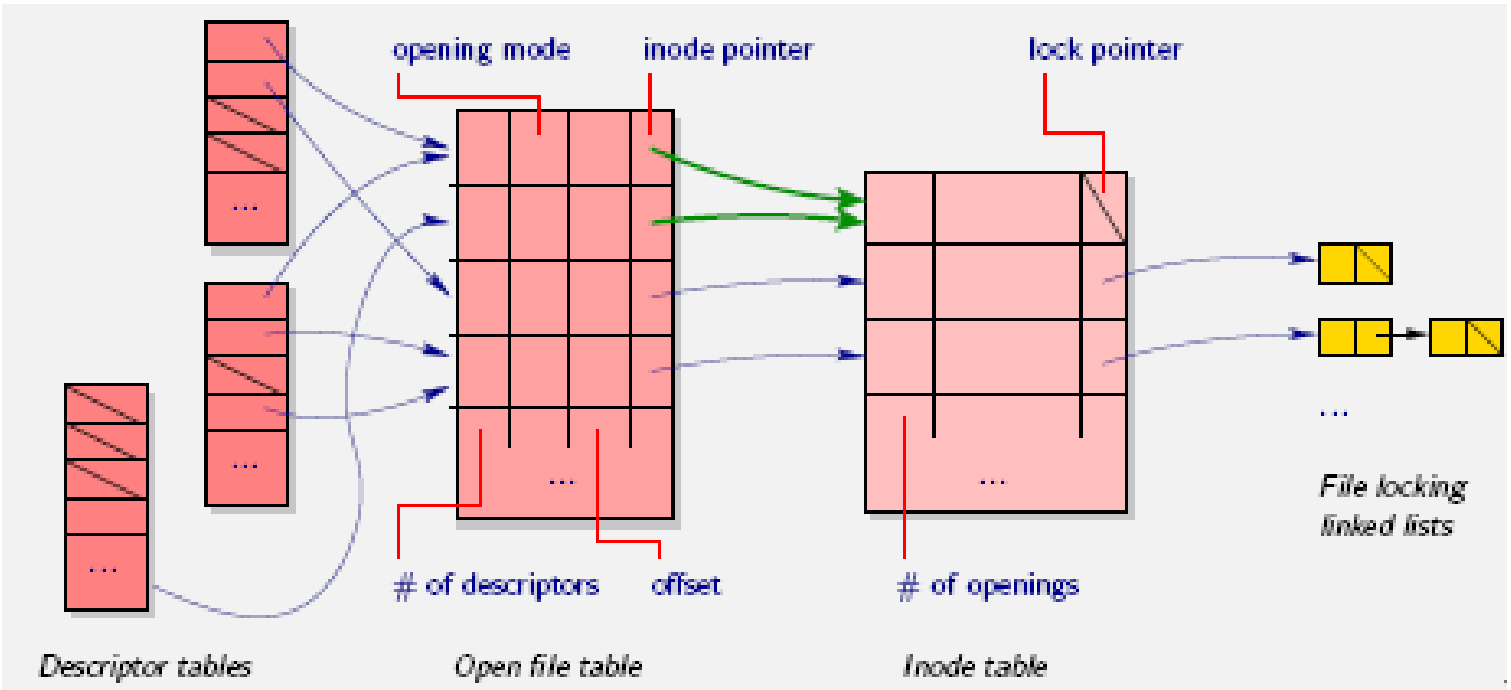
# Fichiers: structure dans un processus

Aliasing sur le descripteur de fichier: par `dup()` ou `fork()`



# Fichiers: structure dans un processus

Aliasing sur les inodes (fichiers ouverts), en faisant de multiples `open()` sur le même fichier.



# Plan

# Communications inter-processus

Quels moyens permettent d'échanger des données?

- Les fichiers normaux.
- Les tubes (ou pipe) ordinaires ou nommés qui sont des mécanismes de communication appartenant au système de gestion des fichiers.
- Les signaux qui peuvent être assimilés à des interruptions logicielles.
- La mémoire partagée qui permet à des processus de partager des mêmes pages physiques en mémoire.
- Les files de messages qui permettent aux processus de s'échanger des informations à travers deux opérations :
  - send (destinataire, message)
  - receive (source, message)

# Plan

# Les tubes
## Présentation (1/2)

- Les tubes sont des mécanismes de communication appartenant au système de gestion des fichiers.
  - Ils sont désignés localement dans les processus par des descripteurs de fichiers.
  - Il sont manipulés au travers des primitives `read` et `write` (comme les fichiers ordinaires).
- Quelques généralités sur les tubes:
  - Les tubes sont des mécanismes de communication unidirectionnels.
  - A un tube correspond **au plus 2 entrées** dans la table des fichiers.
  - L'opération de lecture dans un tube est **destructrice**.
  - Les tubes permettent la communication d'un flot continu de caractères.
  - La gestion des tubes est assurée en mode FIFO.

# Les tubes
## Présentation (2/2)

Il existe deux types de tubes:

- *Les tubes ordinaires*: forcément créés depuis un processus et dont la durée de vie ne peut excéder celle des processus les utilisant.
- *Les tubes nommés*: peuvent être accédé par n'importe quel processus connaissant la référence (son nom) au travers de la primitive `open`.

# Les tubes

## Les tubes ordinaires

```
#include <sys/unistd.h>
int pipe(int fd[2]);
```

- L'appel de la fonction du `pipe` crée un tampon de données, un tube, dans lequel deux processus pourront venir respectivement lire et écrire.
- La fonction fournit, d'autre part, deux descripteurs analogues aux descripteurs de fichiers.
- Les deux processus communicants doivent partager les mêmes descripteurs obtenus par pipe. Il est donc nécessaire qu'ils aient un ancêtre commun car les descripteurs de fichiers sont hérités.

