

# Les Systèmes d'Exploitation



# Objectifs de ce cours

- Comprendre comment fonctionne un SE
  - Ce qu'il gère
  - Comment il le gère
- Savoir utiliser les possibilités du SE dans les programmes
  - Utiliser les ressources
  - Exploiter le parallélisme (multiprocesseur / multithreading)
  - Faire communiquer des processus
  - Synchroniser des processus

# Les Systèmes d'Exploitation

## **1. Introduction**

2. Les Processus
3. Les Threads
4. Concurrency entre processus
5. Communication entre processus
6. Les interruptions logicielles (signaux)
7. Gestion du CPU et de la mémoire
8. Ordonnancement
9. Les Entrées/Sorties
10. Le système de fichiers
11. Démarrage d'un système
12. Interblocage
13. Sécurité
14. Systèmes multiprocesseurs

# Définitions possibles

- Machine étendue :

Machine + SE = **machine plus simple à utiliser** c'est-à-dire offrant des services pour la mémoire, les E/S, ...

- Gestionnaire de ressources :

**Partager les ressources** dans le temps et l'espace

- **dans le temps** par exemple partager le CPU, l'UE/S
- **dans l'espace** par exemple partager la mémoire, le disque, ...

# Point de vue historique

- Au début les SE étaient **mono processus** car on faisait surtout du calcul donc on utilisait le CPU au maximum.
- Plus tard, **programmes interactifs** et avec fichiers => beaucoup d'attente sur les E/S => le CPU est sous utilisé => changer de processus quand le processus courant est en attente = **multiprogrammation**.
- Puis **temps partagé** (le 1<sup>er</sup> est **CTSS** – Compatible Time Sharing System – du MIT - 1961) qui devient **MULTICS** (MULTIplexed Information and Computing System -1964) qui donnera **UNIX** (1978) puis **LINUX** (1991).
- Sur les microordinateurs **CPM** (1978) est *mono processus*, **MPM** (1979) est *multi processus*. Avec le PC retour en arrière avec **MSDOS** (1981) qui est *mono processus* puis **Windows** (1985) qui est *multi processus*.
- Puis les machines deviennent multiprocesseurs => le partage du CPU se fait dans le temps **ET** dans l'espace.

# Complexité

- Logiciels très complexes (plusieurs dizaines de millions de lignes de code)
- Croissance : Windows 3.1 en 1992 : 2,5M, Windows NT4 en 1996 : 11M, Windows 7 en 2009 : 37,5M , Windows 10 en 2015 : 50M
- Fonctionnent en permanence
  - Probabilité d'erreur élevée en cas de bugs
- Fonctionnement imprévisible (événements)
  - Actions de l'utilisateur
  - Réseau
  - Processus qui se terminent, qui démarrent, qui se plantent ...
  - Défauts mémoire
  - Erreurs
  - ...
- Points d'entrée des attaques (doivent être sécurisés)

# Ce que fait un SE

- Le SE doit **gérer** :
  - la mémoire
  - les entrées/sorties
  - les processus
  - les fichiers
  - Les utilisateurs
  - La sécurité
- Les **outils** sont :
  - la mémoire virtuelle pour la mémoire
  - les interruptions 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 processus
  - synchronisation des processus.

# Primitives Système

- Le SE propose des **primitives** (fonctions) que tout programme peut appeler pour utiliser les ressources. En général on trouve :
  - 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, BâL, tubes ...)
  - gestion des E/S (écran graphique, souris, clavier ...)
  - gestion du réseau (sockets)
- Pour UNIX il existe la norme **POSIX** (ISO 9945-1) qui définit des appels standards.
- Pour Windows c'est **l'API win32s** et win32 pour 64 bits.



# Structure d'un SE

- **Monolithique**

Ensemble de **fonctions** qui s'appellent entre elles et que le programmeur peut aussi appeler. Certaines fonctions auxiliaires ne sont pas documentées => le programmeur ne peut pas les utiliser (sauf à trouver l'info). C'était le cas de MSDOS et des 1<sup>ers</sup> Windows.

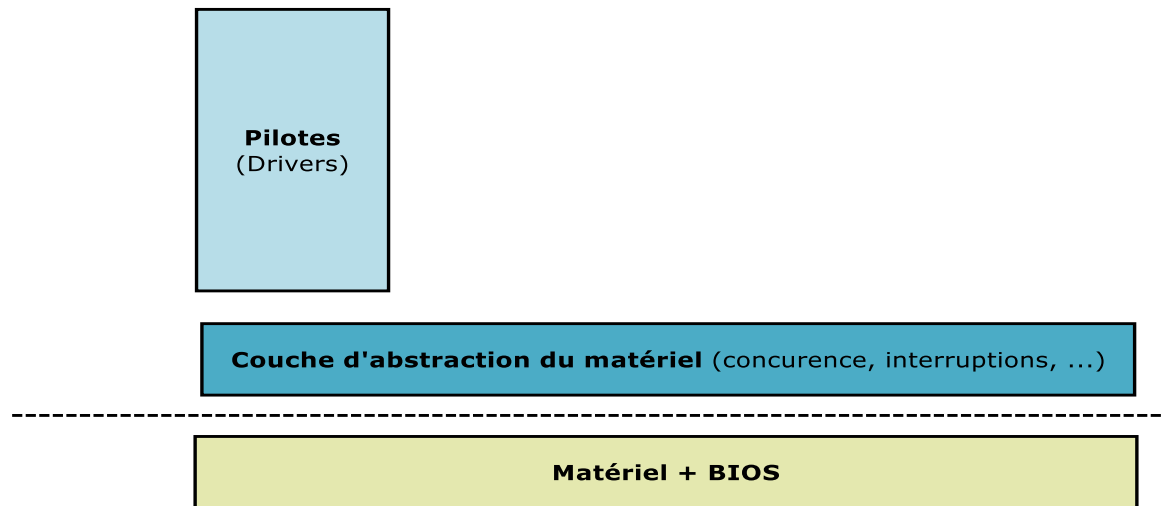
- **Système en couches**

Par exemple :

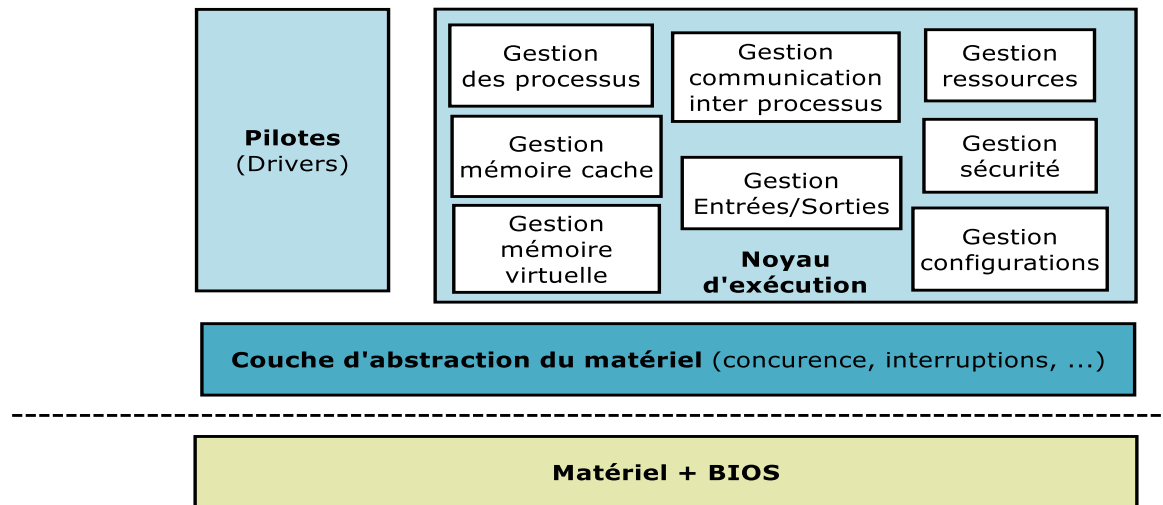
- gestion du CPU
- gestion de la mémoire
- communication entre processus
- gestion des E/S
- programmes

C'est le cas des SE actuels (UNIX, Windows depuis NT).

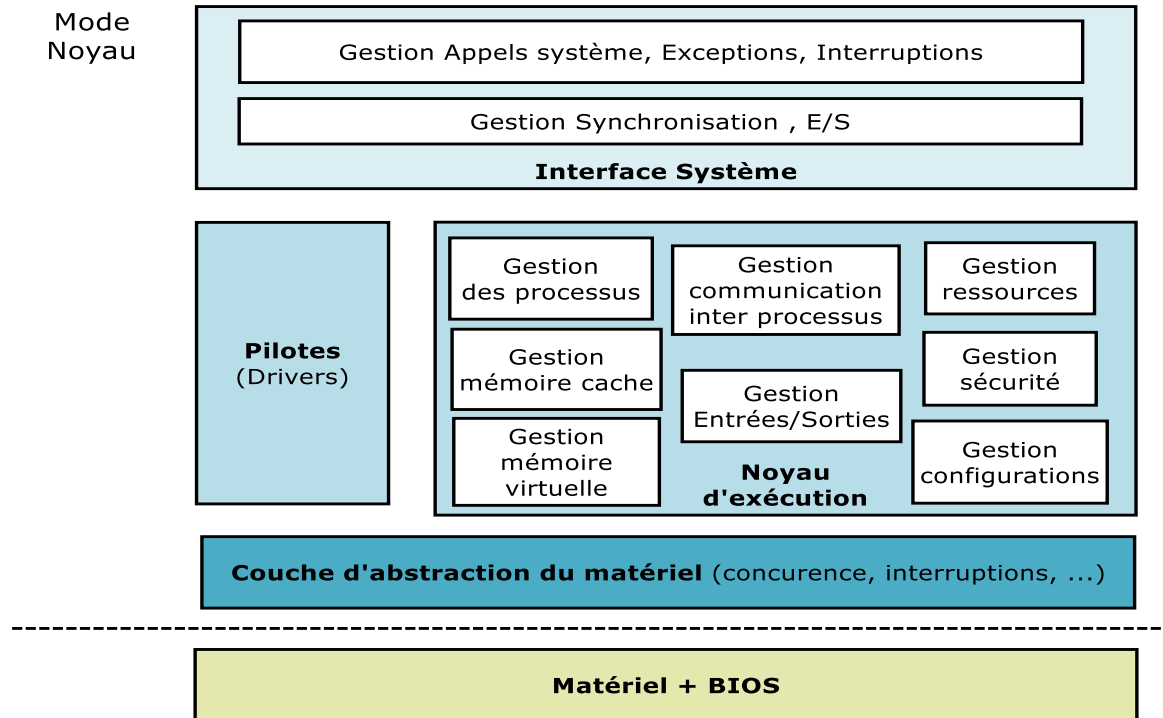
# Couches de Windows



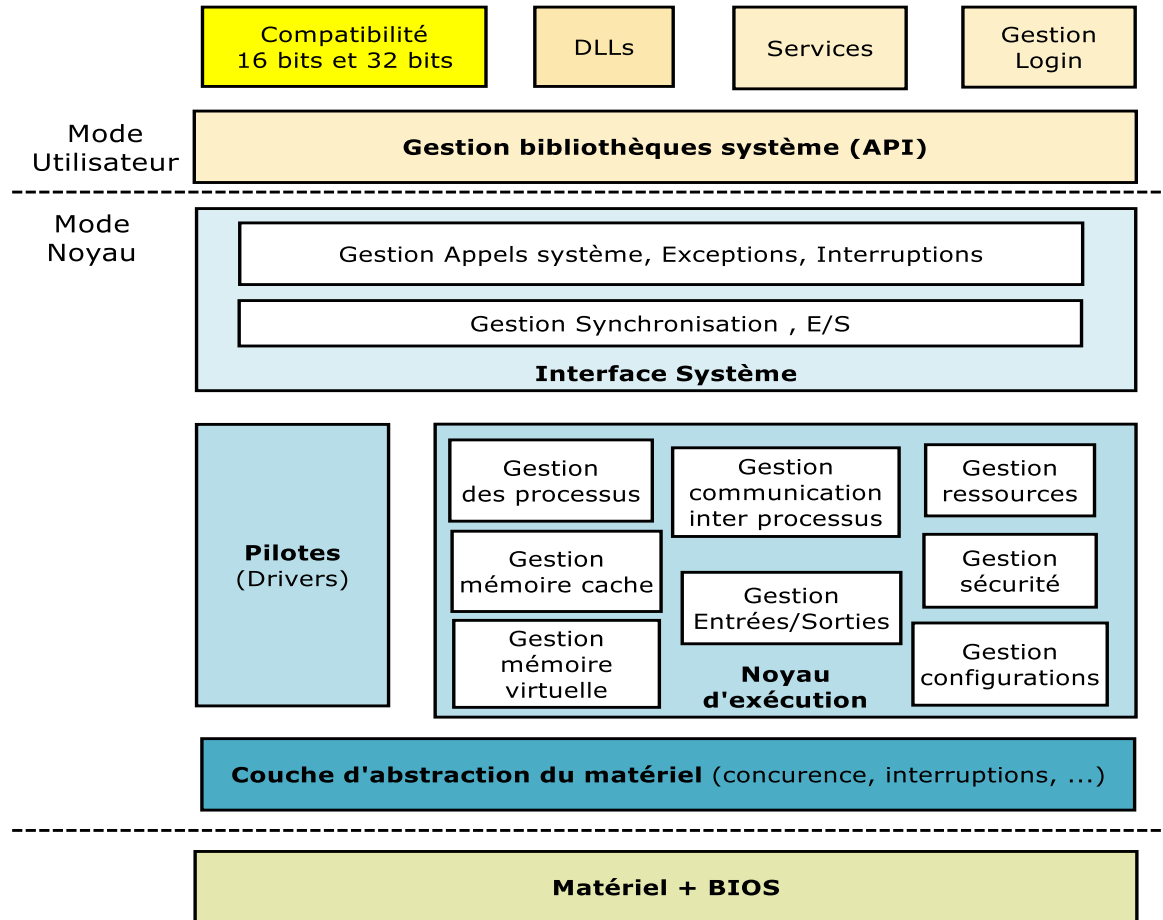
# Couches de Windows



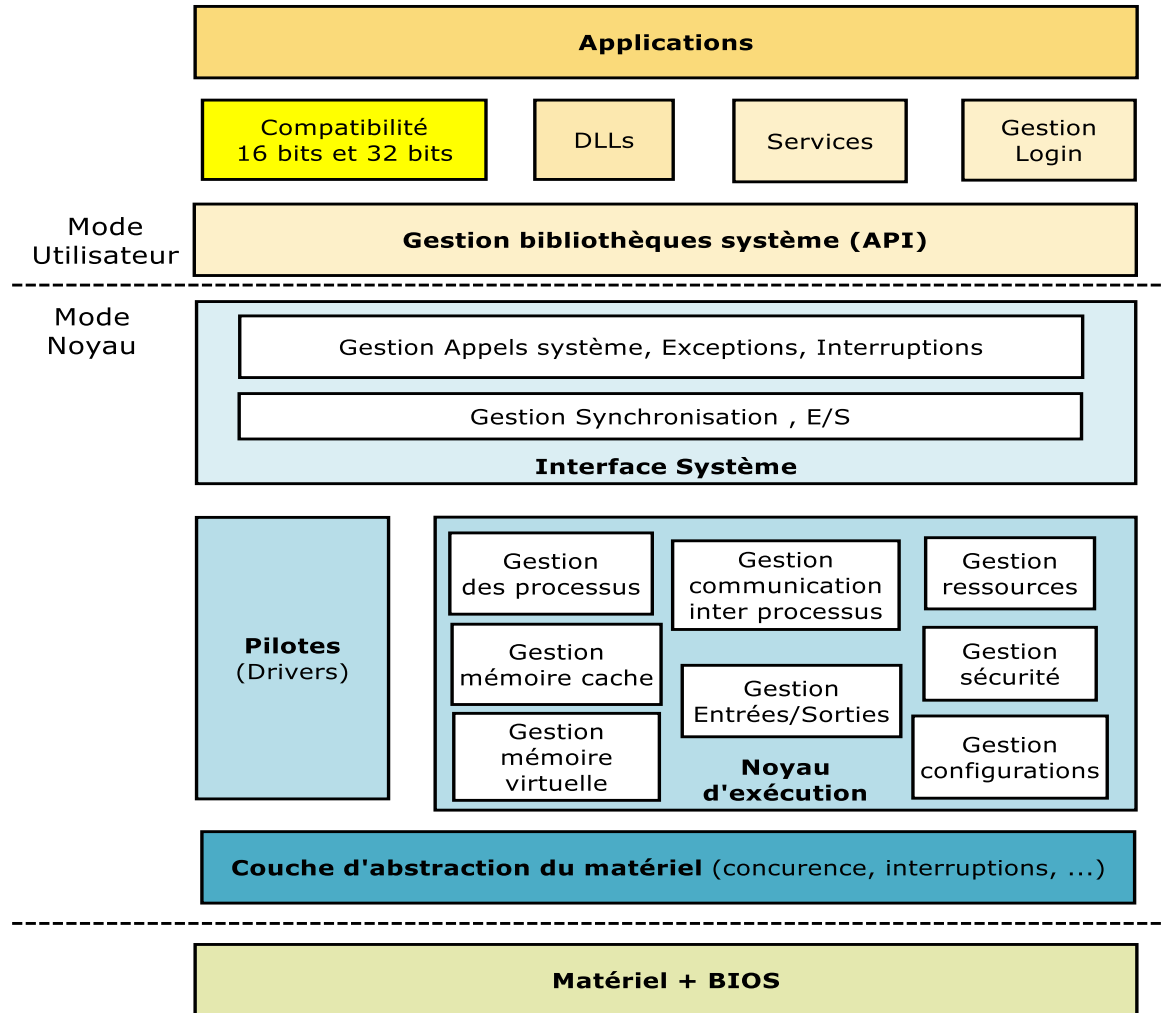
# Couches de Windows



# Couches de Windows



# Couches de Windows



# Structure d'un SE

- **Machines virtuelles**

Un **hyperviseur** définit une machine virtuelle qui est une **vision du matériel** se comportant comme une machine classique. Sur cette machine virtuelle on installe un SE complet qui la voit comme sa machine réelle.

⇒ On peut définir **plusieurs machines virtuelles** sur le même matériel et donc y faire fonctionner **simultanément plusieurs SE**.

*Exemples connus* : MSDOS sous Windows, VMWARE, XEN, java ...

- **Modèle client/serveur**

Les appels système ne sont plus des appels directs mais des **communications C/S**. Le programme qui veut quelque chose devient client d'un service du SE qui fait le travail et lui renvoie le résultat

Dans ce modèle le SE est composé d'un **micronoyau** minimal qui gère les accès direct au matériel et de **services** qui font tout le reste

*Avantage* : adapté aux systèmes distribués.

# Les Systèmes d'Exploitation

1. Introduction
- 2. Les Processus**
3. Les Threads
4. Concurrency entre processus
5. Communication entre processus
6. Les interruptions logicielles (signaux)
7. Gestion du CPU et de la mémoire
8. Ordonnancement
9. Les Entrées/Sorties
10. Le système de fichiers
11. Démarrage d'un système
12. Interblocage
13. Sécurité
14. Systèmes multiprocesseurs



# Notion de Processus

Un ordinateur peut faire **plusieurs choses en parallèle** par exemple du traitement et des E/S car l'UE fonctionne en parallèle du CPU.