# Les tubes

## Exemple de tube ordinaire (1)

Le processus père envoie l'alphabet à son fils.

```
void main ()
{
  int fd[2]; /* descripteur du tube */
  char c, d;
  int ret;
  pipe(fd);  /* On cree le tube */
  if (fork() != 0) {
    /* On est dans le pere */
    close (fd[0]); /* Le pere ne lit pas */
    for (c='a'; c<='z'; c++) {
      write(fd[1], &c, 1); /* Pere ecrit dans tube */
    }
    close(fd[1]);  /* On ferme le tube en ecriture */
    wait(&ret);    /* Le pere attend le fils */
  } else {
    /* On est dans le fils */
    close(fd[1]);  /* On ferme le tube en ecriture */
    /* Fils lit tube */
    while (read(fd[0], &d, 1) > 0) {
      printf("%c\n", d); /* Il ecrit le resultat */
    }
    close (fd[0]); /* On ferme le tube en lecture */
  }
}
```

# Les tubes

## Exemple de tube ordinaire (2)

Utilisation du pipe par l'interpréteur de commande `cmd1 | cmd2`.

```
void synchro(char * cmd1, char * cmd2) {
  int fd[2];    /* descripteurs du tube */
  pipe(fd);     /* creation du tube */
  if (fork() != 0) {
    close(fd[0]);  /* fermeture lecture */
    /* pas d'ecriture
       d ecriture sur l'ecran
       l ecran (stdout) */
    close(1);
    dup(fd[1]);    /* fd[1] devient sortie standard          */
    close(fd[1]); /* descripteur inutile apres redirection */
    execl( "cmd1", "cmd1", 0 );
  } else {
    close(fd[1]);  /* fermeture ecriture                     */
    close(0);      /* pas de lecture clavier                 */
    dup(fd[0]);    /* fd[0] devient entree standard          */
    close(fd[0]);  /* descripteur inutile apres redirection */
    execl( "cmd2", "cmd2", 0 ;
  }
}
```

# Les tubes
## Les tubes nommés

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *ref, mode_t mode);
```

L'appel de la fonction `mkfifo` crée un tube nommé au niveau du système.

- Le paramètre *ref* définit le chemin d'accès au tube et le paramètre *mode* les droits d'accès.
- L'ouverture d'un tube nommé par un processus se fera au moyen de la primitive `open`.

# Plan

# Les signaux
## Introduction

Les signaux diffèrent des messages:

- Un signal peut être émis à tout moment, occasionnellement à partir d'un autre processus, mais plus souvent à partir du noyau comme conséquence d'un événement exceptionnel.
- Un signal n'est pas nécessairement reçu ni pris en compte.
- Par défaut, la plupart des signaux entraînent la terminaison du processus récepteur.
- Un processus peut ignorer les signaux d'un type donné.
- Les signaux n'ont pas de contenu informatif.
- Le récepteur ne peut déterminer l'identité de l'émetteur.
- Les signaux ne peuvent être envoyés qu'à des processus.

# Signaux
## Envoyer un signal

```
int kill(pid_t pid, int sig);
```

Envoie le signal `sig` au(x) processus `pid`

**Quelle valeur possible pour pid ?**

- $pid > 0$: envoie au processus pid
- $pid = 0$: envoie à tous les processus du groupe du processus courant
- $pid < -1$: envoie à tous les processus du groupe -pid
- $pid = -1$: envoie à tous les processus auxquels on a le droit d'envoyer un signal, sauf soi même et init

**Valeurs de retour**

- 0 si succès, -1 si échec, avec comme errno:
- EINVAL: numéro de signal invalide
- EPERM: pas le droit d'envoyer ce signal à ce processus
- ESRCH: le processus (ou groupe) n'existe pas

# Signaux
## Gestion de la réception d'un signal

- A chaque type de signal est associé, dans le système un handler par défaut désigné symboliquement par SIG_DFL.
- Ce handler définit le comportement par défaut pour chaque type de signal:
  - Terminaison du processus.
  - Terminaison du processus avec un image mémoire (core).
  - Signal ignoré.
  - Suspension du processus
  - Reprise d'un processus stoppé.
  - ...
- Tout processus peut installer un nouveau handler pour chaque type de signal (excepté certains: SIGKILL, SIGSTOP et SIGCONT)

# Signaux
## Installation d'un nouveau handler

```
sig_t signal(int sig, sig_t func);
typedef void (*sig_t) (int);
```

- Un processus peut modifier (*masquer*) son comportement aux signaux reçus par l'appel de la fonction `signal`.
- Installe un nouveau gestionnaire pour le signal *sig*.
- Retourne l'ancien gestionnaire de ce signal (ou SIG_ERR)
- Gestionnaires prédéfinis:
  - SIG_DFL: gestionnaire par défaut
  - SIG_IGN: signal ignoré

# Signaux
## Attente de signaux

```
int pause();
```

- Suspend l'exécution jusqu'à la reception d'un signal:
- Qui termine le processus (pause ne revient pas)
- Qui appelle un gestionnaire de signal
- Les signaux ignorés (SIG_IGN) n'interrompent pas pause.

Retourne toujours -1.

# Signaux

## Programmer une alarme / Mettre en sommeil

```
int alarm(unsigned int s);
```

- Envoie le signal SIGALRM au processus appelant après s secondes.
- Par défaut: le signal termine le processus

```
int sleep(unsigned int s);
```

- Mets le processus en attente pendant s secondes.
- Utilise pause, signal et alarm
  → utilise le même signal qu'alarm, ne pas mélanger les deux.

# Signaux
## Exemple

Terminaison d'un processus si l'opérateur n'a pas répondu dans un délai de 30 secondes.

```
void terminaison (int sig) {
   printf("La reponse n'est pas arrivee a temps \n");
   exit(0);
}
main () {
   int reponse ;
   signal(SIGALRM, terminaison);
   printf("Veuillez repondre:\n");
   alarm(30);
   scanf("%d", &reponse);
   alarm(0);
   printf("Reponse lue \n");
}
```

# Plan

# Mémoire partagée

- Avec les segments de mémoire partagée, les processus partagent des pages physiques par l'intermédiaire de leur espace d'adressage virtuel.
  - Il n'y a plus de recopie des informations.
  - Les pages partagées par les processus sont des ressources critiques dont les accès doivent être protégés.
- Sous UNIX:
  - Il peut y avoir plusieurs segments partagés.
  - Un processus peut accéder à plusieurs segments de mémoire partagée.
  - Les segments de mémoire partagée ont une existence indépendante des processus.
  - Un processus doit rattacher le segment à son espace d'adressage avant de pouvoir y accéder.

# Mémoire partagée
## Deux méthodes de partage

### mmap(2), munmap(2)

- Contenu d'un fichier est copié dans des pages partagées.
- Le fichier sert de point de rendez-vous
- Le contenu des pages est synchronisé avec contenu du fichier
- Persistant au reboot (c'est un fichier)

### shmget(2), shm_open(2), shmat(2), ...

- Un nom de fichier sert de rendez-vous
- Le contenu du fichier n'est pas utilisé,
- Pas de persistance au reboot

# Mémoire partagée
## Création/Modification/Destruction

Création du segment en 3 étapes:

1. Générer une clé à partir d'un nom de fichier: `ftok()`
2. Récupérer l'identifiant de la zone mémoire partagée: `shmget()`
3. Attacher le segment mémoire partagé à l'espace mémoire du processus: `shmat()`

Modification:

- Contrôler le segment mémoire: `shmctl()`

Destruction:

- Détacher le segment mémoire de l'espace mémoire du processus: `shmdt()`
- Détruire le segment mémoire si plus personne ne l'utilise: `shmctl()`

# Mémoire partagée

## Création: 1-Créer une clé à partir d'un fichier

```
key_t ftok(const char *path, int id);
```

- Crée une nouvelle clé à partir
  - D'un fichier existant *path*
  - D'un entier *id* choisi par utilisateur
- Le fichier sert de point de rendez-vous
- Retourne -1 si erreur (pas de fichier ou pas d'accès)

# Mémoire partagée

## Création: 2-Créer ou trouver un identificateur de mémoire partagée

```
int shmget(key_t k,size_t size, int shmflg);
```

- *k* est sa clé (cf `ftok()`)
- *size* spécifie la taille du segment partagé
- *shmflg* donne les flags de création comme les droits d'accès.
- Si la clé est déjà associée à un segment de mémoire
  - Retourne l'identifiant existant
- Sinon, ou si `IPC_CREAT` est dans shmflg, ou si `k=IPC_PRIVATE`
  - Retourne l'identifiant du segment nouvellement créé

# Mémoire partagée

## Création: 3-Attacher un segment de mémoire partagée

```
void *shmat(int shmid, void *shmaddr, int shmflg);
```

Associe le segment de mémoire partagée d'identifiant *shmid* à l'espace d'adressage du processus appelant.

- *shmid*: un identifiant de mémoire (voir `shmget()`)
- *shmaddr*: adresse où devra être placé le segment. Si NULL, l'OS choisit (mieux)
- *shmflg*: 0 ou un *ou binaire* entre plusieurs valeurs possibles dont: SHM_RDONLY: le segment est en lecture seulement

# Mémoire partagée

## Destruction: Détacher un segment de mémoire partagée

```
int shmdt(const void *shmaddr);
```

- Désassocie le segment de l'espace d'adressage de l'adresse virtuelle `shmaddr`
- Le segment mémoire n'est pas détruit et peut continuer à être accédé par d'autres processus
- `shmaddr` doit avoir été obtenu avec `shmat`
- La terminaison d'un processus entraîne le détachement de tous les segments qu'il a préalablement attaché.

# Mémoire partagée

## Modification: Contrôle d'un segment mémoire partagé

```
int shmctl(int shmid, int cmd, struct shmid_ds
                                *buf);
```

- Effectue une opération sur le segment identifié par *shmid*
- Exemples pour *cmd*
  - `IPC_RMID`: désalloue le segment et détruit les données associées
  - `IPC_STAT`: collecte des informations sur le segment et les place dans buf (`man shmctl` pour la description des champs)
  - `IPC_SET`: change les droits d'accès sur le segment
- Retourne -1 en cas d'erreur.

# Mémoire partagée
## Résumé

### Créer des zones de mémoire partagées entre processus

- Avec synchronisation sur un fichier (mmap)
- Avec un point de rendez vous sur un fichier (shmget)

### Pointeurs et mémoire partagée

- Les zones de mémoire partagée commencent à différentes adresses pour chaque processus (mémoire virtuelle)
- Ne pas stocker des pointeurs en mémoire partagée!

# Mémoire partagée

## Example