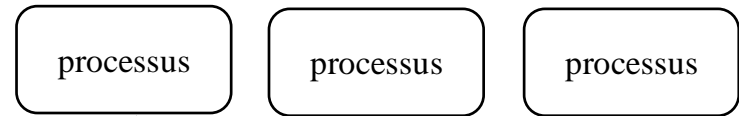
Lorsque l'on **dispose de plusieurs CPUs** (et/ou de plusieurs cœurs) on exécute plusieurs programmes (**processus**) en parallèle et c'est un **vrai parallélisme**.

Lorsque l'on **partage un CPU (ou un cœur) dans le temps** on exécute plusieurs programmes (**processus**) en même temps mais c'est un **faux parallélisme** car le CPU (ou le cœur) n'en exécute qu'un à la fois.

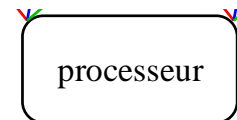
Remarque : les mécanismes mis en œuvre entre les processus en pseudo parallélisme sont applicables à du **vrai parallélisme**  $\Rightarrow$  les SE multiprocessus fonctionnent de façon identique en vrai parallélisme (plusieurs CPU).

# Principe de Processus

Un **processeur** gère N **processus** et bascule de l'un à l'autre sans arrêt.



Chaque processus dispose d'un processeur virtuel représenté par un **contexte** où il laisse son état pour le retrouver quand vient son tour.



Le principe est le même s'il y a **plusieurs processeurs** (chacun gère plusieurs processus)

# Types de processus

- **Processus interactifs**

La plupart des processus s'exécutent en utilisant des **E/S** en lien avec un utilisateur  $\Rightarrow$  beaucoup d'attentes

- **Processus non interactifs**

Certains processus gèrent en **arrière plan** des choses sans interaction (par exemple le processus qui reçoit les mails et les met dans la boîte à lettres). On parle de **démons**.

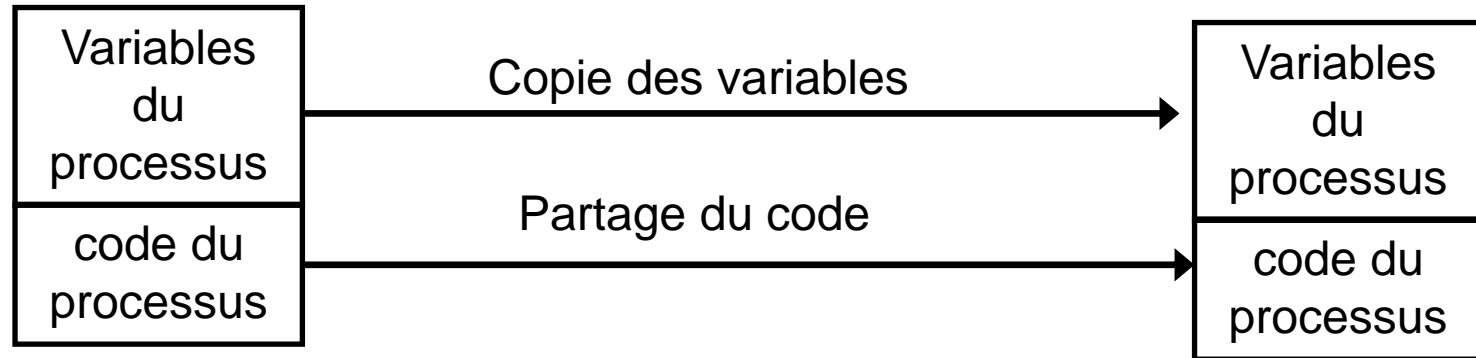
Certains processus sont créés au démarrage du SE et **tournent indéfiniment** d'autres **sont créés à la demande** d'un utilisateur ou d'un autre processus et se terminent au bout d'un moment.

# Création de processus

- **UNIX** utilise un appel système **fork** qui crée un **clone du processus** qui s'exécute après duplication de l'espace mémoire et du contexte.  
Le processus initial est appelé **père** et son clone est le **fils**.  
La fonction **fork** retourne une valeur différente à chacun des 2 processus  $\Rightarrow$  en testant cette valeur on peut différencier le comportement des 2 clones. Cette approche permet de créer des processus **coopérants** :
  - Le fils **hérite d'une copie des variables** du père (incluant les descripteurs de fichiers ouverts, ...)
  - Il peut **partager un travail** avec son père.
  - Un processus peut utiliser un appel système pour **charger un nouveau programme à sa place**  $\Rightarrow$  il modifie l'image mémoire et la remplace par celle du nouveau programme.
- **WINDOWS** utilise un appel système **CreateProcess** qui crée un processus et y charge un programme

Remarque : fork est sans paramètre, CreateProcess a 10 paramètres.

# Principe de fork



```
int main(...) {
```

```
...
```

```
fork()
```

```
...
```

```
}
```

```
Retourne p
```

```
Retourne 0
```

```
fork()
```

```
...
```

```
}
```

**On a maintenant 2 processus**

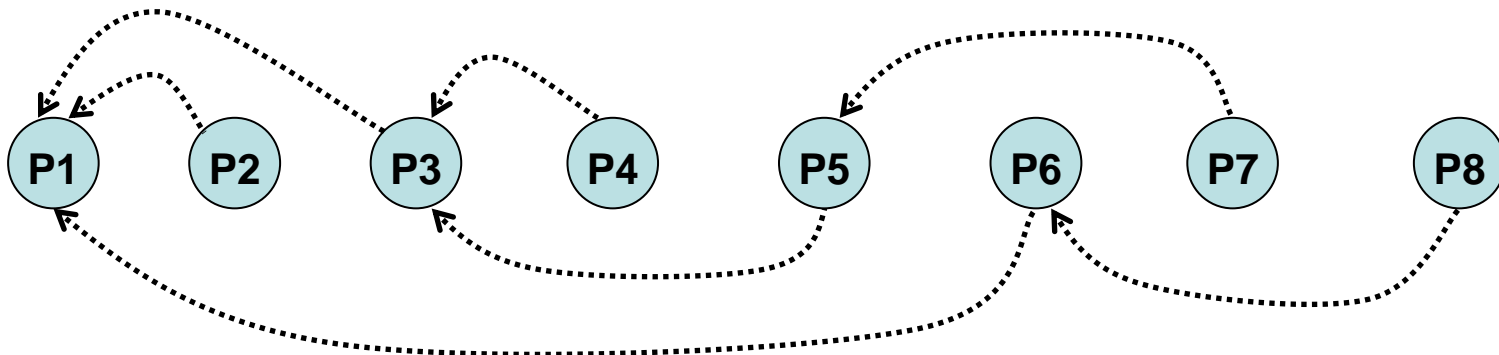
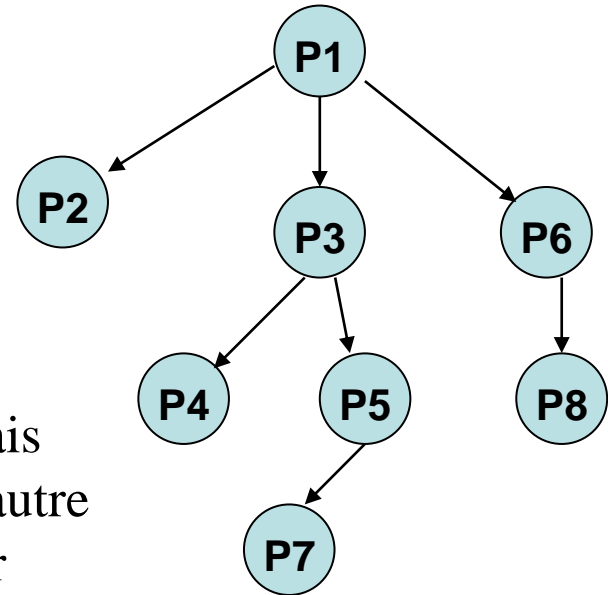
# Arrêt de processus

- Arrêt **normal** par appel d'une primitive système de fin de processus
- Arrêt sur **erreur volontaire** par appel d'une primitive système
- Arrêt sur **erreur involontaire** ou sur erreur fatale : le SE décide d'arrêter le processus
- Arrêt par un **autre processus** ou par l'**utilisateur**

Remarque : quand un processus s'arrête, le SE peut arrêter tous les processus qu'il a créé ou pas (UNIX et Windows ne le font pas).

# Hiérarchie de processus

- Sous **UNIX** tout processus a un père  $\Rightarrow$  arbre
- Sous **UNIX** tout processus a un numéro (**pid**)
- Sous **Windows** ils ont un numéro (**pid**) mais sont « à plat ». Le processus qui en crée un autre récupère une info qui lui permet de le piloter (l'arrêter par exemple).



# Vue des processus (UNIX)

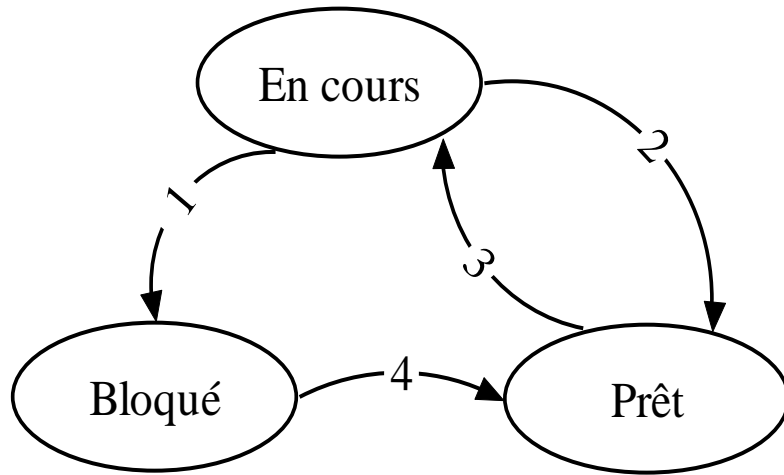
- Par la commande ps :
  - USER : utilisateur ayant lancé le processus
  - PID : n° de processus
  - %CPU : % d'utilisation du CPU
  - %MEM : % d'utilisation de la mémoire
  - VSZ : utilisation des bibliothèques partagées
  - RSS : utilisation de la mémoire physique (en Ko)
  - STAT : état du processus
  - START : date de démarrage du processus
  - TIME : temps CPU utilisé par le processus
  - CMD : nom du programme
  - ...



# Vue des processus (Windows)

Image	PID	Description	Statut	Threads	Processeur	UC moyenne
perfmon.exe	3236	Moniteur de ressources et de performances	En cours d'exécution	19	1	2.27
sidebar.exe	4164	Gadgets du Bureau Windows	En cours d'exécution	26	1	0.94
TeaTimer.exe	4156	System settings protector	En cours d'exécution	4	0	0.47
taskmgr.exe	3060	Gestionnaire des tâches de Windows	En cours d'exécution	6	0	0.38
dwm.exe	1248	Gestionnaire de fenêtres du Bureau	En cours d'exécution	5	0	0.28
ApntEx.exe	4408	Alps Pointing-device Driver for Windows NT/2000/XP/Vista	En cours d'exécution	4	0	0.28
Interruptions système	-	Appels de procédure différés et routines du service d'interru...	En cours d'exécution	-	0	0.24
System	4	NT Kernel & System	En cours d'exécution	133	0	0.13
csrss.exe	568	Processus d'exécution client-serveur	En cours d'exécution	11	0	0.12
explorer.exe	1676	Explorateur Windows	En cours d'exécution	34	0	0.12
Apoint.exe	3904	Alps Pointing-device Driver	En cours d'exécution	4	0	0.06
ApMsgFwd.exe	4292	ApMsgFwd	En cours d'exécution	2	0	0.05
services.exe	616	Applications Services et Contrôleur	En cours d'exécution	7	0	0.03
svchost.exe (LocalServic...	996	Processus hôte pour les services Windows	En cours d'exécution	20	0	0.03
LMS.exe	5912	Local Manageability Service	En cours d'exécution	8	0	0.01
UNS.exe	5940	User Notification Service	En cours d'exécution	18	0	0.01

# États d'un processus



Transitions :

1. Le processus est **bloqué** en attente de quelque chose (par ex une E/S)
2. Le processus est **suspendu** par le SE qui passe à un autre
3. Le processus est **relancé** par le SE qui vient de le choisir
4. Ce que le processus attendait est arrivé

Le changement d'état d'un processus se fait :

- **lors d'un appel de primitive système** (1)
- **lors d'une interruption** (2, 3 et 4)

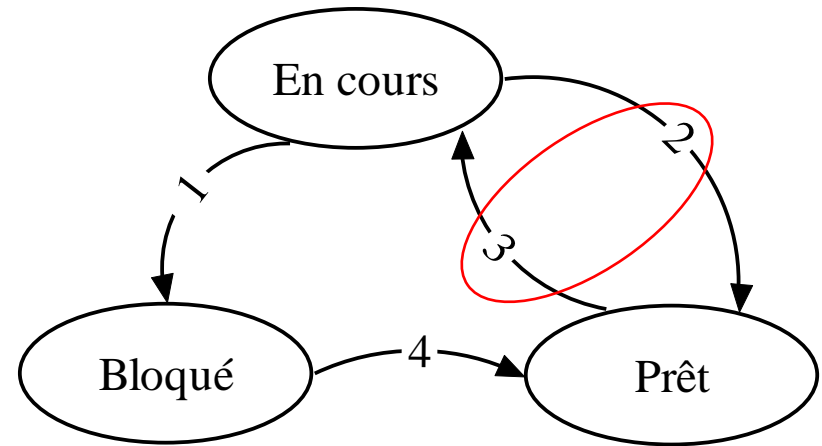
Exemple :

1. Un processus appelle une primitive pour lire sur disque  $\Rightarrow$  transition 1
2. Une IT indique que la lecture disque est terminée  $\Rightarrow$  transition 4

# Temps partagé

Le **temps partagé** est obtenu en utilisant un **timer** pour générer des ITs à intervalles réguliers

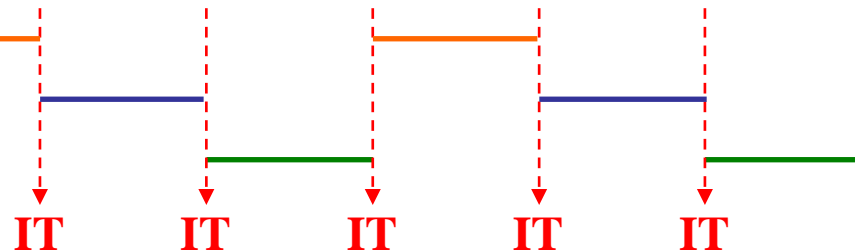
Le SE provoque des transitions 2 et 3  
⇒ exécution en pseudo parallélisme



Processus A

Processus B

Processus C

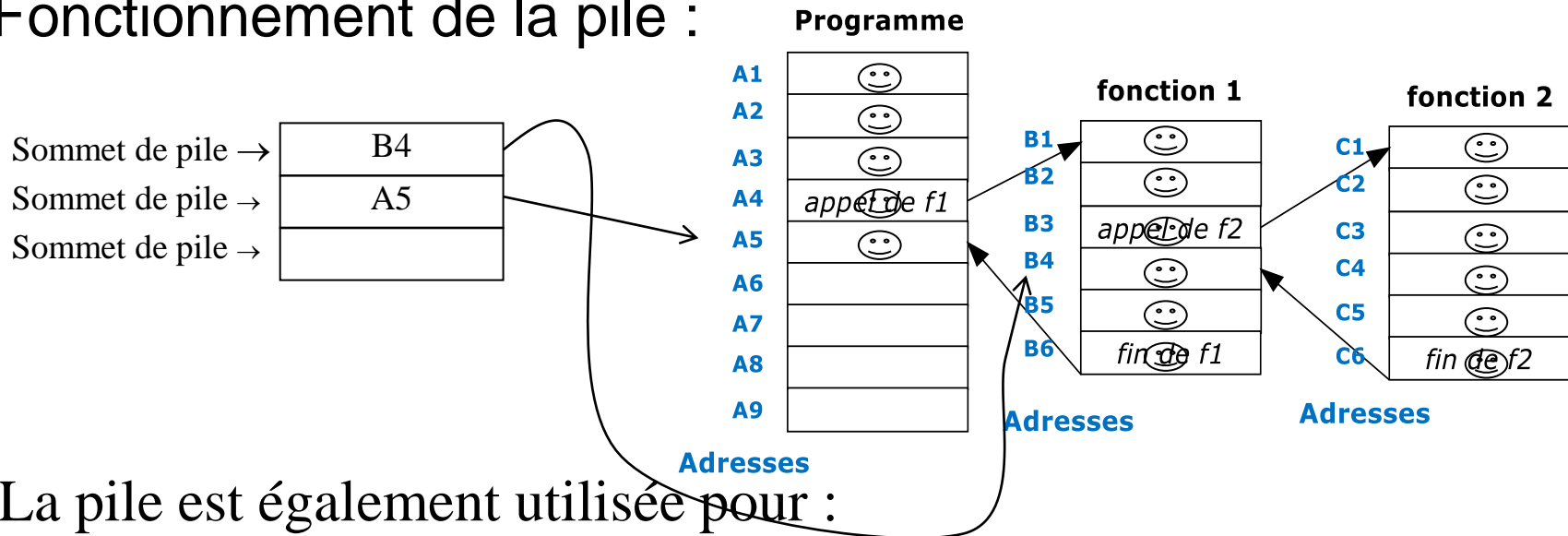


La partie du système qui s'occupe de ça s'appelle l'**Ordonnanceur**. Pour choisir le processus à rendre "En cours" il tient généralement compte de **priorités** entre les processus.

# Mémoire et processus

- Zone de mémoire pour le code
- Zones de mémoire pour les variables
- Zone de mémoire pour la pile

Fonctionnement de la pile :



La pile est également utilisée pour :

- Passer des paramètres à une fonction
- Recueillir sa valeur de retour

# Table des processus

Le SE gère une **table des processus** avec toutes les informations relatives à chaque processus. Pour chaque processus on y trouve des informations sur :

## Le processeur

- Registres de l'UT
- CO
- Registre d'état
- Sommet de pile

## Le processus

- État du processus
- Priorité
- ID du processus
- ID du père
- Infos sur le temps d'exécution

## Les fichiers

- Répertoire racine
- Fichiers ouverts
- Répertoire de travail
- Descripteurs de fichiers
- ID utilisateur
- ID groupe

## La mémoire

- Pointeur vers le segment de code
- Pointeur vers le segment de données
- Pointeur vers le segment de pile

## Autres

- Variables d'environnement
- Etc.

# Types de processus

2 types :

- processus **utilisateurs**
- processus **système**

Parmi les processus système certains sont **résidants** en mémoire et toujours présents (ils apparaissent toujours quand on utilise la commande UNIX **ps -e**).

Ils sont indispensables au bon fonctionnement du SE.

Ce sont les **démons** (init, inetd, ...).

D'autres correspondent à des services et ne s'exécutent qu'à certaines occasions (connexions réseau, ...)

# Contexte de processus sous UNIX

Le **contexte d'un processus** est constitué de 3 choses :

- Environnement utilisateur

- zone de programme *La zone de programme peut être partagée par plusieurs processus*
- zone de données *Les zones de données et de pile sont propres à chaque processus*
- zone de pile

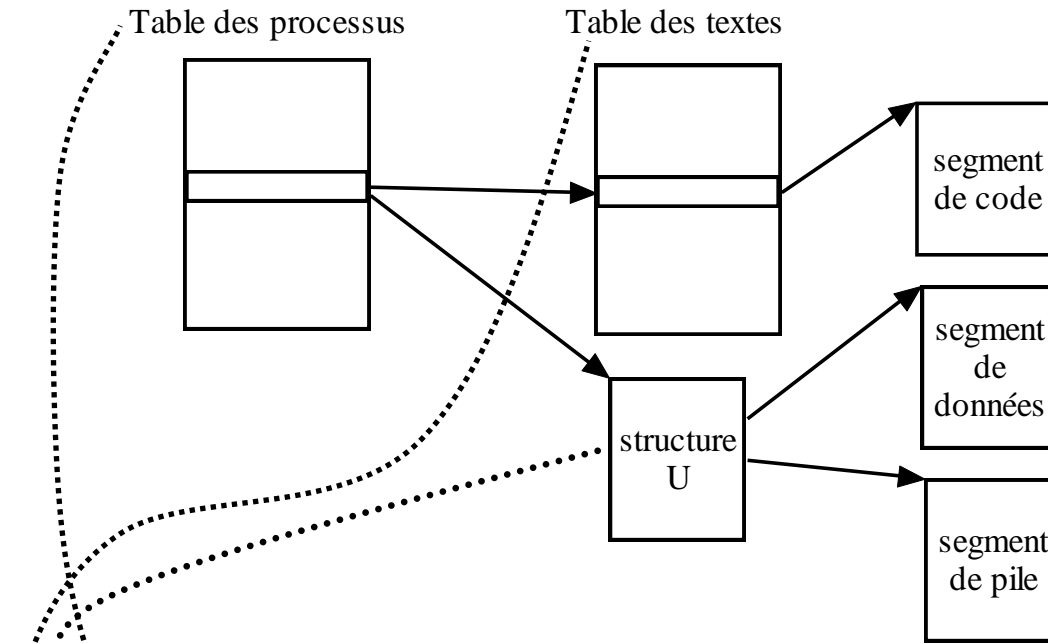
- Environnement machine

- registres

- Environnement système

- table des processus
- structure U
- table des textes

# Environnement système d'un processus



L'environnement système d'un processus est créé par le SE lors de la **création du processus**.

- **table des processus** : 1 entrée par processus avec : pid, état, priorité, utilisateur ...
- **structure U** : pointeurs vers les zones de mémoire (variables et pile), répertoire courant, descripteurs de fichiers ouverts, variables d'environnement
- **table des textes** : pointeurs vers les zones de programme partagées



# Les processus UNIX vus du programmeur

- **Identification des processus**

- pid\_t **getpid()** retourne le numéro de processus
- pid\_t **getppid()** retourne le numéro du processus père

- **Mise en sommeil de processus**

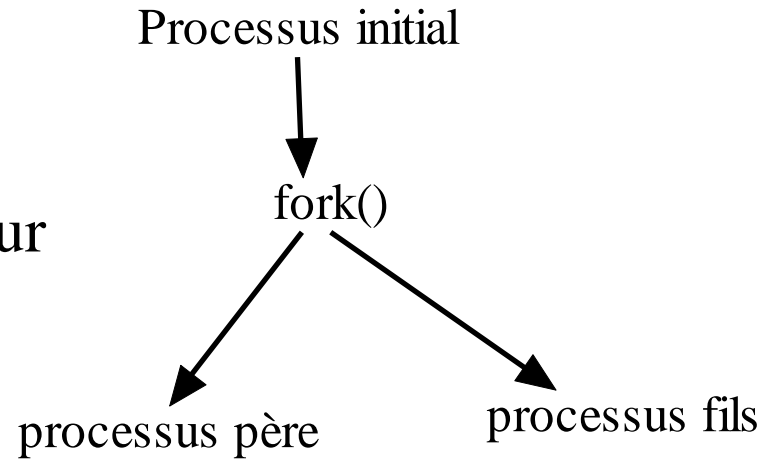
- **sleep**(unsigned int  $n$ ) met en attente le processus pour  $n$  secondes
- **usleep**(unsigned int  $n$ ) met en attente le processus pour  $n$  microsecondes (précision exagérée  $\Rightarrow$  pas forcément exactement respectée)
- **pause**() met en attente le processus jusqu'à l'arrivée d'un signal.

# Les processus UNIX vus du programmeur

## Création de processus

C'est par **clonage** du processus créateur

Le fils démarre à la fin du **fork**



La valeur de retour de la fonction **fork** est :

- le **numéro du pid** du fils dans le père ( $> 0$ )
- **0** dans le fils

# Les processus UNIX vus du programmeur

## Création de processus (suite)

Les 2 processus ont :

- le même **code** non dupliqué
- les mêmes **données** mais dupliquées
- les mêmes **fichiers** ouverts
- le même **environnement**

Remarque : comme ils partagent les mêmes fichiers si le fils se déplace dans un fichier le déplacement est valable pour le père

*Utilisation possible :*

```
pid_t numero;  
numero=fork();  
switch (numero) {  
    case -1 : erreur  
    case 0 : corps du processus fils  
    default : corps du processus père  
}
```

# Les processus UNIX vus du programmeur

- **Terminaison normale d'un processus**

- **exit**(int *etat*)

Quand le processus fils se termine son père peut récupérer sa **valeur de sortie** (terminaison normale par exit ou code d'erreur si terminaison sur erreur)

Le processus père a lui-même un père qui peut, par exemple, être le shell qui l'a créé.

- **Les processus zombies**

Un fils qui se termine envoie un **signal** (**SIGCHLD**) à son père puis reste en état de **zombie** jusqu'à ce que son père ait consulté sa valeur de retour (celle retournée par le exit du fils)

Remarque : si le père se termine sans le faire, le processus reste zombie jusqu'à être adopté par le démon **init** qui récupèrera ce **signal**.

# Les processus UNIX vus du programmeur

## Attente d'un processus

Le processus père peut attendre la terminaison de ses fils par : `int wait(int *etat)`

- Si un fils était **déjà terminé** wait se termine immédiatement et renvoie le pid de ce fils
- S'il n'y a **plus de fils** en cours wait se termine immédiatement et renvoie -1
- Sinon wait attend la **fin du 1<sup>er</sup> fils** et retourne son pid

L'appel de **wait** termine les processus zombies.

Le paramètre de **wait** reçoit le **code de retour** du fils.

Ce code est

- soit constitué à partir de la valeur passée en paramètre à **exit** par le fils (terminaison normale du fils → utiliser **WEXITSTATUS(etat)** )
- soit un **code d'erreur** fabriqué par le SE (terminaison anormale du fils) et indiquant la raison de cette terminaison (par exemple processus tué)

Remarque : si le père veut attendre ou tester la fin d'un processus particulier il peut utiliser la fonction **waitpid**(int *pid*, int \**etat*, int *option*) qui permet d'attendre le fils dont le pid est donné en 1<sup>er</sup> paramètre, le deuxième recevra le code de retour du fils et le 3<sup>ème</sup> permet d'indiquer s'il y a attente ou pas.

# Les processus UNIX vus du programmeur

## Recouvrement de processus

La création d'un processus se faisant par `fork`, le processus fils est **un clone du père**. La seule différence réside dans la valeur retournée par `fork`  $\Rightarrow$  on utilise un *if-else* ou un *switch* pour différencier les codes du père et du fils.

Parfois on veut que le fils soit un programme totalement différent du père pour cela il faut que le fils **recouvre** son propre code par celui d'une autre application.

UNIX offre pour cela les primitives **execl** et **execv** avec des variantes (`execle`, `execlp` et `exece`, `execp`) qui permettent de désigner :

- le fichier contenant le programme à exécuter
- les paramètres passés à ce programme.
- L'environnement

# Les processus sous Windows

La création se fait par **CreateProcess** qui a 10 paramètres :

1. Un pointeur sur le nom de l'exécutable
2. La ligne de commande
3. Un pointeur sur le descripteur de sécurité du processus
4. Un pointeur sur le descripteur de sécurité du processus initial
5. Un bit pour autoriser ou pas l'héritage des descripteurs de fichiers ouverts par le père
6. Des indicateurs de priorité
7. Un pointeur sur le descripteur d'environnement
8. Un pointeur sur le nom du répertoire courant
9. Un pointeur sur la fenêtre initiale du processus
10. Un pointeur sur la structure utilisée pour la valeur de retour du processus (18 valeurs !)

Remarque : sous Windows il n'y a pas de hiérarchie père-fils tous les processus sont créés au même niveau. Mais les paramètres de création (4, 5 et 10) permettent de garder un lien.

# Les Systèmes d'Exploitation

1. Introduction
2. Les Processus
- 3. Les Threads**
4. Concurrency entre processus
5. Communication entre processus
6. Les interruptions logicielles (signaux)
7. Gestion du CPU et de la mémoire
8. Ordonnancement
9. Les Entrées/Sorties
10. Le système de fichiers
11. Démarrage d'un système
12. Interblocage
13. Sécurité
14. Systèmes multiprocesseurs



# Les Threads

Aussi appelés **processus légers**.

Il s'agit d'une fonction s'exécutant en **parallèle mais partageant la même mémoire que le processus qui l'a créé** (pas une copie comme avec fork).

Utilisation favorisée par la technologie de **l'hyperthreading** (actuellement disponible sur tous les microprocesseurs) :

Introduit par Intel :

- Les registres de l'UC et de l'UT ainsi que les registres de gestion de la mémoire sont dupliqués
- En changeant de banque de registres on change **plus rapidement** de processus.

Remarque : les SE considèrent un cœur avec hyperthreading comme 2 cœurs mais il n'y a pas de vrai parallélisme.

# Threads ou processus ?

La différence essentielle est que tous les threads **partagent l'espace d'adressage** du même processus  $\Rightarrow$  ils peuvent modifier les variables en concurrence.

Quand un thread est créé **il accède aux variables du processus** et peut les modifier  $\Rightarrow$  aucune protection  $\Rightarrow$  un thread peut planter les autres threads du même processus. **MAIS** la protection est conservée entre processus.

Remarque : Quand un processus est créé sous UNIX **il récupère une copie des variables de son père**  $\Rightarrow$  il en connaît les valeurs au moment où il démarre mais s'il les modifie il ne modifie que sa copie et le père ne le voit pas.

Quand un processus se termine les threads qu'il a créé **se terminent** alors que quand un processus se termine les processus qu'il a créés **continuent à s'exécuter**.

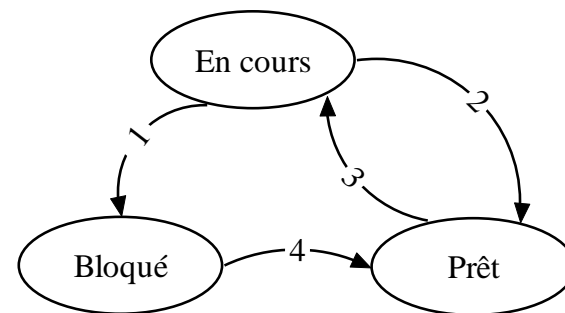
# Les Threads

Les threads sont utilisés pour faire des **programmes coopérants** tandis que les processus sont utilisés pour faire des **programmes coopérants** ou **concurrents**.

UNIX avec fork proposait une forme de processus coopérants parce qu'à l'époque les threads n'existaient pas.

Un thread a :

- ses registres
- son CO
- son registre d'état
- son sommet de pile
- **Tout le reste est partagé.**



Les threads ont accès à une primitive **yield** qui leur permet de volontairement passer leur tour (transition 2) avant que ne se termine leur quota de temps ou avant d'être bloqués. Ceci a un sens car les threads **coopèrent** alors que les processus sont généralement **concurrents** pour l'accès au CPU.

# Les threads sous UNIX

## Création d'un thread

```
int pthread_create(pthread_t * ident, pthread_attr_t * attributs,  
void * (*fonction) (void *), void * arguments)
```

Crée un thread qui exécute la fonction passée en 3<sup>ème</sup> paramètre qui reçoit le paramètre pointé par le **dernier paramètre** (NULL si pas de paramètre)

Le 1<sup>er</sup> paramètre recevra l'identificateur du thread (une sorte de pid)

Le 2<sup>ème</sup> paramètre peut être NULL si on veut adopter le comportement par défaut dans le cas contraire il faut :

1. initialiser une structure **pthread\_attr\_t** aux valeurs par défaut à l'aide de la fonction **pthread\_attr\_init**
2. en modifier certaines valeurs avec les fonctions **set\_attr\_xxx**.

# Les threads sous UNIX

## Attributs de thread

Modifier les attributs des threads revient à remplir la structure des attributs de threads qui est du type **pthread\_attr\_t**, puis à la passer en tant que deuxième argument à **pthread\_create()**.

**pthread\_attr\_init()** initialise la structure d'attributs de thread (2<sup>ème</sup> paramètre de *pthread\_create*) et la remplit avec les valeurs par défaut pour tous les attributs.

Chaque attribut *attrname* peut être individuellement modifié en utilisant la fonction **pthread\_attr\_setattrname()** et récupéré à l'aide de la fonction **pthread\_attr\_getattrname()**.

Les valeurs que l'on peut modifier concernent :

- la concurrence de ressources dans le processus
- la possibilité de consulter le code de retour du thread
- la pile allouée à ce thread
- la priorité du thread
- l'ordonnancement
- la possibilité d'hériter des paramètres du père.

# Exemple de création d'un thread sous UNIX

```
pthread_attr_t  attributs;  
pthread_t  identificateur;  
.....  
pthread_attr_init(&attributs);  
pthread_attr_setscope(&attributs, PTHREAD_SCOPE_SYSTEM);  
pthread_create(&identificateur, &attributs, void * calcul (void *), NULL);
```

Et on écrit une fonction calcul qui sera exécutée par le thread

```
void calcul(void) { ..... }
```

on a utilisé **pthread\_attr\_init** pour récupérer les attributs standard

**pthread\_attr\_setscope** pour modifier, pour ce thread, le type d'ordonnancement (*avec tous les autres threads*) (la valeur par défaut est PTHREAD\_SCOPE\_PROCESS – *avec les autres threads du même processus*)

**pthread\_create** pour créer le thread.

# Les threads sous UNIX

## Terminaison d'un Thread

**pthread\_exit**(void \* retour);

Le paramètre sera le code de retour du thread (on peut mettre NULL si on ne veut pas de code de retour).

## Attente de terminaison d'un thread

**pthread\_join**(pthread\_t ident, void \*\* retour);

Attend la fin d'un processus dont l'identificateur (récupéré lors du pthread\_create) est donné en 1<sup>er</sup> paramètre.

Le 2<sup>ème</sup> paramètre recevra la valeur de retour du thread : il s'agit d'un pointeur vers la valeur de retour, or la valeur de retour est un pointeur vers le code de retour (d'où le void \*\*).

On peut mettre NULL si on ne veut pas récupérer cette valeur.

# Les threads sous UNIX

## Libération de ressources

Contrairement à un processus un thread ne libère pas les ressources quand il se termine car elles peuvent être utilisées par d'autres threads.

On peut libérer les ressources à la terminaison d'un thread explicitement par **pthread\_detach**(pthread\_t ident) ça peut être fait par un autre thread ou par un processus qui attend sa fin par **pthread\_join**

Attention : Quand un thread a libéré ses ressources **il ne peut plus être attendu par un autre** (pthread\_join ne fonctionne plus).



# Les threads sous Windows

- Ils sont créés par **CreateThread**(...) dont les paramètres sont :
  - Attributs de sécurité (en général NULL)
  - Taille de la pile à allouer (0 si taille par défaut)
  - Fonction exécutée par le thread
  - Paramètres passés à cette fonction
  - Indicateurs de mode de création du thread (semblable à pthread\_attr\_t)
  - Identificateur du thread (en général NULL car défini par Windows)
- Ils se terminent par **ExitThread**(code\_de\_sortie)
- On peut en modifier la priorité par **SetThreadPriority**(identificateur, priorité).

# Intérêt du parallélisme

- Exemple : programme qui télécharge N images sur Internet, les traite et les enregistre.
- Solution classique :
  - Faire N fois
    - télécharger une image
    - la traiter et l'enregistrer
- Durée :
  - Si T1 est le temps de téléchargement
  - Si T2 est le temps de traitement et d'enregistrement

Le programme dure  $N \cdot (T1 + T2)$

# Intérêt du parallélisme

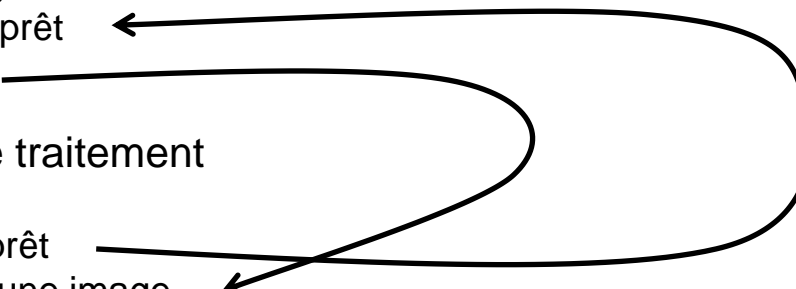
- Solution parallélisée :
  - Un processus P1 fait le téléchargement
    - Faire N fois
      - télécharger une image
      - attendre que P2 soit prêt
      - lui envoyer l'image
  - Un processus P2 fait le traitement
    - Faire N fois
      - signaler que P2 est prêt
      - attendre de recevoir une image
      - la traiter et l'enregistrer
- Durée :

Le programme ne dure plus que :  $T1 + (N-1) * \text{MAX}(T1, T2) + T2$

# Intérêt du parallélisme

- La durée du programme est  $T1 + (N-1) * \text{MAX}(T1, T2) + T2$ . Donc si  $T1$  (temps de téléchargement) est plus long que  $T2$  c'est  $N * T1 + T2$ 
  - Dans ce cas on peut lancer deux (ou trois ou plus) processus de téléchargement en parallèle
  - Durée :  
Le programme dure  $T1 + N * T2$  (les téléchargements se font pendant les traitements sauf le 1<sup>er</sup>)
- Si c'est  $T2$  qui est plus long que  $T1$  on lancera deux ou plus de processus de traitement en parallèle
  - Durée :  
Le programme dure  $N * T1 + T2$  (les traitements se font pendant les téléchargements sauf le dernier)
- Conclusion : le programme dure  $T1 + (N-1) * \text{MIN}(T1, T2) + T2$

# Mécanismes pour le parallélisme

- Solution parallélisée :
    - Le processus P1 fait le téléchargement
      - Faire N fois
      - télécharger une image
      - attendre que P2 soit prêt
      - lui envoyer l'image
    - Le processus P2 fait le traitement
      - Faire N fois
      - signaler que P2 est prêt
      - attendre de recevoir une image
      - la traiter et l'enregistrer
- 

**On a besoin de mécanismes de synchronisation et de communication entre processus**

# Les Systèmes d'Exploitation

1. Introduction
2. Les Processus
3. Les Threads
- 4. Concurrency entre processus**
5. Communication entre processus
6. Les interruptions logicielles (signaux)
7. Gestion du CPU et de la mémoire
8. Ordonnancement
9. Les Entrées/Sorties
10. Le système de fichiers
11. Démarrage d'un système
12. Interblocage
13. Sécurité
14. Systèmes multiprocesseurs

# Concurrence entre processus

Trois questions :

1. Comment **passer des informations** d'un processus à un autre ?
2. Comment s'assurer que le **séquencement** est assuré c'est à dire que le processus B attende que le processus A soit prêt ?
3. Comment éviter que des processus n'**entrent en conflit** par exemple quand ils veulent accéder à la même ressource ?