```
// Create the key
key_t key = ftok("/etc/bash.bashrc", 1);
if ( key == -1 ) { perror("ftok"); exit(1); }

// Get the shmid
id = shmget( key, SIZE, IPC_CREAT|0644 );
if ( id == -1 ) { perror("shmget"); exit(1); }

// Attach
A = shmat( id, NULL, 0 );
if ( A == (void*)-1 ) { perror("shmat"); exit(1); }

// Detach/destroy
shmdt(A);
shctl( id, IPC_RMID, NULL);
```

# Plan

# Les files de messages

## Principe

- Les communications de messages se font à travers deux opérations fondamentales:
  - **send** (destinataire, message)
  - **receive** (source, message)
- Les messages sont de tailles variables ou fixes.
- Les opérations d'envoi et de réception peuvent être:
  - soit directes entre les processus,
  - soit indirectes par l'intermédiaire d'une *boîte aux lettres*.
- Les communications se font de manière synchrone ou asynchrone.

# Les files de messages

## Exemple du producteur-consommateur (1/2)

```
#define N       100 /* Nb emplacements */
#define TAILLE    4 /* taille du message */

typedef int message[TAILLE];

void consommateur (void) {
  int objet, i;
  message m;
  for (i=0; i<N; i++)           /* Envoyer N boites vides au */
     send(producteur,&m);       /* producteur */

  while (1) {
    receive(producteur, &m);    /* attendre un message */
    retirer_objet(&m, &objet); /* retirer l'objet du message*/
    send (producteur, &m);      /* renvoyer un msg vide */
    utiliser_objet(objet);
  }
}
```

# Les files de messages

## Exemple du producteur-consommateur (2/2)

- Si le producteur travaille plus vite que le consommateur:

# Les files de messages

## Exemple du producteur-consommateur (2/2)

- Si le producteur travaille plus vite que le consommateur:
  - Tous les messages se retrouveront pleins et attendront que le consommateur les consomme.
  - Le producteur se bloquera dans l'attente d'un message vide.

# Les files de messages

## Exemple du producteur-consommateur (2/2)

- Si le producteur travaille plus vite que le consommateur:
  - Tous les messages se retrouveront pleins et attendront que le consommateur les consomme.
  - Le producteur se bloquera dans l'attente d'un message vide.
- Si le consommateur est le plus rapide:

# Les files de messages

## Exemple du producteur-consommateur (2/2)

- Si le producteur travaille plus vite que le consommateur:
  - Tous les messages se retrouveront pleins et attendront que le consommateur les consomme.
  - Le producteur se bloquera dans l'attente d'un message vide.
- Si le consommateur est le plus rapide:
  - Tous les messages seront vides et attendront que le producteur les remplisse.
  - Le consommateur sera bloqué tant qu'un message plein ne lui parviendra pas.

# Les files de messages
## Implémentation de files de message

- Les files de messages - `msqid_ds`
- Les types de messages - `msgbuf`
- La création d'une file - `msgget`
- L'envoi de messages - `msgsnd`
- La réception de messages - `msgrcv`
- Le contrôle des files - `msgctl`

# Les files de messages

## La structure msqid_ds

Elle correspond à une entrée dans la table des files de messages.

```
/* Structure of record for one message inside the kernel.
   The type 'struct msg' is opaque.  */
struct msqid_ds
{
  struct ipc_perm msg_perm; /* structure describing operation permission */
  __time_t msg_stime; /* time of last msgsnd command */
  __time_t msg_rtime; /* time of last msgrcv command */
  __time_t msg_ctime; /* time of last change */
  __syscall_ulong_t __msg_cbytes; /* current number of bytes on queue */
  msgqnum_t msg_qnum; /* number of messages currently on queue */
  msglen_t msg_qbytes; /* max number of bytes allowed on queue */
  __pid_t msg_lspid; /* pid of last msgsnd() */
  __pid_t msg_lrpid; /* pid of last msgrcv() */
};
```

# Les files de messages

## La structure msgbuf (1/2)

- La manipulation des files de messages suppose la définition d'une structure spécifique à l'application développée sur le modèle donné par msgbuf.

```
struct msgbuf {
{
   long mtype;       /* Type du message */
   char mtext[5];    /* Texte du message */
};
```

- Le type d'un message doit être strictement positif.
- La suite de la définition de la structure d'un message peut contenir des objets quelconques.

# Les files de messages

## La structure msgbuf (2/2)

- ```
  struct msgbuf_user1 {
  {
     long   mtype;
     float  alpha;
     int    tab[5];
  };
  ```

  Cette structure est ?
  ```

# Les files de messages

## La structure msgbuf (2/2)

- ```
  struct msgbuf_user1 {
  {
    long   mtype;
    float  alpha;
    int    tab[5];
  };
  ```

  Cette structure est ? **correcte**:

- ```
  struct msgbuf_user2 {
  {
    long mtype;       /* Type du message */
    char *mtext;   /* Texte du message */
  };
  ```

  Cette structure est ?