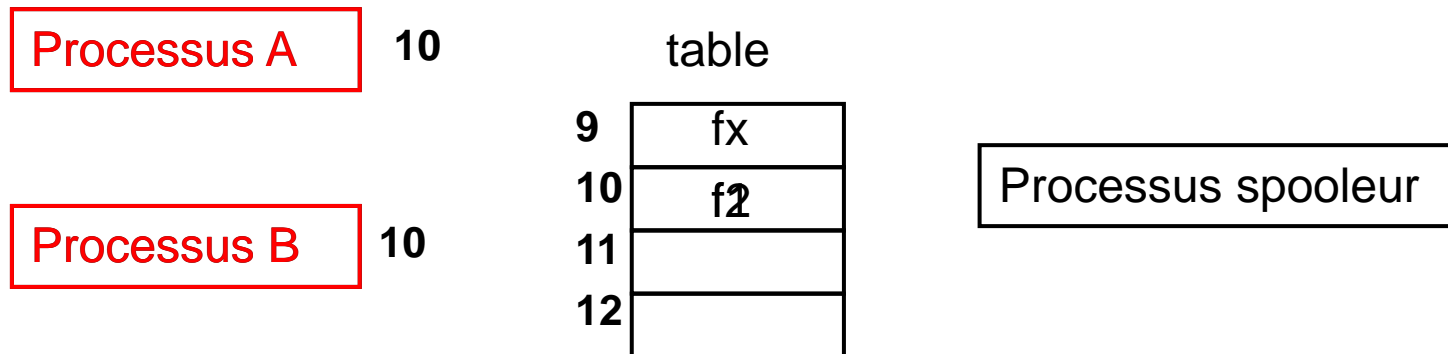
Remarque : pour les threads la 1<sup>ère</sup> question est facilement résolue puisque l'espace mémoire est partagé mais pas les 2 autres.

# Le problème de la concurrence

Lorsque plusieurs processus utilisent une même ressource se pose le problème de **concurrence** => risque de dysfonctionnements.

Exemple : spooleur d'impression :

- 1 processus (spooleur) explore une table où sont indiqués les fichiers à imprimer et les imprime puis les retire de la table
- 2 processus A et B veulent imprimer un fichier (f1 pour A et f2 pour B).



A écrit 10 en position 10 et le contenu de la table est la 10

**f2 ne sera jamais imprimé et l'erreur est non détectable.**



# À la recherche d'une solution

Il faudrait éviter que B ne s'exécute tant que A **n'a pas terminé la séquence** :

- trouver la 1<sup>ère</sup> entrée libre de la table du spoleur
- y mettre le nom du fichier à imprimer

On dit que cette séquence constitue une **section critique**.

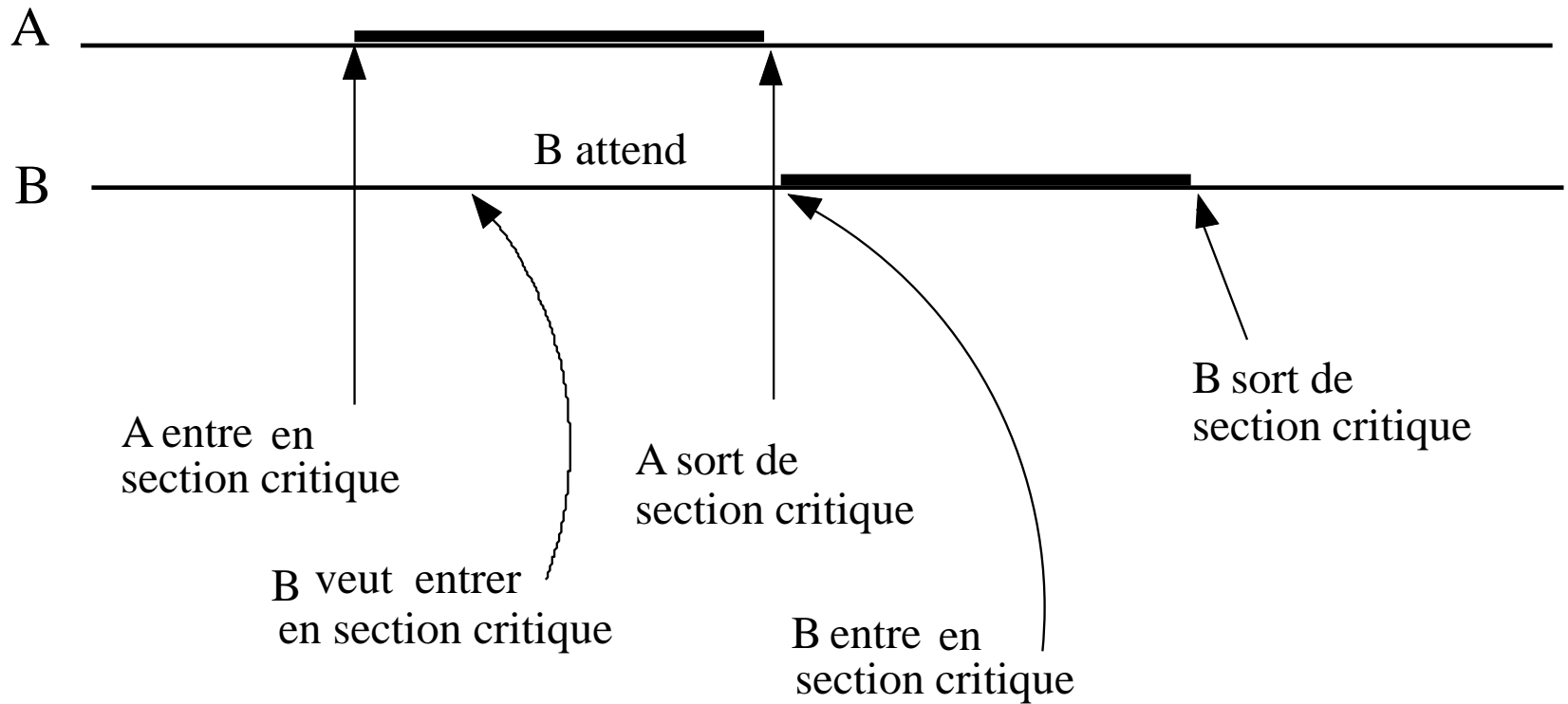
**Le SE ne sait pas ce que font A et B**  $\Rightarrow$  il ne peut pas savoir qu'il ne faut pas exécuter le processus B pendant cette séquence.

C'est un problème d'**exclusion mutuelle**  $\Rightarrow$  il faut respecter les règles suivantes :

- 2 processus ne peuvent être **simultanément** dans une section critique
- on ne peut rien supposer sur le **temps d'exécution d'un processus**
- on ne peut rien supposer sur l'**ordre d'exécution des processus**
- un processus ne doit pas **rester trop longtemps** dans une section critique donc un processus ne va pas **attendre trop longtemps** pour rentrer dans une section critique.

# Solution souhaitée

On arrive au schéma suivant :



# Résoudre le problème de l'exclusion mutuelle

## Solution 1

**Désactiver les ITs** en entrant en **section critique** et les réactiver en sortant

⇒ Le processus en cours conserve le CPU

Car aucun autre processus ne peut démarrer par temps partagé (IT du timer)  
mais même un pilote de périphérique ne peut pas s'exécuter (IT de l'UE).

**MAIS** : Cette solution ne marche que **s'il n'y a qu'un CPU** (cœur)

**DE PLUS** elle est **dangereuse** car si un processus utilisateur peut désactiver les ITs **il peut tout bloquer** (problème avec les virus).

Remarque : en général seul le SE peut désactiver les ITs.

# Résoudre le problème de l'exclusion mutuelle

## Solution 2

Utiliser une **variable de verrou**



Principe :

Une variable globale est initialisée à **FAUX**

Les processus qui utilisent des **sections critiques** suivent l'**algorithme** suivant :

tant que (verrou = VRAI) attendre

verrou  $\leftarrow$  VRAI

**exécution de la section critique**

verrou  $\leftarrow$  FAUX

# Résoudre le problème de l'exclusion mutuelle

Le problème est qu'on ne sait pas **comment le CPU exécute ces instructions**.

Par exemple, si on suppose que *tant que (verrou = VRAI) attendre* se traduit par :

attente:      `LOAD    registrex, verrou    // mettre verrou dans un registre`  
                 `CMP     registrex, VRAI    // comparer ce registre à VRAI`  
                 `BEQ     attente                    // si c'est égal boucler`

Processus A (cœur1)	Processus B (cœur2)
<code>LOAD    registrex, verrou    ; registrex=FAUX</code> <code>CMP     registrex, VRAI</code> <code>BEQ     attente                    ; on n'attend pas</code> <code>MOV     verrou, VRAI</code> <b>entrée dans la section critique</b>	<code>LOAD    registrex, verrou    ; registrex=FAUX</code> <code>CMP     registrex, VRAI</code> <code>BEQ     attente                    ; on n'attend pas</code> <code>MOV     verrou, VRAI</code> <b>entrée dans la section critique</b>

**Les 2 processus sont simultanément en section critique**

Le problème vient du fait que **l'accès à la variable de verrou est lui-même critique !!!!**

# Résoudre le problème de l'exclusion mutuelle

## Solution 3

Pour éviter le problème précédent on pourrait utiliser des **variables de verrou différentes** pour chaque processus ainsi le verrou n'est modifié que par son propriétaire mais est consulté par les autres.

Proposition : Utiliser un tableau de booléens (1 verrou par processus mis en concurrence). Quand un processus veut entrer en section critique **il met son booléen à VRAI (il ferme son verrou) et attend que tous les autres soient à FAUX (qu'ils soient tous ouverts)**. De cette façon on n'a plus d'accès concurrent en écriture et le problème d'accès critique à la variable de verrou est résolu.

Exemple d'algorithme pour 2 processus qui utilisent des sections critiques :

```
veut_entrer[moi] ← VRAI  
tant que (veut_entrer[lui]) attendre  
exécution de la section critique  
veut_entrer[moi] ← FAUX
```

# Etude de cette solution

Processus A	Processus B
<pre>veut_entrer[moi] ← VRAI tant que (veut_entrer[lui]) attendre exécution de la section critique veut_entrer[moi] ← FAUX</pre>	<pre>veut_entrer[moi] ← VRAI tant que (veut_entrer[lui]) attendre exécution de la section critique veut_entrer[moi] ← FAUX</pre>

~~À exécuter les 2 processus en parallèle~~  
A exécuté la section critique TQ B n'entre pas en section critique => bloqué dans le TQ

Les 2 processus sont bloqués dans le TQ

Aucun n'entre en section critique

**On voulait éviter que les 2 processus n'y rentrent simultanément et le résultat c'est qu'aucun n'y rentre !**

# Résoudre le problème de l'exclusion mutuelle

## Solution 4

En 1965 **Dekker** propose un algorithme qui marche pour 2 processus. Il utilise la même table de verrous : **veut\_entrer** + une variable **tour**.

L'idée est d'**éviter le blocage dans le tant que (veut\_entrer[lui])**. Pour cela on va, pendant cette boucle, céder son tour si l'autre n'a pas encore pu passer.

La variable **tour** est une sorte de privilège c'est à dire qu'un processus qui a effectué sa section critique accepte de céder son tour à l'autre en cas de conflit.



# Résoudre le problème de l'exclusion mutuelle

## Solution 4

L'algo pour les 2 processus est alors le suivant :

```
veut_entrer[moi] ← VRAI // on veut entrer en section critique
tant que (veut_entrer[lui]) // l'autre est déjà en section critique
    si (tour = lui) // si c'est son tour
        veut_entrer[moi] ← FAUX // on lui cède la place
        TQ (tour = lui) attendre // on attend qu'il ait fini
        veut_entrer[moi] ← VRAI // on rentre en section critique
    finsi
fintq
exécution de la section critique
veut_entrer[moi] ← FAUX // on sort de section critique
tour ← lui // on cède le privilège
```

**Cette méthode marche mais elle devient compliquée au-delà de 2 processus !**

# Résoudre le problème de l'exclusion mutuelle

## Solution 4

En 1982 **Peterson** en propose une solution plus simple dont le principe est le même.

L'algo pour les 2 processus est alors le suivant :

veut\_entrer[moi]  $\leftarrow$  VRAI

tour  $\leftarrow$  moi

tant que ((tour = moi) ET (veut\_entrer[lui])) attendre  
**exécution de la section critique**

veut\_entrer[moi]  $\leftarrow$  FAUX

# Etude de cette solution (cas 1)

## Processus 1

veut\_entrer[1]  $\leftarrow$  VRAI  
tour  $\leftarrow$  1

entrée dans la section critique

Fin de la section critique  
veut\_entrer[1]  $\leftarrow$  FAUX

## Processus 2

veut\_entrer[2]  $\leftarrow$  VRAI  
tour  $\leftarrow$  2

Boucle car tour=2 et  
veut\_entrer[1] = VRAI

Sort de la boucle car veut\_entrer[1] =  
FAUX

entrée dans la section critique

Changement de processus : 1  $\rightarrow$  2

# Etude de cette solution (cas 2)

## Processus 1

veut\_entrer[1] ← VRAI

tour ← 1

Boucle car tour=1 et veut\_entrer[2]  
= VRAI

Sort de la boucle car veut\_entrer[2] = F  
**entrée dans la section critique**  
veut\_entrer[1] ← FAUX

## Processus 2

veut\_entrer[2] ← VRAI

tour ← 2

Boucle car tour=2 et veut\_entrer[1] =  
VRAI

Sort de la boucle car tour=1  
**entrée dans la section critique**  
veut\_entrer[2] ← FAUX

Changement de processus : 2 → 2

# Résoudre le problème de l'exclusion mutuelle

## Solution 5

La solution 2 (verrou unique) est simple quel que soit le nombre de processus mais ne marche pas si on ne peut pas faire en sorte que les opérations sur le verrou soient **indivisibles**.

On a ajouté aux processeurs des **instructions spéciales** pour manipuler les verrous en 1 seule opération  $\Rightarrow$  un processus termine forcément l'instruction de manipulation du verrou avant qu'un autre ne puisse y accéder  $\Rightarrow$  Le problème est résolu.

Ce sont des instructions **TSL** (Test and Set Lock).

TSL registre, variable fait les opérations suivantes :

registre  $\leftarrow$  variable

variable  $\leftarrow$  1

Pendant l'exécution de cette instruction :

- on ne peut pas être **interrompu** (normal puisque c'est une seule instruction)
- aucun **accès à la mémoire** n'est possible (pour éviter qu'un autre CPU ne puisse modifier le verrou) – **LOCK = verrouillage de la mémoire pendant l'instruction**.

# Verrou avec l'instruction TSL

## L'algorithme de verrou et sa traduction

tant que (verrou=VRAI) attendre  
verrou  $\leftarrow$  VRAI

exécution de la section critique

verrou  $\leftarrow$  FAUX

attendre: TSL registre, verrou

CMP registre, 0

BNE attendre

exécution de la section critique

MOV verrou, 0

Ça marche !

# Solutions au problème de concurrence

On a des solutions au problème de concurrence par :

1. définition de **section critique**
2. utilisation d'un **mécanisme d'entrée en section critique**
3. utilisation d'un **mécanisme de sortie de section critique**

Avec des instructions **TSL** le matériel résout le problème sinon il faut utiliser l'algorithme de **Dekker** ou de **Peterson**

Problème : Toutes ces méthodes utilisent une **attente active** (boucle TQ) càd que le processus se bloque en exécutant une boucle => il utilise du temps processeur juste pour tester le verrou et attendre qu'il s'ouvre.

Remarque : L'attente active peut poser des problèmes si les processus ne sont pas de **même priorité** : Si le processus qui est en section critique a une priorité plus faible que celui qui attend, le CPU consacrerait beaucoup de temps à boucler pour rien et très peu à faire avancer l'autre processus qui débloquerait le 1<sup>er</sup> !!!!

Proposition : **mettre en sommeil** (état bloqué) les processus qui attendent une entrée en section critique plutôt que de les faire boucler pour rien.

# Recherche d'une solution correcte

On va étudier un problème classique pour illustrer ça :

## **Le problème du producteur/consommateur :**

2 processus **partagent un tampon** de taille fixe l'un (le producteur) le remplit et l'autre (le consommateur) le vide.

Les problèmes se posent quand le producteur trouve le tampon **plein** parce que le consommateur ne va pas assez vite ou quand le consommateur trouve le tampon **vide** parce que le producteur ne va pas assez vite

On va utiliser une variable **taille** qui donne la taille occupée du tampon sachant que la taille maxi est MAX.



# Le problème du producteur/consommateur

Producteur	Consommateur
TQ vrai Préparer un élément Si ( <b>taille</b> =MAX) <b>se bloquer</b> Mettre l'élément dans le tampon <b>taille</b> $\leftarrow$ <b>taille</b> + 1 Si ( <b>taille</b> =1) <b>débloquer le consommateur</b> FTQ	TQ vrai Si ( <b>taille</b> =0) <b>se bloquer</b> Récupérer un élément dans le tampon <b>taille</b> $\leftarrow$ <b>taille</b> - 1 Si ( <b>taille</b> =MAX-1) <b>débloquer le producteur</b> Traiter l'élément FTQ

Problème : L'accès à **taille** n'est pas géré comme une **section critique**

- Cœur 1 : le consommateur lit **taille** et trouve 0 puis le SE passe à un autre processus sur ce cœur
- Cœur 2 : en même temps le producteur ajoute un élément, incrémente **taille** et débloque le consommateur mais comme il n'était pas bloqué ça n'a aucun effet.
- Cœur 1 : le consommateur redémarre et **se bloque et ne sera jamais débloqué.**

**Ça coince !**

# Étude de cette solution

Solution à ce problème : le **déblocage** dit être **mémorisé** c'est à dire que quand le producteur débloque le consommateur c'est mémorisé et donc, dès que le consommateur voudra se bloquer, il sera instantanément débloqué => **ça marche !**

Remarque : cette méthode devient plus complexe à mettre en œuvre si on a plus de 2 processus (**lequel est débloqué ?**).

C'est le cas, par exemple, d'un spooleur d'impression (N producteurs pour 1 seul consommateur).

Il faut trouver une solution simple à utiliser dans tous les cas.

# Vers une solution universelle

- Problème de la concurrence :
  - Quand : Partage d'une ressource
  - Où : on ne sait pas comment les processus s'enchaînent. Avec un langage de programmation on ne sait pas ce qui est réellement exécuté par le processeur (instructions).
- Solution par **exclusion mutuelle** :
  - 2 processus ne doivent pas être simultanément dans une **section critique**.

# L'exclusion mutuelle (problème)

- L'exemple du producteur/consommateur contient :
    - Un problème de blocage/déblocage des processus quand le tampon est plein/vide
    - Un problème d'exclusion mutuelle d'accès au tampon
  - $\Rightarrow$  Il faut un outil unique pour gérer tout ça.
  - Remarque : C'est le même problème dans les 2 cas car :
    - Tampon de taille N : on peut se bloquer quand il est plein ou quand il est vide
    - Tampon de taille 1 : joue le rôle de verrou (il ne peut être que plein ou vide = ouvert ou fermé)
      - Le remplir
      - Entrer en section critique
      - Le vider
- } Exclusion mutuelle pour section critique

# Solution unique

L'outil unique est le **sémaphore**.

Il utilise le même principe mais le tampon n'est représenté que par son nombre de cases libres qui équivaut à un **compteur de déblocages**. On peut débloquent autant de processus que ce qu'indique le compteur,

**C'est un verrou qu'on peut ouvrir plusieurs fois d'avance mais qu'on ne peut fermer que s'il est ouvert au moins une fois.**

Il offre deux opérations correspondant à :

L'entrée en section critique (avec endormissement si nécessaire)  
≡ ajouter un élément au tampon ≡ diminuer le compteur de déblocages

La sortie de section critique (avec réveil des processus endormis s'il y en a)  
≡ enlever un élément au tampon ≡ augmenter le compteur de déblocages.

# Les sémaphores

Inventés par **Dijkstra** en 1965. L'idée est d'utiliser un compteur de déblocages.

Un **sémaphore** est un entier  $S$  manipulable par 2 opérations :

$P(S) :$	si $(S \leq 0)$ <b>bloquer le processus</b>	// on ne peut pas fermer
	$S \leftarrow S - 1$	// on ferme
$V(S) :$	$S \leftarrow S + 1$	// on ouvre
	<b>débloquer les processus endormis</b>	// quelqu'un peut passer

- (note) : les noms  $P$  et  $V$  viennent du Hollandais Proberen (tester), et Verhogen (augmenter). Il faut que les opérations  $P$  et  $V$  soient **atomiques** c'est à dire que quand un processus exécute  $P$  ou  $V$  il ne peut pas être interrompu tant qu'il n'a pas fini (blocage/déblocage des ITs).
- Il faut aussi que l'accès à la mémoire soit **verrouillé** pour qu'un autre processeur ne puisse pas modifier  $S$  (multicœurs). Et que  $S$  **ne soit pas mis en cache mémoire** pour ne pas modifier une copie.

**C'est la solution universelle à tous les problèmes de concurrence entre processus**

# Sémaphores et sections critiques

Pour le cas particulier des **sections critiques**, il suffit :

- D'associer un **sémaphore** à chaque section critique
- Qu'un processus fasse **P** avant d'entrer en section critique
- Qu'il fasse **V** dès qu'il en sort

On parle alors de **MUTEX** c'est à dire de sémaphore binaire (ne valant que 0 ou 1 = fermé/ouvert) qui permet **l'exclusion mutuelle** d'entrée en section critique.

# Application au problème du producteur/consommateur

Dans le problème du producteur/consommateur on va utiliser :

- un **MUTEX** pour éviter l'accès concurrent au tampon (**section critique**)
- un **sémaphore** A pour **bloquer le producteur** quand le tampon est plein
- un **sémaphore** B pour **bloquer le consommateur** quand le tampon est vide

Remarques :

**A** indique le **nombre de places disponibles dans le tampon**

(il est décrémenté par le producteur (P) et incrémenté par le consommateur (V))

**B** indique le **nombre de places occupées dans le tampon**

(il est décrémenté par le consommateur (P) et incrémenté par le producteur (V)).



# Le producteur

TQ vrai

Préparer un élément

**P(A)** // si A est à 0 le producteur se bloque car le tampon est plein

**P(MUTEX)** // utilisé pour éviter l'accès concurrent au tampon

Mettre l'élément dans le tampon

**V(MUTEX)** // libérer l'accès au tampon

**V(B)** // réveiller le consommateur car le tampon n'est pas vide

FTQ

# Le consommateur

TQ vrai

**P(B)** // si B est à 0 le consommateur se bloque car le tampon est vide

**P(MUTEX)** // utilisé pour éviter l'accès concurrent au tampon

Récupérer un élément dans le tampon

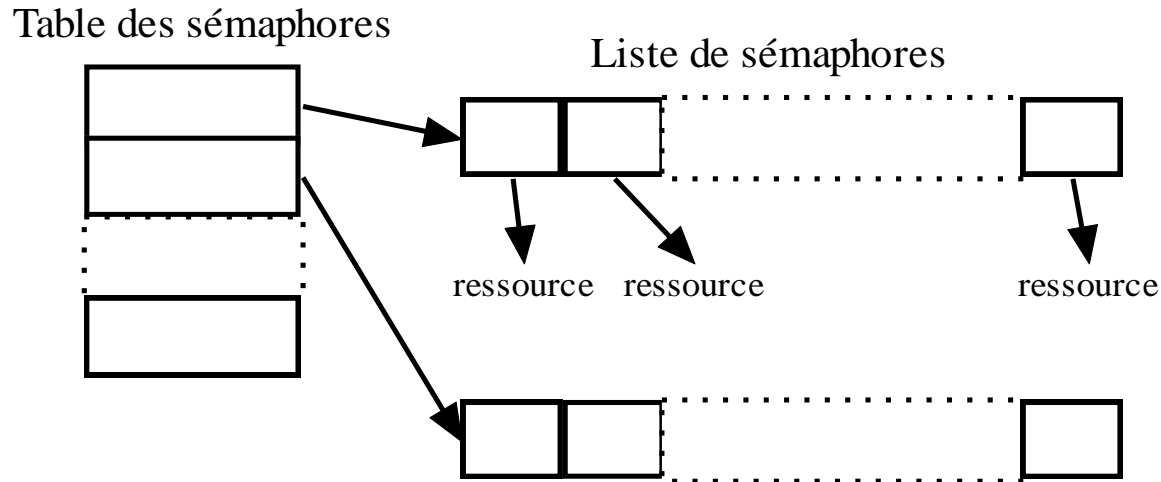
**V(MUTEX)** // libérer l'accès au tampon

**V(A)** // réveiller le producteur car le tampon n'est pas plein

Traiter l'élément

FTQ

# Les sémaphores pour les processus d'UNIX



Unix gère une **table des sémaphores**.

Chaque entrée dans cette table est un pointeur vers une **liste de sémaphores** (chacun des sémaphores de cette liste gère une ressource)

Les opérations portent sur un élément d'une liste (donc **un sémaphore**).

Unix propose des opérations plus générales que **P** et **V** mais qui permettent de réaliser P et V donc de résoudre tout problème

De plus il permet de faire plusieurs opérations sur un ou plusieurs sémaphores de la même liste **de façon atomique**.

# Création d'une liste de sémaphores (UNIX)

Principe : UNIX gère un **identifiant interne** pour chaque **liste de sémaphores**. C'est la même chose que le numéro associé à un descripteur de fichier que l'on obtient par open.

Mais pour un fichier on a un moyen de **l'identifier de façon unique** par chemin + nom.  
2 processus voulant ouvrir le même fichier peuvent le faire car ils connaissent son nom.

Par contre ils ne peuvent le faire en utilisant son **identifiant interne** (n°) que s'ils sont père et fils (le fils a récupéré une copie de la variable contenant le numéro obtenu par le père).

Pour les sémaphores on a le même problème sauf qu'on n'a pas de nom pour le désigner.  
UNIX propose alors comme solution d'utiliser une **clé** qui est une sorte de nom que tous les processus peuvent connaître.

## Primitive de création d'une liste de sémaphores

**semget**(key\_t cle, int nombre, IPC\_CREAT|0xyz)

Cette fonction retourne l'**identificateur interne** de cette liste.

- Le 1<sup>er</sup> paramètre est la clé associée
- Le 2<sup>ème</sup> paramètre est le nombre de sémaphores de la liste
- Le 3<sup>ème</sup> paramètre permet la création et donne des droits d'accès (comme sur les fichiers `xrwxrwxrw`)

# Accès à une liste de sémaphores (UNIX)

La même fonction **semget**(key\_t cle, int nombre, 0) permet de retrouver l'identifiant d'une liste de sémaphores dont on a la clé : il suffit de mettre 0 dans le dernier paramètre.

## Le problème des clés

UNIX propose une fonction de création de clés à partir d'un nom de fichier et d'un entier.

**ftok**(char \* nomfich, int numero) retourne une cle (key\_t).

Remarque : le fichier doit exister mais il n'est pas utilisé par ftok.

Le 2<sup>ème</sup> paramètre permet de générer des clés différentes à partir du même nom de fichier

Remarque : si le processus qui doit utiliser la liste de sémaphores peut connaître la valeur retournée lors de sa création (c'est un processus fils) on n'a pas besoin de clé. Dans ce cas on utilise, lors de la création, la constante **IPC\_PRIVATE** en lieu de clé :

**semget**(IPC\_PRIVATE, int nombre, IPC\_CREAT|0xyz)

# Opérations sur une liste de sémaphores (UNIX)

La primitive **semop** permet d'effectuer un ensemble d'opérations sur une liste de sémaphores. Le processus qui l'utilise sera **mis en sommeil** si l'une de ces opérations est **bloquante**.

**semop**(int **ident**, struct sembuf \***operations**, size\_t **nombres\_d\_ops**)

- Le 1<sup>er</sup> paramètre est l'identifiant de la liste (obtenu par *semget*)
- Le 2<sup>ème</sup> paramètre est un tableau dont chaque élément est une structure désignant une opération sous la forme :

```
struct sembuf {  
    ushort_t    sem_num;    // n° du sémaphore dans la liste  
    short       sem_op;     // opération à effectuer (c'est un entier)  
    short       sem_flg;    // options de l'opération  
}
```

- Le 3<sup>ème</sup> paramètre indique le nombre d'opérations à faire (c'est à dire la taille du tableau en 2<sup>ème</sup> paramètre ).

# Opérations sur une liste de sémaphores (UNIX)

- Valeurs possibles de l'entier **sem\_op** de la structure décrivant l'opération :
  - si `sem_op > 0` On **ajoute** cette valeur au sémaphore et on **réveille les processus** qui attendaient que la valeur augmente
  - si `sem_op < 0` On **ajoute** cette valeur au sémaphore sans qu'il devienne `< 0`  
Si on ne peut pas le faire **le processus est bloqué** jusqu'à ce que la valeur du sémaphore augmente (et que l'opération devienne faisable)  
Si la valeur devient nulle tous les processus qui attendent que la valeur soit nulle (voir ci-dessous) **sont réveillés**
  - si `sem_op = 0` On ne modifie pas le sémaphore mais **le processus est bloqué** s'il n'est pas nul.
- L'entier **sem\_flg** de la structure décrivant l'opération est normalement nul mais d'autres valeurs peuvent permettre par exemple d'annuler des opérations.

```
struct sembuf {  
    ushort_t  sem_num;  
    short     sem_op;  
    short     sem_flg;  
}
```

Remarque : avec **sem\_op = -1** et **sem\_flg = 0** on obtient l'opération de base **P**  
avec **sem\_op = 1** et **sem\_flg = 0** on obtient l'opération de base **V**

# Contrôle d'une liste de sémaphores (UNIX)

La primitive **semctl** permet d'effectuer diverses opérations de contrôle sur une liste de sémaphores. Les opérations possibles sont :

- lecture du nombre de processus en attente d'augmentation d'un sémaphore
- lecture du nombre de processus en attente de nullité d'un sémaphore
- lecture de la valeur d'un sémaphore ou de tous
- lecture du pid du dernier processus ayant effectué une opération de sémaphore
- modification de la valeur d'un sémaphore ou de tous
- suppression de la liste de sémaphores

La fonction est :

**semctl**(int **ident**, int **semnum**, int **opération**, **paramètre**)

- **ident** est l'identificateur de la liste de sémaphores
- **semnum** est le n° du sémaphore dans la liste ou le nombre de sémaphores selon l'opération
- **opération** est une constante qui désigne l'opération
- **paramètre** est soit un entier soit un tableau d'entiers soit un pointeur selon l'opération.

# Contrôle d'une liste de sémaphores UNIX (suite)

On utilise La primitive **semctl** en particulier pour :

- initialiser les valeurs des sémaphores
- supprimer une liste de sémaphores

## Exemples :

**semctl**(*id*, *num*, SETVAL, *val*) initialise le sémaphore de n° *num* de la liste de sémaphores identifiée par *id* à la valeur *val*.

**semctl**(*id*, 0, IPC\_RMID, 0) supprime la liste de sémaphores identifiée par *id* (les 2<sup>ème</sup> et 4<sup>ème</sup> paramètres ne servent pas).



# Les sémaphores pour les threads d'UNIX

Les threads peuvent communiquer par **variables partagées** mais l'accès à ces variables étant concurrent ils ont besoin de sémaphores de type **MUTEX**.

UNIX propose des primitives pour manipuler de tels sémaphores de façon simple:

**Création d'un MUTEX** par sa **déclaration**:

**pthread\_mutex\_t** *mon\_mutex* = **PTHREAD\_MUTEX\_INITIALIZER**

**Opération P** par la fonction:

**pthread\_mutex\_lock**(&*mon\_mutex*)

**Opération V** par la fonction:

**pthread\_mutex\_unlock**(&*mon\_mutex*)

**Destruction d'un MUTEX** par la fonction:

**pthread\_mutex\_destroy**(&*mon\_mutex*)

Remarque : UNIX propose d'autres fonctions comme **pthread\_mutex\_init** qui permet de créer un mutex avec vérification d'erreur. Il propose aussi des fonctions permettant de modifier certains attributs des mutex. Mais **ces fonctions ne sont pas portables** d'un UNIX à un autre donc il vaut mieux éviter de les utiliser.

# Les moniteurs

Principe : Offrir des **fonctions** (ou des méthodes) et des **structures de données** (ou des objets) pour l'accès aux ressources partagées.

L'accès ne peut se faire **que par ces fonctions (ou méthodes)** et les structures de données (ou les objets) associés sont **inaccessibles de l'extérieur de ces fonctions (ou méthodes)**.

Il faut que le langage permette ça (java le fait).

Lorsqu'un processus exécute l'une de ces méthodes d'un objet A **aucun autre processus ne peut exécuter la même ou une autre de ces méthodes du même objet A.**

Lorsqu'un processus veut exécuter l'une de ces méthodes **il peut être suspendu** jusqu'à ce que l'exécution de cette méthode soit possible (il se met en état bloqué).

Remarque : en java si on écrit une méthode avec l'attribut **synchronized** elle devient une procédure de moniteur.

En Java **tout objet** offre des méthodes **wait** (P) et **notify** (V) pour bloquer et débloquer les processus.

# Moniteurs en Java

```
class RessourcePartagee {  
    private Truc ressource;   
    public synchronized void m1(...) { ... }  
    public synchronized char m2(...) { ... }  
    public int m3(...) { ... }  
}
```

La ressource est inaccessible de l'extérieur (**private**)

Des méthodes permettent d'accéder à la ressource

Un processus P1 exécute `monit.m1(...)`

Pendant l'exécution de `m1` par P1 :

Un autre processus P2 peut exécuter `monit.m3(...)` mais s'il fait `monit.m1(...)` ou `monit.m2(...)` il sera bloqué jusqu'à ce que P1 ait terminé `m1`

# Synchronisation en java

```
class RessourcePartagee {  
    private Truc ressource;  
    public synchronized void m1(...)  
    {  
        ...  
        wait();  
        ...  
    }  
    public synchronized char m2(...)  
    {  
        ...  
        notify();  
        ...  
    }  
    int m3(...) { ... }  
}
```

P1 exécute monit.m1(...)

Si P2 veut exécuter monit.m2(...) il est **bloqué**

- Lorsque P1 arrive au **wait()** il est **bloqué** et P2 est **débloqué**

- Lorsque P2 arrive au **notify()** il **débloquent** P1 qui pourra terminer l'exécution de m1

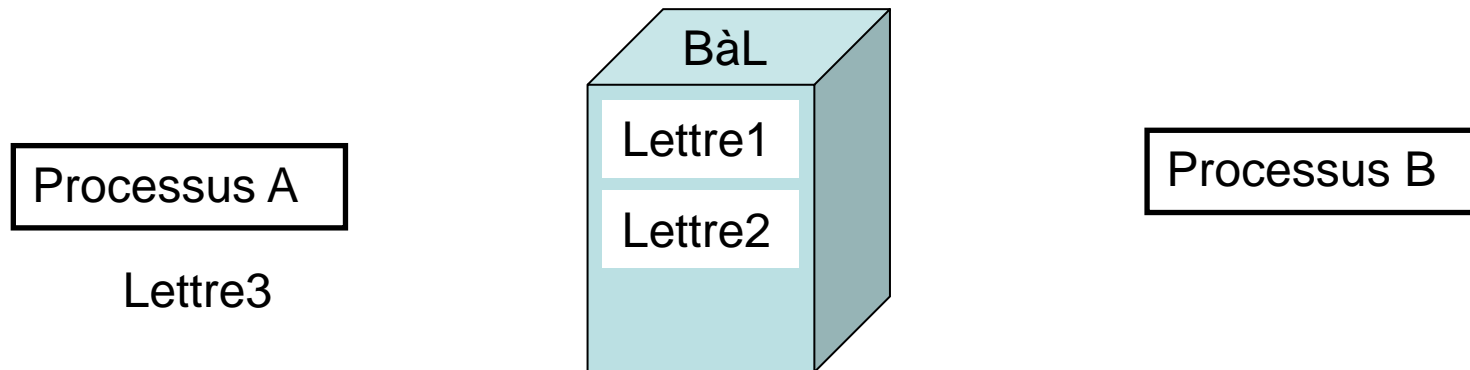
- Remarque : si P2 fait **notifyAll()** il débloquent **tous les processus bloqués**

# Les Systèmes d'Exploitation

1. Introduction
2. Les Processus
3. Les Threads
4. Concurrency entre processus
- 5. Communication entre processus**
6. Les interruptions logicielles (signaux)
7. Gestion du CPU et de la mémoire
8. Ordonnancement
9. Les Entrées/Sorties
10. Le système de fichiers
11. Démarrage d'un système
12. Interblocage
13. Sécurité
14. Systèmes multiprocesseurs

# Communication par messages

C'est le principe de Boîte à Lettres avec deux primitives  
**envoyer** et **recevoir**.



Remarque : c'est utilisable même si les processus sont sur des **machines différentes**. Mais dans ce cas s'ajoutent les problèmes de perte de message et d'identification des interlocuteurs.

# Réalisation de BàL sous UNIX

Les **messages** sont des structures à 2 champs :

- un type de message
- un contenu de message

Le **type de message** est un entier long qui pourra être utilisé par la primitive de lecture pour chercher dans la BàL un message correspondant à ce type.

Le **contenu de message** peut être une chaîne de caractères ou une structure quelconque.

## *Exemples de messages*

```
struct reponse {  
    long type;  
    char message[100];  
}
```

```
struct autre_reponse {  
    long type;  
    msg message;  
}  
typedef struct msg {  
    .....  
}
```

# Création d'une BâL sous UNIX

On utilise la primitive :

**msgget**( key\_t cle, int options)

qui retourne un descripteur de BâL

On retrouve le même principe de clés que pour les sémaphores  
(**IPC\_PRIVATE** ou **ftok**)

Les options sont construites par un OU entre :

- **IPC\_CREAT** si on veut créer la BâL
- **IPC\_EXCL** si on veut la créer à condition qu'elle n'existe pas déjà
- **Oxyz** comme pour les fichiers

Exemple : msgget(cle, IPC\_CREAT|IPC\_EXCL|0666).