# Les files de messages

## La structure msgbuf (2/2)

- ```
  struct msgbuf_user1 {
  {
    long   mtype;
    float  alpha;
    int    tab[5];
  };
  ```

  Cette structure est ? **correcte**:

- ```
  struct msgbuf_user2 {
  {
    long mtype;       /* Type du message */
    char *mtext;   /* Texte du message */
  };
  ```

  Cette structure est ? **incorrecte**:

# Les files de messages

## La primitive msgget

- Elle permet la création d'une nouvelle file ou la recherche de l'identification d'une file existante **à partir de sa clé**.
- Le paramètre option est construit comme une combinaison des constantes: IPC_PRIVATE, IPC_CREAT, IPC_EXCL et des droits d'accès.
- ```
  #include <sys/types.h>
  #include <sys/ipc.h>
  #include <sys/msg.h>
  int msgget (key_t cle, int option);

  int msqid = msgget( ftok("/etc/bashrc", 1),
                      IPC_CREAT | IPC_EXCL | 0666);
  ```
  ```

# Les files de messages

## La primitive msgsnd

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgsnd(int msqid, const void *p_msg,
           int size, int option);
```

La primitive `msgsnd` envoi dans la file d'identification `msqid` le message `msgbuf` pointé par `p_msg`.

- Le paramètre size correspond à la taille du message.
- La primitive renvoie la valeur 0 en cas de succès et $-1$ si non.
- Le paramètre optionnel peut avoir la valeur IPC_NOWAIT. Dans ce cas si la file est pleine, l'appel à la primitive msgsnd n'est pas bloquant alors qu'il l'est par défaut.
- Attention, un appel bloquant à la primitive msgsnd est interruptible.

# Les files de messages

## La primitive msgrcv

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgrcv(int msqid, void *p_msg, int size,
           long type, int option);
```

La primitive `msgrcv` permet de lire un message dans la file identifiée par `msqid`.

- Le pointeur `p_msg` pointe sur une zone mémoire susceptible de recevoir un message de longueur inférieure ou égale à `size`.
- Le paramètre IPC_NOWAIT permet de ne pas bloquer si la file est vide.
- Le type permet de spécifier le type du message à extraire:
  - $> 0$, le message le plus vieux de type égal à type.
  - 0, le message le plus vieux quel que soit son type.
  - $< 0$ le message le plus vieux du type le plus petit $\leq |type|$
- La valeur renvoyée en cas de réussite est la longueur du message ($-1$ sinon).

# Les files de messages
## La primitive msgctl

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgctl(int msqid, int op,
           /* struct msqid_ds p_msqid */);
```

La primitive `msgctl` permet:

- d'accéder aux informations contenues dans l'entrée de la table des files de messages,
- et d'en modifier certains attributs.

Les valeurs possibles du paramètre `op` sont:

- IPC_RMID suppression de la file de messages.
- IPC_STAT récupération des informations msqid_ds sur la file.
- IPC_SET modification des informations de la file.

# Plan

# Plan

# Objectif de l'exclusion mutuelle

- Comment partager les ressources limitées entre les processus?
- L'exclusion mutuelle a pour but de limiter l'accès à une ressource à un nombre donné de processus (1 ou N)
- On appelle l'accès exclusif à une ressource la **section critique**

# Quelques exemples

- L'accès à un fichier donné ouvert par plusieurs processus
- Le pool d'impression
- Gestion des pages physiques de la mémoire
- ...

# Accès concurrents à une donnée partagée

La plupart du temps, les processus travaillent sur données différentes contenues dans leur espace de mémoire privée

Processus 1:
x = 1;

Processus 2:
y = 2;

# Accès concurrents à une donnée partagée

Que se passe-t-il si on les deux processus trouvent un moyen de partager l'accès à x et y avec le code suivant: (et initialement y=4)

Processus 1:
x = 1 ;
x = y-1;

Processus 2:
y = 2;
y = y*2;

# Accès concurrents à une donnée partagée

→ Non déterministe!

- On parle d'**accès concurrent** sur $y$

# Concurrence
## Section critique

- La partie de programme où peut se produire un conflit d'accès est la **section critique**
- Il faut limiter l'accès à la section critique à un seul processus à la fois: besoin d'un mécanisme d'exclusion mutuelle entre les processus/threads
- Rq: toutes les ressources partagées dans un système d'exploitation ne son sont accessible qu'en exclusion mutuelle.

# Concurrence
## Section critique (2)

Les mécanismes d'exclusion mutuelle garantissent que deux processus n'ont jamais accès simultanément à une section critique, ce qui résout les conflits d'accès mais d'autres conditions sont nécessaires pour permettre un bonne coopération:

1. On ne doit pas faire d'hypothèses sur l'ordre d'exécution des instruction de chaque processus, ou sur la vitesse d'exécution de ceux-ci
2. Un processus suspendu ne doit pas empêcher l'exécution des autres processus
3. Il faut minimiser autant que possible l'attente avant d'antrer en section critique

# Plan

# Solutions logicielles
## Analogie vie courante

- Rend les choses plus faciles à comprendre
- Mais ordinateurs moins intelligents que les personnes

Exemple de coordination:

| Personne 1 | Personne 2 |
|---|---|
| Vérifie frigo | |
| Plus de lait! | |
| Va au supermarché | Vérifie frigo |
| Achète du lait | Plus de lait! |
| Revient | Va au supermarché |
| | Achète du lait |
| | Revient |

# Solutions logicielles
## Analogie vie courante

- Rend les choses plus faciles à comprendre
- Mais ordinateurs moins intelligents que les personnes

Exemple de coordination:

| Personne 1 | Personne 2 |
|---|---|
| Vérifie frigo | |
| Plus de lait! | |
| Va au supermarché | Vérifie frigo |
| Achète du lait | Plus de lait! |
| Revient | Va au supermarché |
| | Achète du lait |
| | Revient |

**Résultat: trop de lait !**

# Exemple
## Solution 1

Verrouille le frigo en partant.