# Envoi d'un message sous UNIX

On utilise la primitive :

**msgsnd**( int **descripteur**, struct msgbuf \* **message**, size\_t **taille**,  
int **options**)

qui retourne -1 en cas d'erreur et 0 sinon

Le 1<sup>er</sup> paramètre est le descripteur retourné par **msgget**

Le 2<sup>ème</sup> paramètre pointe sur un message (type + corps)

Le 3<sup>ème</sup> paramètre est la taille en octets du corps du message. On peut utiliser **sizeof**

Le 4<sup>ème</sup> est normalement à 0 mais on peut mettre **IPC\_NOWAIT** si on ne veut pas se bloquer si la BâL est pleine.

# Réception d'un message sous UNIX

On utilise la primitive :

**msgrcv**(int descripteur, struct msgbuf \* message, size\_t taille,  
long **msgtype**, int options)

qui retourne -1 en cas d'erreur et la taille du message reçu sinon

Le 4<sup>ème</sup> paramètre permet de choisir les messages (en fonction du champ type):

- 0 => le 1<sup>er</sup> message disponible
- >0 => le 1<sup>er</sup> message de ce type

Remarque : avec **MSG\_EXCEPT** dans le dernier paramètre on peut demander tout message sauf ce type.

Remarque : avec **MSG\_NOERROR** dans le dernier paramètre on peut demander qu'un message plus long que le paramètre *taille* soit tronqué. Sans cette option il n'est pas extrait de la BàL et il y a erreur.

# Contrôle de BâL sous UNIX

La primitive

**msgctl** (int descripteur, int commande, struct msqid\_ds \***valeur**)

permet de :

- Récupérer la totalité de la BâL (dans **valeur**)
- Modifier certains champs de la BâL (modèle dans **valeur**)
- Supprimer une BâL

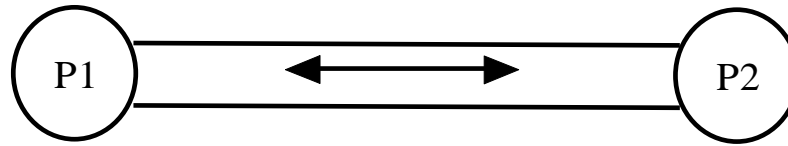
Exemple : pour supprimer la BâL désignée par *ident* on fera

**msgctl**(*ident*, **IPC\_RMID**, NULL)

Le dernier paramètre n'est pas utilisé dans ce cas.

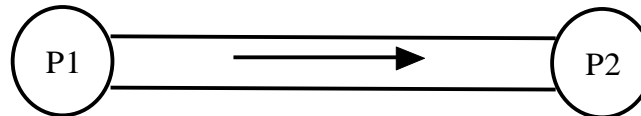
# Communication par canal

Principe : créer un **canal** entre deux processus :

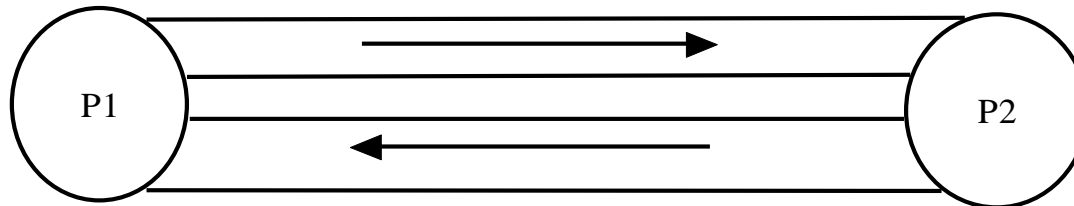


## Réalisation sous UNIX

UNIX propose des **tubes** (pipes) pour ça. Les tubes sont à sens unique



Si on veut communiquer dans les 2 sens il faut créer 2 tubes :



# Communication par canal sous UNIX

Pour que 2 processus connaissent le même canal il faut que :

- soit l'un est le fils de l'autre et il a **hérité du descripteur**
- soit ils peuvent le **désigner de façon unique**

- **pipes** : ça répond au 1<sup>er</sup> cas

On crée un tube par la fct **pipe** et on obtient 2 descripteurs (1 pour lire et 1 pour écrire). L'un des processus utilise l'un et l'autre utilise l'autre

- **ffios** : ça répond au 2<sup>ème</sup> cas

On crée un tube par la fct **mkfifo** en lui donnant un nom et des droits comme pour un fichier et on obtient 1 descripteur

Les données sont écrites/lues par des primitives classiques pour les fichiers de bas niveau (**write** / **read**).

# Communication par mémoire partagée

C'est une **zone de mémoire** utilisable par plusieurs processus

## Réalisation sous d'UNIX

Le problème est toujours le même :

- soit le segment partagé est **connu de tous les processus** car ils sont fils de celui qui l'a créé et ont hérité du descripteur
- soit le segment est partagé par des **processus non liés**

La solution est aussi la même on utilise une **clé** qui est privée ou publique.

# Création d'un segment de mémoire partagée (UNIX)

par **shmget**(cle, taille, options)

- La clé est **IPC\_PRIVATE** ou obtenue par **ftok**
- La taille est en octets
- Les options peuvent être :
  - création **IPC\_CREAT**
  - permission en lecture **SHM\_R**
  - permission en écriture **SHM\_W**

Cette fonction retourne le **descripteur du segment**

## Utilisation :

1. Un processus crée le segment avec **IPC\_CREAT** et une **clé**
2. Les autres récupèrent son identificateur avec la **clé**
3. Le dernier qui l'utilise le **détruit**.

# Accès à un segment de mémoire partagée (UNIX)

par **shmat**(ident, NULL, options)

Cette fonction retourne un pointeur donnant accès à la mémoire partagée.

Le 1<sup>er</sup> paramètre est le descripteur du segment

Le 2<sup>ème</sup> paramètre peut servir à contraindre la façon dont est calculé le pointeur donnant accès à la mémoire (en général on met **NULL** et on laisse UNIX choisir)

Le 3<sup>ème</sup> paramètre permet de préciser les droits **SHM\_R** et/ou **SHM\_W**



# Libération d'un segment de mémoire partagée (UNIX)

par **shmdt**(adresse)

Le processus n'a plus accès à ce segment (mais le segment n'est pas détruit)

Le paramètre est le pointeur retourné par **shmat**

# Destruction d'un segment de mémoire partagée (UNIX)

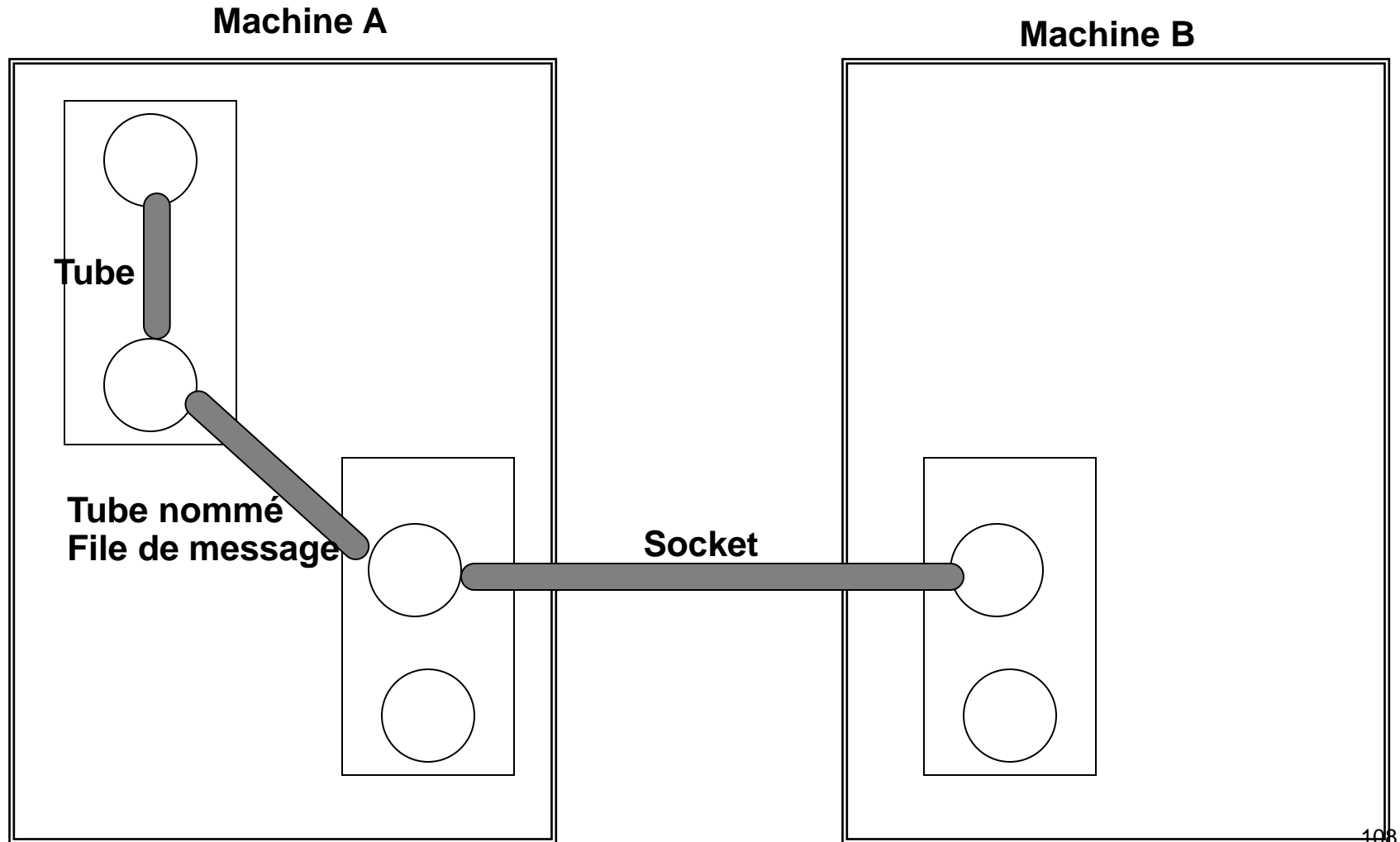
par **shmctl**(*ident*, **IPC\_RMID**, NULL)

*ident* est le descripteur du segment

Remarque : cette fonction peut aussi être utilisée pour récupérer des informations sur le segment ou les modifier (en changeant le 2<sup>ème</sup> paramètre et en utilisant le dernier).

# Communications interprocess

# Schéma général : communications Unix



# Les Communications Entre Processus UNIX

- Intra-tache ou entre threads :
  - partage de variables globales,
  - tas
- Inter processus sur une même machine :
  - signaux,
  - pipes (processus d'une même famille)
  - IPC = Inter Process Communication (Unix system V)
    - Files de messages
    - segment de memoire partagée,
    - sémaphores

# Les Communications Entre Processus UNIX (2)

- Inter tâches distantes :
  - Sockets (TCP, UDP , IP)
  - Appels de procédures distantes (RPC)
  - Architectures bus logiciel (Corba,Java RMI...)
- Outils de haut niveau
  - Systèmes de fichiers répartis (NFS, RFS, AFS)
  - Bases de données réparties (NIS..)

# Communications Inter-machine

# Communications inter machine

- Basé sur les « socket »

# Modèle de référence ISO

- ISO 1 et 2 : Physique et Liaison :
  - ARPANET, SATNET, LAN ethernet
- ISO 3 : Réseau
  - Internet Protocol (IP)
- ISO 4 : Transport
  - TCP (Transmission Control Protocol)
  - UDP (User Datagram)
  - ICMP (Internet Control Message)
  - IGMP (Internet Group Management)



# Modèle de référence ISO (2)

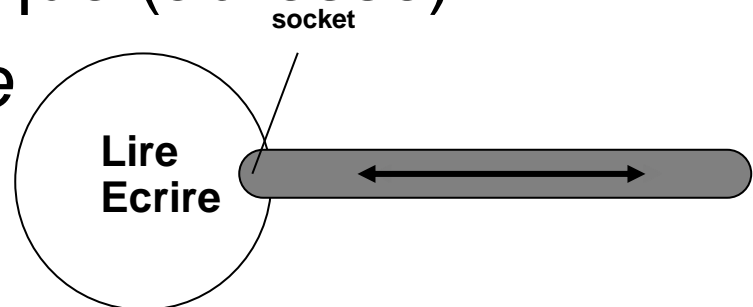
- ISO 5 : Session :
  - RPC
- ISO 6 : Présentation
  - XDR (eXternal Data Representation)
  - ASN.1 (Abstract Syntax Notation number One)
- ISO 7 : Application
  - telnet, ssh
  - ftp, http, smtp
  - dns,nfs

# Les modes de communication

- Connecté :
  - similaire à un pipe (i.e. TCP)
  - Acquittements (augmente la charge réseau)
  - Ni pertes, ni désequencement
- Non connecté :
  - Envois non bloquant (asynchrone) (i.e. UDP)
  - Pertes et désequencement possibles
  - multicast possible efficacement

# Sockets

- Extension de la notion de tube nommé
  - associée à un file descriptor
- Socket = Point extreme de communication
- Caractéristiques :
  - Bidirectionnelle
  - Associé à un nom unique (adresse)
  - Associé à un *domaine*
  - Possède un type



# Domaine de socket

- Les plus courants :
  - AF\_INET : Socket IP, domaine internet IPv4
  - AF\_UNIX : Socket de communication locale
  - AF\_INET6 : IPv6

# Types de service

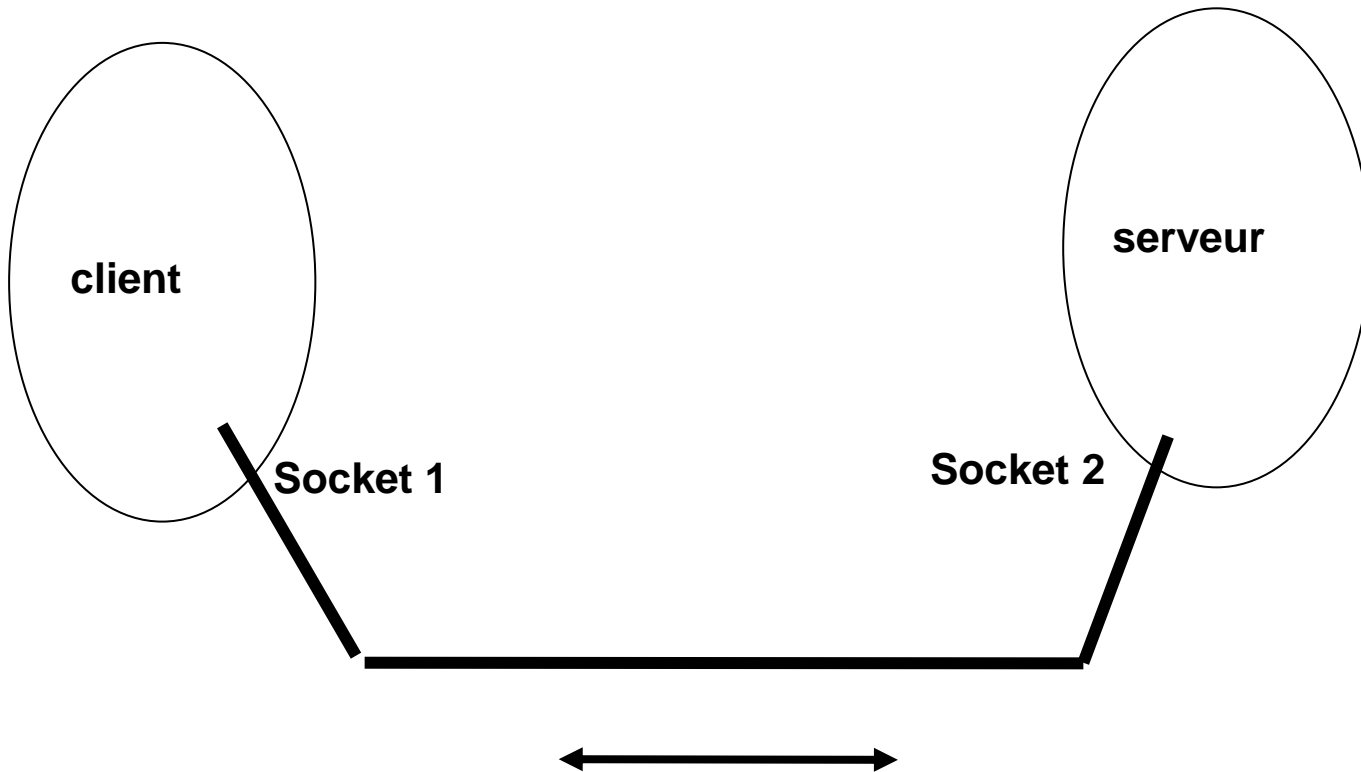
- Les plus courants :
  - SOCK\_STREAM :
    - i.e. TCP (IPPROTO\_TCP)
    - Mode connecté
    - Contrôle de flux (congestion)
    - bidirectionnel
  - SOCK\_DGRAM :
    - i.e. UDP (IPPROTO\_UDP) ou IGMP (IPPROTO\_IGMP)
    - Non connecté
    - Pas de contrôle de flux
    - Transmission par paquet

# Types de Service (2)

- D'autres services :
  - SOCK\_RAW :
    - i.e. ICMP (IPPROTO\_ICMP), protocoles ad hoc (IPPROTO\_RAW)
    - Accès direct à la couche IP
    - Réservé au super-utilisateur
  - SOCK\_SEQPACKET, SOCK\_RDM ...
    - i.e. UDP, ICMP
    - Garantissent différentes propriétés: intégrité, contrôle de flux ...

# Modèle de communication par socket

# Modèle client serveur





# Modèle client serveur

- Le serveur crée une socket, la nomme (bind) et attend des connexions
- Le client crée sa propre socket et se connecte à celle du serveur (doit connaître le nom de la socket serveur)
- En UDP ou TCP, nom de socket = numéro de port + IP
- En AF\_UNIX, nom de socket = nom de fichier

Attention : Serveur = Serviteur, il offre des services au client, le client est celui qui initie la connexion mais les communications sont bidirectionnelles

# socket : phases d'utilisation

1. Création (client / serveur) (socket)
2. Nommage (serveur) (bind)
3. Communication
  1. Communication en mode connecté (TCP)
  2. Communication en mode non connecté (UDP)
4. Fermeture (close/unlink)

# Socket : etape 1 creation

- **socket associée à un descripteur de fichiers**

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

```
#include <sys/socket.h>
```

```
int socket (int domaine, int type, int protocole)
```

- **domaine : AF\_UNIX (local uniquement), AF\_INET**
- **type : SOCK\_STREAM, SOCK\_DGRAM ...**
- **protocole : 0 => choix automatique, (IPPROTO\_TCP, IPPROTO\_UDP)**
- **Retourne un entier (le descripteur de fichier ou -1 en cas d'erreur),**
- **ex:**

```
int s;  
s = socket (AF_INET, SOCK_STREAM, 0);
```

# Socket : etape 2 nommage

- Objectif : Associé un nom à une socket = identifier un serveur de manière unique
- Domaine unix : nom = nom de fichiers
- Domaine internet (inet) :  
nom = <numéro de port, adresse IP>

```
int bind(int sock,
```

```
    struct sockaddr *nom, int lg_nom) ;
```

- sock : numéro socket (retourné par socket())
- nom : le nom de la socket (dépendant du domaine)
- lg\_nom : la longueur de struct sockaddr

# Socket UNIX domain local

# Socket AF\_UNIX : domaine local

- Utilisée pour communiquer dans le domaine local uniquement
- Nom = nom de fichier
- Extension bidirectionnelle de tube nommé
- Portage aisé vers d'autres protocoles (TCP) réseau
- Largement utilisé par les services internes du noyau
- Uniquement sur plateforme UNIX

# Socket AF\_UNIX : domaine local

```
#include <sys/un.h>
#define UNIX_PATH_MAX 108
struct sockaddr_un {
    sa_family_t sun_family;
    /* AF_UNIX (AF_LOCAL) */

    char sun_path[UNIX_PATH_MAX];
    /* Chemin du fichier */
};
```

# Socket AF\_UNIX :

## Exemple de nommage

```
#include <stdlib.h>  #include <sys/types.h> #include <sys/socket.h>
#include <sys/un.h>  #include <stdio.h> #include <string.h>

int main(int argc, const char **argv)
{
    int sock; /* La socket */
    struct sockaddr_un adr; /* Son "nom" */

    sock = socket(AF_UNIX, SOCK_STREAM, 0);
    if (sock == -1) {
        perror("Erreur creation socket");
        exit(1);
    }
    adr.sun_family = AF_UNIX;
    strcpy(adr.sun_path, "/tmp/MaSocket");
    if (bind(sock, (struct sockaddr *) &adr, SUN_LEN(&adr)) == -1) {
        perror("Erreur au nommage");
        exit(2);
    }
    return (0);
}
```

**mode connecté**  
**destruction avec unlink**

```
> ls -l /tmp/MaSocket
```

```
srwxr-xr-x  1 thierry  rech      0 Jan 21 13:01 /tmp/masocket
```



# Nommage dans le domaine Internet

# Socket AF\_INET : Internet

- La plus utilisée
- Compatible toutes plateformes
- Deux protocoles en général : UDP et TCP
- Nommage de socket = (Adr. IP, Num Port)
- Bidirectionnelle
- Sert de base aux services les plus courants (http, ftp, telnet, ssh, ...)

# Nommage dans le domaine AF\_INET

```
#include <netinet/in.h>
struct sockaddr_in {
    short sin_family;           domaine de communication (AF_INET)
    u_short sin_port;           numéro du port
    struct in_addr sin_addr;    adresse IP
    char sin_zero[8];
};

struct in_addr {
    u_long s_addr;
};
```

**Obj : remplir les champs `sin_port` (port) et `sin_addr.s_addr`(IP)**

Remarque sur les adresses :

`sin_addr.s_addr = INADDR_ANY` : machine locale

# sin\_addr.s\_addr : adresse IP

- **Entier sur 32 bits (4 octets)**
  - Classe,
  - identification de réseau,
  - identification d'ordinateur
- **5 classes d'adresse (IPv4)**
  - classe A ( 0 + id\_res sur 7 bits + id\_ord sur 24 bits)  
126 réseaux jusqu'à 16 millions d'ordinateurs
  - classe B ( 10 + id\_res sur 14 bits + id\_ord sur 16 bits)  
16382 réseaux jusqu'à 65536 d'ordinateurs
  - classe C ( 110 + id\_res sur 21 bits + id\_ord sur 8 bits)  
2 millions de réseaux jusqu'à 254 d'ordinateurs
  - classe D ( 1110 + adresse multidestinaire sur 28 bits )
  - classe E ( 11110 + réservé pour usage ultérieur)
- **Délivré par le Network Information Center**

# Format d'adresse IP et conversions

- Adresses codées en format « réseau »
- Fonctions de conversion :
  - **u\_short htons(u\_short) /\* host to network short \*/**
    - Convertir numéro de port
  - **u\_long htonl(u\_long) /\* host to network long \*/**
    - Convertir adresse IP
  - **u\_short ntohs(u\_short) /\* network to host short \*/**
  - **u\_short ntohl(u\_short) /\* network to host long \*/**
- Adresse <-> Chaîne de caractère « a.b.c.d » :
  - **char \*inet\_ntoa(struct in\_addr adr)**
    - /\* adr → chaîne (affichage) \*/
  - **u\_long inet\_addr(char \*chaîne)**
    - /\* chaîne → adr \*/

# Nom d'hôte et adresse IP

```
#include<netdb.h>
```

```
struct hostent *gethostbyname (char * name)
```

- Retourne un pointeur sur un struct hostent,
- Recherche les informations dans
  - /etc/hosts
  - pages jaune (NIS)
  - serveur de noms (DNS)
  - Ordre défini dans /etc/nsswitch.conf

```
struct hostent {
```

```
    char *h_name          /* nom de la machine */
```

```
    char **h_alias /* liste des alias */
```

```
    int h_addrtype /* type de domaine */
```

```
    int h_lenght    /* nombre d'octets d'une adresse */
```

```
    char **h_addr_list /* liste des adresses */
```

```
}
```

```
#define h_addr h_addr_list[0]
```

```
struct hostent gethostbyaddr (char *addr, int len, int  
    domaine)
```

- domaine AF\_INET pour l'instant

# Exemple : name2ip.c

## Conversion nom d'hôte vers IP

```
#include <stdio.h> #include <string.h> #include <sys/socket.h>
#include <netinet/in.h> #include <arpa/inet.h> #include <netdb.h>

int main(int argc, const char **argv)
{
    struct hostent *hp;
    struct in_addr ad;
    if (argc != 2) {
        printf("usage: %s Name\n", argv[0]);
        exit (1);
    }
    hp = gethostbyname((char *)argv[1]);
    if (hp == NULL) {
        printf("host information for %s not found\n", argv[1]);
        exit (2);
    }
    bcopy(hp->h_addr, &ad.s_addr, sizeof(ad.s_addr));
    printf("IP Address = %s\n", inet_ntoa(ad));
    return (0);
}
```

```
>>>name2ip ippc1
IP Address = 134.157.116.1
```

# Exemple 2 : ip2name.c

## Conversion IP vers nom d'hôte

```
#include <stdlib.h> #include <stdio.h> #include <string.h>
#include <sys/types.h> #include <sys/socket.h> #include <netinet/in.h>
#include <arpa/inet.h> #include <netdb.h>

int main(int argc, const char **argv)
{
    in_addr_t addr;
    struct hostent *hp;
    if (argc != 2) {
        printf("usage: %s IP-address\n", argv[0]);
        exit (1);
    }
    if ((int) (addr = inet_addr(argv[1])) == -1) {
        printf("IP-address must be of the form a.b.c.d\n");
        exit (2);
    }
    hp = gethostbyaddr((char *)&addr, sizeof (addr), AF_INET);
    if (hp == NULL) {
        printf("host information for %s not found\n", argv[1]);
        exit (3);
    }
    printf("NAME =\t%s\n", hp->h_name);
    return (0);
}
```

```
>>> ip2name 134.157.116.1
NAME = ippc1.infop6.jussieu.fr
```



# Numéro de port et services

- **Numéro de port `sin_port` = Entier sur 16 bits**
- **Fichier `/etc/services`**
  - Numéros de port utilisés par les services systèmes (à peu près 300),
  - trois champs (type de services, numéro de port, protocole utilisé),
  - FTP (port 21), SSH (port 22), TELNET (port 23), HTTP (port 80)
  - ports 0-1023 système réservés à root

# Numéro de port et services (2)

- **Appels système**
  - **getservbyname** (char \* name, char \*protocole)
  - **getservbyport** (int port, char \* protocole)  
Retourne un pointeur sur un struct servent,  
recherche les informations dans /etc/services
- **struct servent {**

<b>char * s_name ;</b>	nom du service
<b>char ** s_aliases;</b>	liste des alias
<b>int s_port ;</b>	numéro du port
<b>char * s_proto;</b>	protocole à utiliser

**}**

# Socket Connectées : principes

# Socket Connectées

- **Coté serveur**

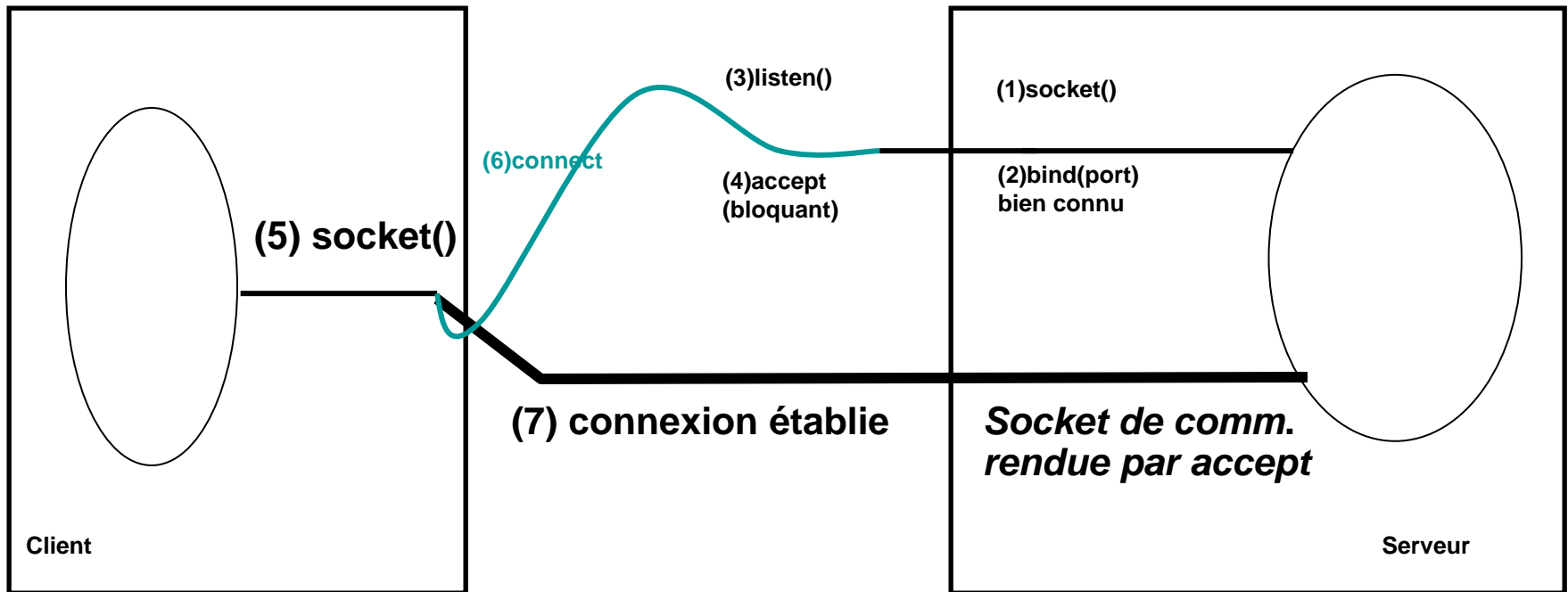
1. Autorisation d'un nombre maximum de connections,
2. Attente d'une connexion,
3. Acceptation d'une connexion,
4. Entrées/sorties,
5. Fermeture de la connexion

- **Coté client**

1. Demande de connexion
2. Entrées/sorties,
3. Fermeture de la connexion

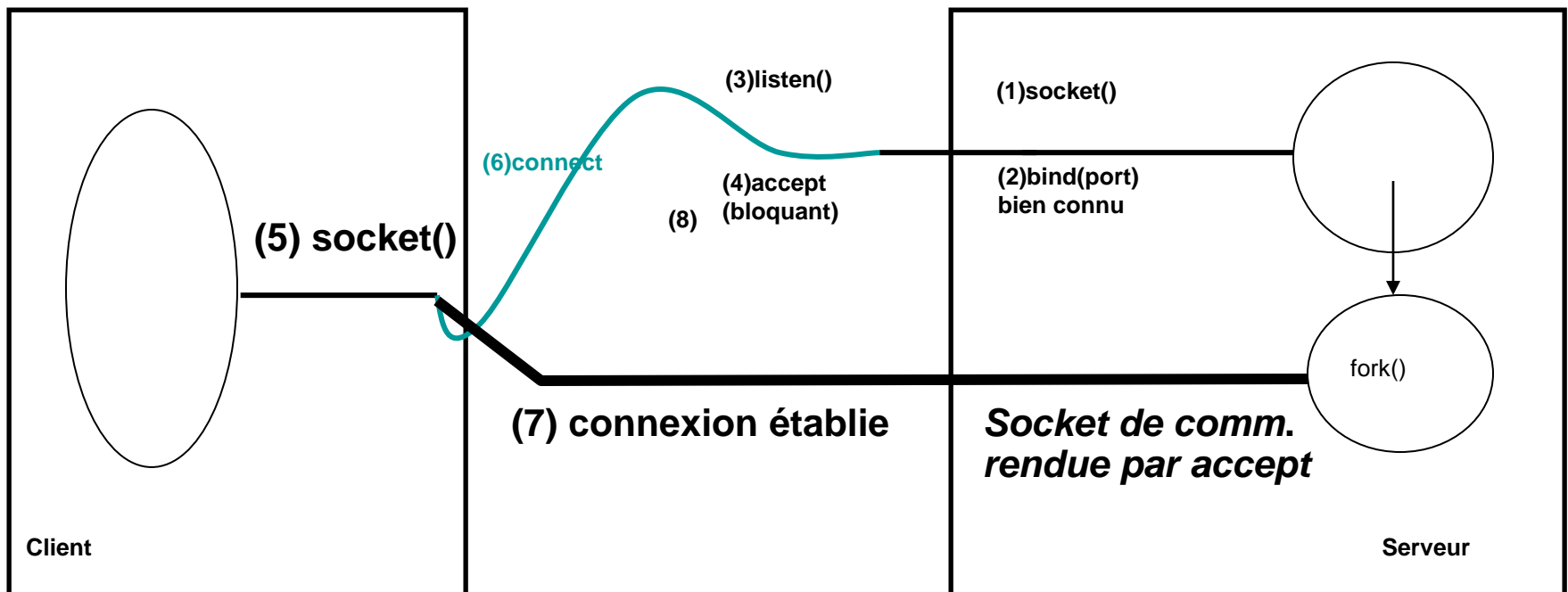
# Etablissement de connexion TCP (2)

- Côté serveur : 1 socket pour demande de connexion, 1 pour communication
- Côté client : 1 socket

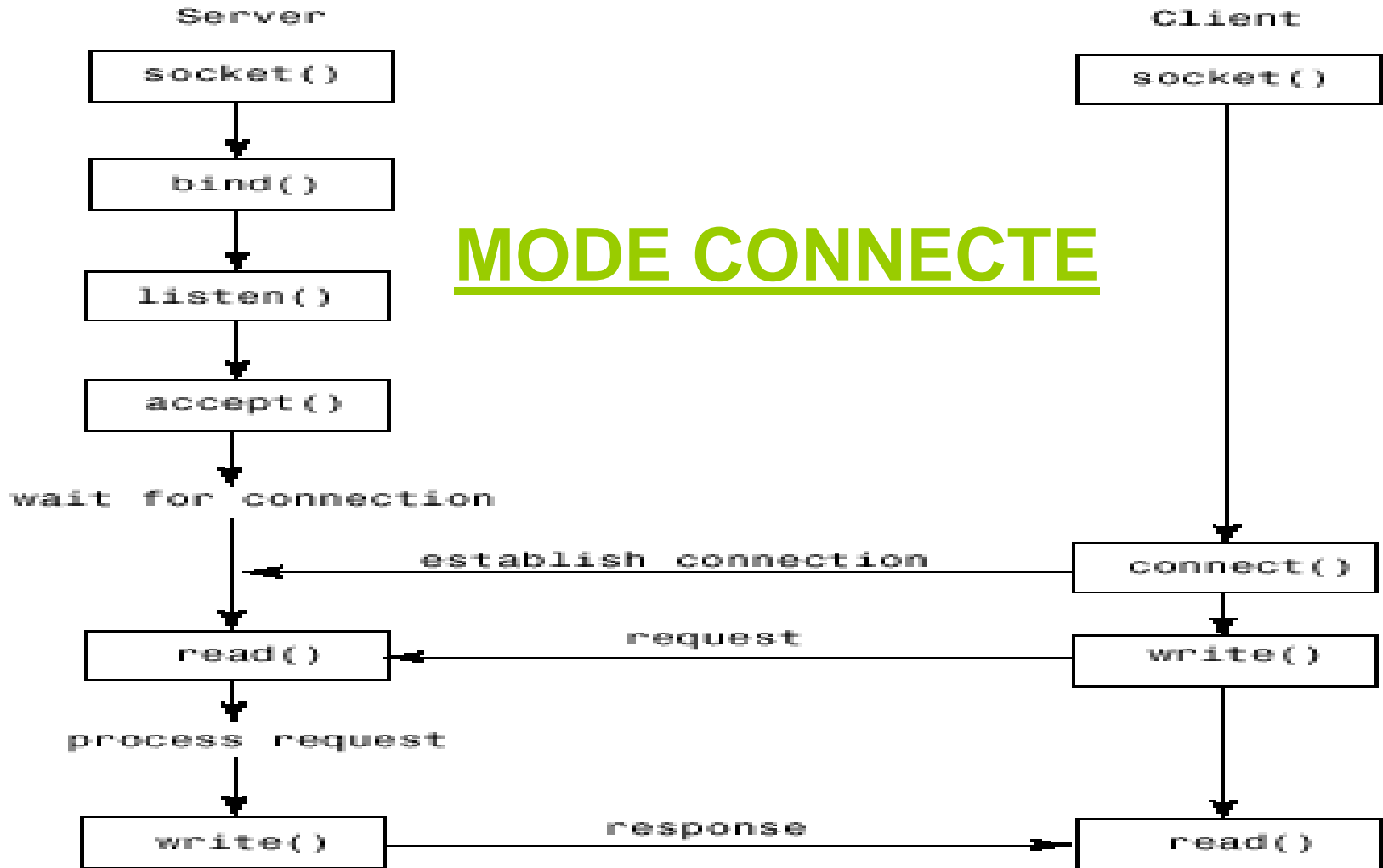


# Etablissement de connexion TCP (3)

- Problème : multi-client
- Solution : listen(taille de file d'attente)
- Multi thread



# Schéma de communication TCP



# Socket connectée : initialisation

- **Côté serveur :**

- int **listen** (int sock, int taille\_file)
  - Création de la file d'attente des requêtes de connexion, non bloquant,
- int **accept** (int sock, struct sockaddr \*addrclt, socklen\_t \*taille)
  - Acceptation d'une connexion, identité du client fournie dans l'adresse addrclt,
  - bloquant,
  - création d'une nouvelle socket « de communication »
    - Renvoie l'identificateur de la socket de communication

- **Côté client :**

- int **connect** (int sock, struct sockaddr \*addrsrv, socklen\_t taille)
  - Demande d'une connexion, bloquant



# Socket connectée : communication

- **int read** (int sock, char \*buffer, int tbuf)
- **int write** (int sock, char \*buffer, int tbuf)
  - Primitives UNIX usuelles,
- **int recv** (int sock, char \*buffer, int tbuf, int attrb)
- **int send** (int sock, char \*buffer, int tbuf, int attrb)
  - Mêmes attributs que pour le mode non connecté,
  - attr = MSG\_OOB , MSG\_PEEK, MSG\_DONTWAIT
- **Bloquants par défaut**