# Exemple
## Solution 1

Verrouille le frigo en partant.
$\rightarrow$ Très contraignant. Empêche l'utilisation du frigo pour prendre un jus d'orange!

Comment implémenter un verrou en mémoire ?

# Exemple
## Solution 2

- Laisser un post-it sur le frigo en partant
- L'enlever en revenant
- Si post-it, ne pas acheter du lait (attendre)

```
if (!lait) {
  if (!postit) {
    postit=true;
    lait++;
    postit=false;
  }
}
```

# Exemple
## Solution 2

- Laisser un post-it sur le frigo en partant
- L'enlever en revenant
- Si post-it, ne pas acheter du lait (attendre)

**Résultat:** Toujours trop de lait!

```
if (!lait) {
  if (!postit) {
    postit=true;
    lait++;
    postit=false;
  }
}
```

# Exemple
## Solution 3

Utiliser des post-its différents par colocataire

- Mettre son post-it
- Si pas de post-it de l'autre, et si plus de lait, en chercher
- Enlever son post-it
- Si post-it de l'autre, attendre.

```
//Personne 1
postit_1=true;
if (!postit_2) {
  // Sec. crit
  if (!lait) lait++;
}
postit_1=false;

//Personne 2:
postit_2=true;
if (!postit_1) {
  // Sec. crit
  if (!lait) lait++;
}
postit_2=false;
```

# Exemple
## Solution 3

Utiliser des post-its différents par colocataire

- Mettre son post-it
- Si pas de post-it de l'autre, et si plus de lait, en chercher
- Enlever son post-it
- Si post-it de l'autre, attendre.

**Résultat:** Possible que personne n'achète du lait!
Ce type de deadlock = famine

```
//Personne 1
postit_1=true;
if (!postit_2) {
  // Sec. crit
  if (!lait) lait++;
}
postit_1=false;

//Personne 2:
postit_2=true;
if (!postit_1) {
  // Sec. crit
  if (!lait) lait++;
}
postit_2=false;
```

# Exemple
## Solution 4

Algorithme de Peterson

- Mettre son post-it
- Ecrire que c'est son tour
- Tant qu'il y a l'autre post-it et que c'est le tour de l'autre, attendre
- Si pas de lait, en prendre
- Enlever son post-it

```
//Personne 1
postit_1 = true; tour=2;
while (postit_2 && tour==2);
if (!lait) lait++;// Sec. crit.
postit_1 = false;

//Personne 2
postit_2 = true; tour=1;
while (postit_1 && tour==1) ;
if  (!lait) lait++; // Sec. crit.
postit_2 = false;
```

# Exemple
## Solution 4

Algorithme de Peterson

- Mettre son post-it
- Ecrire que c'est son tour
- Tant qu'il y a l'autre post-it et que c'est le tour de l'autre, attendre
- Si pas de lait, en prendre
- Enlever son post-it

**Résultat:** correct

```
//Personne 1
postit_1 = true; tour=2;
while (postit_2 && tour==2);
if (!lait) lait++;// Sec. crit.
postit_1 = false;

//Personne 2
postit_2 = true; tour=1;
while (postit_1 && tour==1) ;
if (!lait) lait++; // Sec. crit.
postit_2 = false;
```

# Solutions logicielles

- **Algorithme de Peterson**
  - Peut être étendu à plus de 2 threads/processus
- **Bilan solution logicielle:**
  - Solution relativement compliquée juste pour 2 processus
  - Suppose accès mémoire faits dans l'ordre séquentiel pour chaque processus
    - peut être changé par compilateur si dépendances respectées
    - Peut être changé par le matériel (Out of order execution)
  - Fait de l'attente active ( while (p); ). Très coûteux en temps CPU !

Problème: **Solutions logicielles ne marchent pas!!!**

# Concurrence
## Opération atomiques

- Opérations qui s'exécutent complètement ou pas du tout
- Séquence d'opération indivisible
- N'est pas interrompue en plein milieu par une interruption (ordonnancement d'un autre processus ou thread par exemple)
- Opérations atomiques usuelles:
  - un accès mémoire (pour la longueur native des variables)
  - une opération de calcul (correspond à une instruction asm)
- Exemple d'opérations non atomiques:
  - Certains opérateurs de calculs (divisions par exemple, prend plusieurs instructions asm)
  - Accès à une variable 64 bits sur machine 32 bits

# Plan

# Solutions *matérielles* pour la synchronisation

- Verrous (pthread)
- Sémaphores (POSIX, IPC)
- Moniteurs et variables conditionnelles (Java)

# Verrous

## Solution 1 au problème du lait

Si on peut mettre un verrou uniquement pour le lait, solution facile:

- *Verrouille()*: attend tant que verrou mis, ensuite met le verrou
- *Deverrouille():* enlève le verrou et reveille ceux qui attendent

Protège accès section critique (1 thread à la fois)

```
//Thread 1 et 2:
verrou_lait.verrouille();
// Section Critique
if (!lait) lait++;
verrou_lait.deverrouille();
```

Verrouille et deverrouille doivent être atomiques

# Verrous

## Séquence atomique d'instructions

- Test & set:
  - Met le verrou et vérifie s'il y était déjà
- Sur la plupart des architectures:
  - Une instruction *asm*
- Variantes:
  - Permute deux valeurs dont une est en mémoire (x86)
  - Compare et swap (68xxx)

```
int verrou=0;
test&set() { // Atomique
    resultat = verrou;
    verrou = 1;
    return resultat;
}
```

# Verrous
## Utilisation du verrou

- Si verrou pas mis, *test&set* lit 0 et le met. Le while quitte.
- Si verrou mis, *test&set* lit 1 et le met à 1 (ne change rien). Le while boucle.

Attente active: consomme des cycles en attendant

```
int verrou=0;
test&set(p) { // Atomique
    resultat = *p;
    *p = 1;
    return resultat;
}
verrouille() {
  // Attente active
  while (test&set(&verrou));
}
deverouille() {
  verrou = 0;
}
```

# Verrous
## Attente active

- Consomme des cycles en attendant
- Ralentit autres threads/processus qui ne sont pas en attente
- Inversion des priorités: si thread en attente active a priorité plus haute que le thread ayant le verrou $\rightarrow$ rien n'avance!

- Possible de faire un verrou sans attente ? **Non**
- Possible de faire un verrou sans attente active ? **En partie**

# Verrous

## Attente active limitée

```
verrouille() {
  // Courte attente active
  while (test&set(&porte));
  if (verrou == OCCUPE) {
    PutThrdInWaitingList;
    porte = 0 & sleep();
  } else {
    verrou = OCCUPE;
    porte = 0;
  }
}
```

```
deverrouille() {
  if (un thread dors) {
    RemoveThrdFromWaitingList;
    PutThrdInActiveList;
  }
  else {
    verrou = LIBRE;
  }
  porte=0;
}
```

- Idée: Faire de l'attente active uniquement pour vérifier atomiquement la valeur de lock
- C'est la solution fournie par les systèmes au travers des IPC (Inter Process Communication)

# Verrous
## Programmation

Utilisation d'une variable partagée:

- Pour verrouiller: on met la variable à 0
- Pour déverrouiller: on met la variable à 1

# Plan

# Sémaphores
## Introduction

- Sémaphores généralisent les verrous
- Inventés par Dijkstra à la fin des années 1960
- Principale primitive de synchronisation à l'origine dans UNIX
- **Définition**: un sémaphore est un entier non négatif qui a deux opérations
  - `P()` : opération atomique qui attend que le sémaphore soit positif et le décrémente de 1.
    *Proberen* (tester en neerlandais) = puis-je ?
  - `V()` : opération atomique qui incrémente le sémaphore de 1, et réveille les threads/processus bloquant sur P(). *Verhogen* (incrémente en néerlandais) = vas-y

# Sémaphores

- Comme des entiers $>= 0$
- Seules opérations:
  - initialisation lors de la création
  - P(), atomique
  - V(), atomique

**Valeur initiale = 1**

$\rightarrow$ Sémaphore = verrou

```
semaphore.P();
// section critique
semaphore.V();
```

# Sémaphores

- Comme des entiers >= 0
- Seules opérations:
  - initialisation lors de la création
  - P(), atomique
  - V(), atomique

**Valeur initiale = 0**

→ Permet d'attendre un autre thread.

Exemple d'attente d'un thread devant lors de la terminaison d'un autre thread:

```
Threadjoin() / wait() {
  semaphore.P();
}
Threadexit() / exit() {
  semaphore.V();
}
```

# Sémaphores

**Valeur initiale** $k > 1$

$\rightarrow$ Utilisé pour limiter accès à une ressource par $k$ threads au plus.

```
semaphore.P();
// k threads ici au plus en meme temps
semaphore.V();
```

Utile si ressource en $k$ exemplaires seulement (processeur, périphériques, ...).

# Sémaphores
## Exemple producteur/consommateurs


**Définition du problème:**

- Un producteur écrit des données dans un buffer partagé (taille limitée)
- Un consommateur les enlève du buffer
- Besoin d'une synchronisation entre consommateur et producteur
  - Le producteur doit attendre si buffer plein
  - Le consommateur doit attendre si buffer vide
- Buffer = ressource partagée modifiée.
  $\rightarrow$ Besoin section critique

Exemples:

- séquence de pipe, gcc (cpp — cc1 — cc2 — as — ld)
- filtres sur des images

# Sémaphores
## Programmation

### Création d'un sémaphore

```
sem_t *sem_open(char *name, int flags);
sem_t *sem_open(char *name, int flags, mode_t mode,
unsigned int value);
```

Arguments:

- Seulement quelques valeurs possibles de flags parmi celles du open: O_CREAT et O_EXCL pour flags
- name: nom utilisé pour que d'autres processus utilisent le même sémaphore
- value: valeur d'initialisation du sémaphore. Par défaut, 1.

Retourne -1 si une erreur.

# Sémaphores
## Programmation

**Incrémente un sémaphore, V()**

```
int sem_post(sem_t *s);
```

Débloque le sémaphore et l'incrémente de 1.

**Décrémente un sémaphore, P() et bloque processus si nécessaire**

```
int sem_wait(sem_t *s);
```

Bloque si la valeur est $<= 0$. Le processus est débloqué si la valeur $> 0$.
Retourne $-1$ si erreur.

**Désalloue les ressources du sémaphore:**

```
int sem_close(sem_t *s);
```

# Plan

# Deadlocks
## Introduction

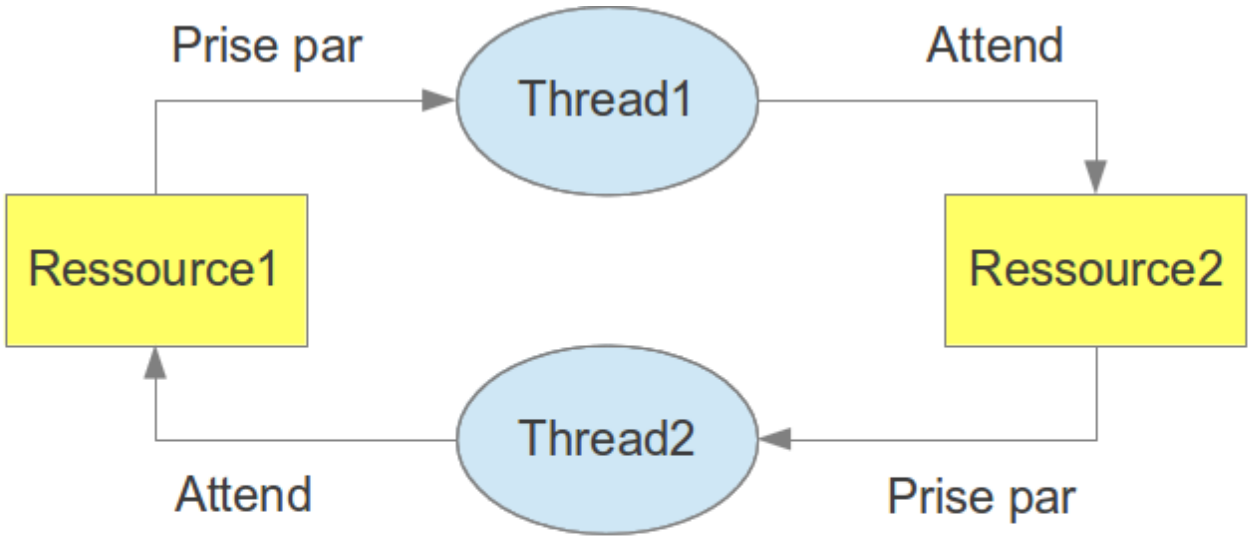
**Ressources**: entités nécessaires aux threads pour leur exécution

- Processeur, temps, espace disque, mémoire, périphérique
- Certaines ressources sont partageables entre threads, d'autres non (nécessite section critique):
  - Imprimante non partageable pendant l'impression
  - Fichiers en lecture seule sont partageables
- Ressources préemptables: on peut les retirer aux threads

Un but essentiel des systèmes d'exploitation est de gérer les ressources

# Deadlocks
## Introduction

**Deadlocks**: situations d'inter-blocage dû au partage de ressources entre threads/processus.




Le thread1 a pris la ressource 1 et attend la ressource 2,
Le thread2 a pris la ressource 2 et attend la ressource 1

# Deadlocks

## Conditions pour un deadlock

- Dépend de l'exécution, pas déterministe

| Thread 1 | Thread 2 |
|---|---|
| x.P(); | y.P(); |
| y.P(); | x.P(); |
| // deadlock free | // deadlock free |
| y.V(); | x.V(); |
| x.V(); | y.V(); |

  *Un deadlock ne se produit pas toujours pour ce code.*
- Doit impliquer de multiples ressources
  - Impossible de débugger pour chaque ressource indépendamment
- Pas de débloquage sans intervention extérieure

# Deadlocks
## Quelques exemples

- **Carrefour avec priorité à droite**
  - Chaque voiture possède un segment de route et veut acquérir le segment suivant
  - Pour traverser, doit avoir deux segments
- **Résolution du deadlock**
  - Une voiture fait marche arrière (rollback). Plusieurs voitures devront reculer éventuellement.
  - Une voiture est enlevée (préemption). Problème si la voiture enlevée a une autre ressource.
- **Famine possible:** impossibilité de traverser si beaucoup de trafic dans une direction.

# Deadlocks

## Dîner des philosophes (Djikstra, Hoare)

- 5 fourchettes pour 5 personnes
- Besoin de 2 fourchettes pour manger
- Philosophes prennent celle de droite puis de gauche
- $\rightarrow$ Deadlock possible

**Prévention:**

- Un des philosophes repose une fourchette
- Ne jamais prendre la dernière fourchette si aucun philosophe affamé n'a 2 fourchettes après

Famine possible.

# Deadlocks

## 4 Conditions pour un deadlock

1. Exclusion mutuelle
2. Les threads/processus attendent en possédant des ressources
3. Pas de préemption
   - Les ressources ne sont relâchées que volontairement par un thread/processus
4. Attente circulaire
   - Il y a un ensemble T1..Tn de threads en attente, avec
     T1 attend une ressource prise par T2
     T2 attend une ressource prise par T3
     ...Tn attend une ressource prise par T1

# Deadlocks
## Empêcher les deadlocks

- **Éliminer les symétries ou attentes circulaires**
  - Pas toujours possible ni souhaitable
  - Force un ordre pour prendre les ressources
- **Utiliser préemption**
  - Risque d'état incohérent (tue un processus par ex.)
- **Détecte situation potentielle de deadlock**
  - Construit graphe d'utilisation des ressources
  - Détecte les cycles (algorithme d'allocation du banquier)
- **Imposer des timeouts**
  - Pas de garantie en soit que deadlock ne se reproduira pas
- **Détection (par autre thread) de deadlock**
  - Rollback possible.

# Deadlocks
## Conclusion

- **Problèmes de concurrence**
  - Non déterminisme des calculs
  - Inter-blocages, famines
- **Solutions**
  - Tout accès en écriture d'une ressource partagée
    $\rightarrow$ dans section critique
  - Exclusion mutuelle logicielle possible, si accès mémoire atomique
  - Solution matérielle préférable (verrous, sémaphores)
- **Deadlocks**
  - Bug difficile à reproduire
  - Nécessite 4 conditions pour se produire