# Socket connectée : deconnexion

- `int shutdown(int s, int how);`
  - `s`: la socket
  - `How = 0` : réception désactivée
  - `How = 1` : émission désactivée
  - `How = 2` : émission et réception désactivées
- Faire suivre le shutdown d'un close
- ex:  
`shutdown(s,2);`  
`close(s);`

# Socket TCP : exemple complet

# Exemple connecté :

## Partie cliente (clientTCP.c) initialisation

```
...
#define PORTSERV 1664
int main(int argc, char *argv[])
{
    struct sockaddr_in dest; /* Nom du serveur */
    struct hostent *hp;
    int sock;
    int fromlen = sizeof(dest);
    int msg;
    int reponse;

    if (argc != 2) {
        fprintf(stderr, "Usage : %s machine\n",
            argv[0]);
        exit(1);
    }

    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    {
        perror("socket");
        exit(1);
    }
}
```

## Exemple connecté :

### Partie cliente (clientTCP.c) connection

```
/* Remplir la structure dest */
if ((hp = gethostbyname(argv[1])) == NULL)
{
    perror("gethostbyname");
    exit(1);
}

bzero((char *)&dest, sizeof(dest));
bcopy(hp->h_addr, (char *)&dest.sin_addr
      ,hp->h_length);
dest.sin_family = AF_INET;
dest.sin_port = htons(PORTSERV);

/* Etablir la connection */
if (connect(sock, (struct sockaddr*)&dest
           , sizeof(dest)) == -1) {
    perror("connect");
    exit(1);
}
```

## Exemple connecté :

### Partie cliente (clientTCP.c) communication

```
msg = 10;
/* Envoyer le message (un entier ici )*/
if (write(sock, &msg, sizeof(msg)) == -1) {
    perror("write");
    exit(1);
}

/* Recevoir la reponse */
if (read(sock, &reponse, sizeof(reponse)) ==
-1) {
    perror("recvfrom");
    exit(1);
}
printf("Reponse : %d\n", reponse);
```

## Exemple connecté : Partie cliente (clientTCP.c) déconnection

```
/* Fermer la connexion */  
shutdown(sock, 2);  
close(sock);  
return(0);  
}
```

# Exemple connecté :

## Partie serveur (servTCP.c) initialisation

```
#define PORTSERV 1664
int main(int argc, char *argv[])
{
    struct sockaddr_in sin; /* Nom de la socket de
    connexion */
    int sc ;                /* Socket de connexion */
    int scom;               /* Socket de communication
    */
    struct hostent *hp;
    int fromlen = sizeof exp;
    int cpt=0;

    /* creation de la socket */
    if ((sc = socket(AF_INET,SOCK_STREAM,0)) < 0) {
        perror("socket");
        exit(1);
    }
```



## Exemple connecté :

### Partie serveur (servTCP.c) nommage

```
bzero((char *)&sin, sizeof(sin));
sin.sin_addr.s_addr =
    htonl(INADDR_ANY);
sin.sin_port = htons(PORTSERV);
sin.sin_family = AF_INET;

/* nommage */
if (bind(sc, (struct sockaddr*)&sin,
        sizeof(sin)) < 0) {
    perror("bind");
    exit(1);
}
listen(sc, 5);
```

## Exemple connecté :

### Partie serveur (servTCP.c) attente

```
/* Boucle principale */
for (;;) {
    if ( (scom = accept(sc
                        , (struct sockaddr*)&exp
                        , &fromlen)
        ) == -1) {
        perror("accept");
        exit(3);
    }
    if (fork() != 0)
        /* le pere se remet en attente */
        continue;
    else
        {
            /* Création d'un processus fils
             qui traite la requete */
        }
}
```

# Exemple connecté :

## Partie serveur (servTCP.c) traitement

```
/* Processus fils */

if (read(scom,&cpt, sizeof(cpt)) < 0) {
    perror("read");
    exit(4);
}
/**/ Traitement du message /**/
cpt++;
if (write(scom, &cpt, sizeof(cpt)) == -1) {
    perror("write");
    exit(2);
}

/* Fermer la communication */
shutdown(scom,2);
close(scom);
exit(0);
} /* fin du fils */
} /* fin de boucle principale */
```

## Exemple connecté : Partie serveur (servTCP.c) terminaison

```
/* pas de connection  
=> pas de shutdown */
```

```
close(sc);  
return 0;  
}
```

## Exemple connecté : Execution

```
>SockTCPS &  
[1] 5489
```

```
>netstat -a | grep 1664
```

```
tcp 0 0 localhost.localdom:1664 *:* LISTEN
```

```
>SockTCPC
```

Réponse 1

```
>netstat -a | grep 1664
```

```
tcp 0 0 localhost.localdom:1664 *:* LISTEN
```

# Socket UDP : exemple complet

# Exemple non-connecté : initialisation

## Partie cliente (clientUDP.c)

```
#define PORTSERV 1664 /* Port du serveur */

int main(int argc, char *argv[])
{
    int reponse;
    struct sockaddr_in dest;
    struct hostent *hp;
    int sock;
    int fromlen = sizeof(dest);
    char message[80];

    /* Le nom de la machine du serveur est passé en argument */
    if (argc != 2) {
        fprintf(stderr, "Usage : %s machine \n", argv[0]);
        exit(1);
    }

    if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }
}
```

# Exemple non-connecté : choix destinataire

## Partie cliente (clientUDP.c)

```
/* Remplir la structure dest */

if ((hp = gethostbyname(argv[1])) == NULL)
{
    perror("gethostbyname");
    exit(1);
}
bzero((char *)&dest, sizeof(dest));
bcopy(hp->h_addr, (char *)&dest.sin_addr,
      hp->h_length);
dest.sin_family = AF_INET;
dest.sin_port = htons(PORTSERV);

/* Construire le message ... */
strcpy(message, "MESSAGE DU CLIENT");
```



# Exemple non-connecté : communication

## Partie cliente (clientUDP.c)

```
/* Envoyer le message */
if (sendto(sock,message,strlen(message)+1,0,(struct
sockaddr *)&dest,
    sizeof(dest)) == -1) {
    perror("sendto");
    exit(1);
}

/* Recevoir la reponse */
if (recvfrom(sock,&reponse,
    sizeof(reponse),0,0,&fromlen)
    == -1) {
    perror("recvfrom");
    exit(1);
}
printf("Reponse : %d\n", reponse);
```

# Exemple non-connecté : terminaison

## Partie cliente (clientUDP.c)

```
/* pas de connection  
=> pas de shutdown */
```

```
close(sock) ;  
return 0 ;  
}
```

# Exemple non-connecté : initialisation

## Partie serveur (servUDP.c)

```
#define PORTSERV 1664
int main(int argc, char *argv[])
{
    struct sockaddr_in sin; /* Nom de la
                             socket du serveur */
    struct sockaddr_in exp; /* Nom de l'expediteur */
    struct hostent *hp;
    int sc ;
    int fromlen = sizeof(exp);
    char message[80];
    int cpt = 0;

    /* creation de la socket */
    if ((sc = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
        perror("socket");
        exit(1);
    }
}
```

## Exemple non-connecté : nommage

### Partie serveur (servUDP.c)

```
/* remplir le « nom » */
bzero((char *)&sin, sizeof(sin));
sin.sin_addr.s_addr =
    htonl(INADDR_ANY);
sin.sin_port = htons(PORTSERV);
sin.sin_family = AF_INET;

/* nommage */
if ( bind(sc, (struct sockaddr *)&sin,
        sizeof(sin)) < 0) {
    perror("bind");
    exit(2);
}
```

# Exemple non-connecté : attente requête

## Partie serveur (servUDP.c)

```
/** Reception du message */  
if (  
    recvfrom( sc,message,  
              sizeof(message),0,  
              (struct sockaddr*)&exp,  
              &fromlen)  
    == -1) {  
    perror("recvfrom");  
    exit(2);  
}
```

**BLOQUANT**

# Exemple non-connecté : analyse message

## Partie serveur (servUDP.c)

```
/**Affichage de l'expediteur**/  
printf("Exp : <IP = %s,PORT = %d> \n",  
    inet_ntoa(exp.sin_addr),  
    ntohs(exp.sin_port));  
/* utiliser gethostbyaddr pour un nom  
d'hote au lieu de l'IP */  
  
/**Affichage Message**/  
printf("Message : %s \n", message);  
  
/*** Traitement ***/  
cpt++;
```

# Exemple non-connecté : réponse et fin

## Partie serveur (servUDP.c)

```
/** Envoyer la reponse **/  
    if (sendto(sc, &cpt, sizeof(cpt), 0,  
              (struct sockaddr *)&exp,  
              fromlen )  
        == -1) {  
        perror("sendto");  
        exit(4);  
    }  
    close(sc); /*pas de shutdown*/  
    return (0);  
}
```

## Exemple non-connecté : Execution

```
>SockUDPS &
```

```
[1] 5879
```

```
>netstat -a | grep 1664
```

```
udp  0  0 localhost.localdom:1664 *:*
```



# Broadcast et Multicast

# Diffusion : Broadcast UDP

- Envoie un message à tous les hotes d'un sous-réseau
- Client :
  - mettre en destinataire une adresse de broadcast
  - = IP + tous les bits d'adresse de machine à 1
  - Nécessite la permission de diffuser (broadcast)
- Serveur
  - identique servUDP ci-dessus

# Multidiffusion : Multicast UDP

- Multicast : implementation efficace coté tables de routage pour la multi-diffusion vers un groupe de machines
- Une classe d'adresses dédiées :
  - Classe D : 224.0.0.1 à 239.255.255.255
- Réception (serveur):
  - Abonnement à un groupe = adresse de classe D
  - `recvfrom`
  - Désabonnement
- Emission (client):
  - choisir l'adresse du groupe
  - `send`

# Exemple : Multicast

## Emetteur ClientMC.c

```
#define MON_ADR_DIFF "225.0.0.10"
#define PORTSERV 3328

int main(int argc, char *argv[])
{
    struct sockaddr_in dest;
    int sock;
    char message[80];

    if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
        perror("socket"); exit(1);
    }

    /* Remplir la structure dest */
    bzero((char *)&dest, sizeof(dest));
    dest.sin_addr.s_addr = inet_addr(MON_ADR_DIFF);
    dest.sin_family = AF_INET;
    dest.sin_port = htons(PORTSERV);

    /* Contruire le message ... */
    strcpy(message, "MESSAGE DU CLIENT");

    /* Envoyer le message */
    if ( sendto(sock, message, strlen(message)+1, 0, (struct sockaddr *)&dest,
                sizeof(dest)) == -1) {
        perror("sendto"); exit(1);
    }
    close(sock);
    return(0);
}
```

# Exemple : Multicast abonnement

## Réception ServMC.c

```
#define MON_ADR_DIFF "225.0.0.10" /* Adresse multi-cast */
#define PORTSERV 7200 /* Port du serveur */

int main(int argc, char *argv[])
{
    int sock;
    struct sockaddr_in sin;
    struct ip_mreq imr; /* Structure pour setsockopt */
    char message[80];

    if((sock = socket(AF_INET, SOCK_DGRAM, 0)) < 0){
        perror("socket");
        exit(1);
    }

    imr.imr_multiaddr.s_addr = inet_addr(MON_ADR_DIFF);
    imr.imr_interface.s_addr = INADDR_ANY;

    if(setsockopt(sock, IPPROTO_IP, IP_ADD_MEMBERSHIP, (char
        *)&imr,
                sizeof(struct ip_mreq)) == -1){
        perror("setsockopt");
        exit(2);
    }
}
```

**Abonnement**

# Exemple : Multicast nommage

## Réception ServMC.c

```
/* remplir le « nom » */
bzero((char *)&sin, sizeof(sin));
sin.sin_addr.s_addr =
    htonl(INADDR_ANY);
sin.sin_port = htons(PORTSERV);
sin.sin_family = AF_INET;

/* nommage */
if (bind(sock,
        (struct sockaddr *)&sin,
        sizeof(sin))
    < 0) {
    perror("bind");
    exit(3);
}
```

# Exemple : Multicast lecture

## Réception ServMC.c

```
/* Reception du message */
if (recvfrom(sock,message,
             sizeof(message),0,
             NULL,NULL)
    == -1)
{
    perror("recvfrom");
    exit(4);
}

printf("Message reçu :%s\n",
       message);
close(sock);
return (0);
}
```

# Conclusion sockets



# Choix d'un protocole : UDP ou TCP ?

- TCP : STREAM
  - Ni perte, ni déséquencelement
  - Mode flux (taille non bornée)
  - Coûteux (connexion, contrôle d'erreur...)
- UDP : DGRAM
  - Pertes et déséquencelements possibles (sur un LAN, le problème se pose peu)
  - Taille limitée par paquet
  - Performant (surcouche par rap. à IP)
  - Multicast possible

# « Visualisation » des socket

- Commande netstat
  - -x : unix
  - -u : udp
  - -t : tcp
  - -a : toutes
- Plus d'informations pour root

# TCP/IP Utilitaires

- arp
- domainname
- ftp
- dig
- hostname
- netstat
- netconfig
- route
- telnet
- traceroute
- whois
- ping
- ifconfig
- finger...

# Utilitaires DNS (noms d'hotes)

- `hostname`
- `nslookup`
- `host`
- `dig`



# Les Systèmes d'Exploitation

1. Introduction
2. Les Processus
3. Les Threads
4. Concurrency entre processus
5. Communication entre processus
- 6. Les interruptions logicielles (signaux)**
7. Gestion du CPU et de la mémoire
8. Ordonnancement
9. Les Entrées/Sorties
10. Le système de fichiers
11. Démarrage d'un système
12. Interblocage
13. Sécurité
14. Systèmes multiprocesseurs

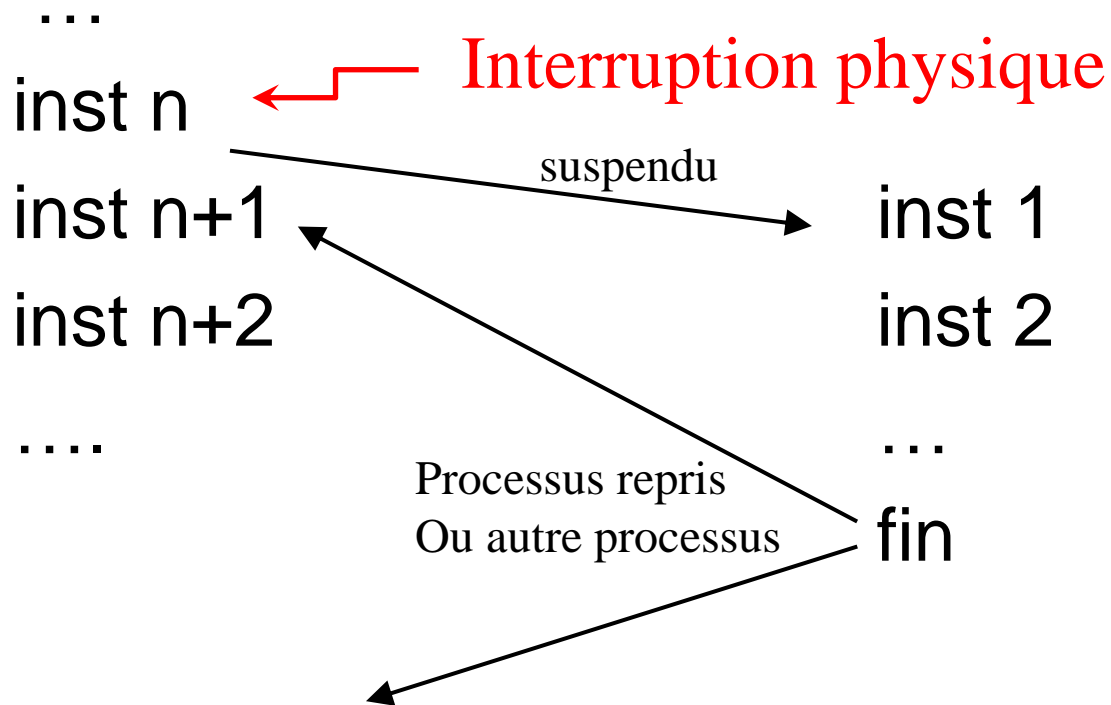
# Principe

- Une **interruption physique** permet de traiter un événement physique (périphérique, délai, erreur ...).
- Une **interruption logicielle** permet de traiter un événement non physique. Par exemple un événement généré par un processus pour indiquer qu'il a besoin de quelque chose ou qu'il a fini ...
- L'arrivée d'une interruption physique **suspend le processus en cours** pour que le processus destinataire traite l'événement (**immédiat**).
- L'arrivée d'une interruption logicielle **ne suspend pas le processus en cours** : elle sera traité par le processus destinataire lorsqu'il sera actif (**non immédiat**)

# Interruption physique

Processus en cours

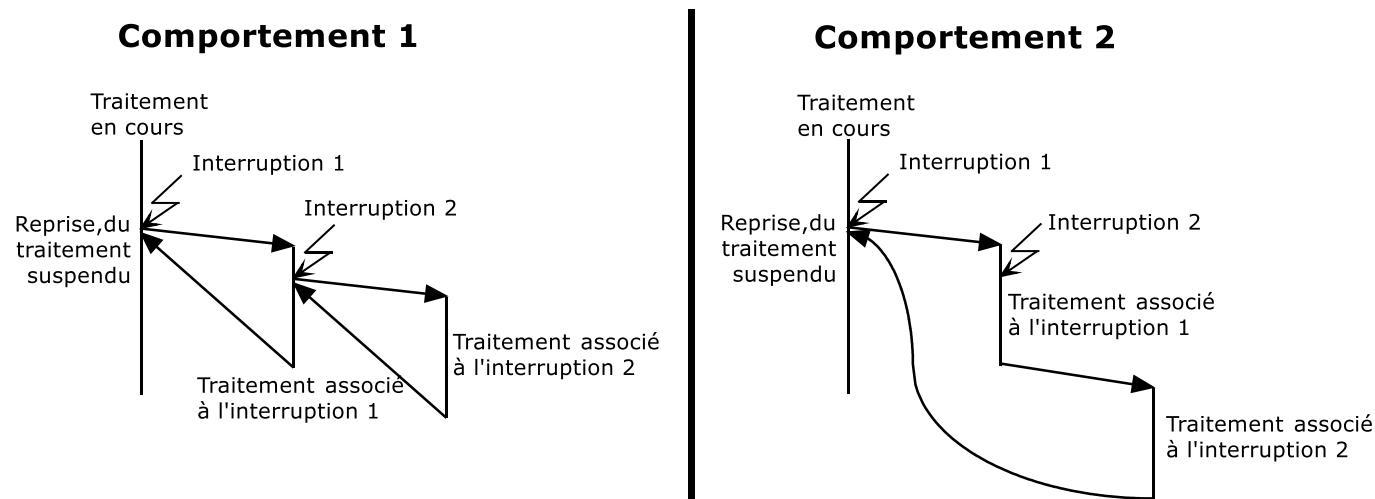
Code de l'IT





# Interruptions physiques

- Que se passe t-il si une interruption survient pendant l'exécution du code associé à une interruption ?



- Dépend du niveau de **préemption** des interruptions 1 et 2 :
  - Si le niveau de préemption de l'interruption 2 est **inférieur** à celui de la 1 : **comportement 1**
  - Si le niveau de préemption de l'interruption 2 est **supérieur ou égal** à celui de la 2 : **comportement 2**

# Les signaux

Les SE gèrent, pour chaque processus, un ensemble de bits correspondant aux **signaux** possibles (1 bit / signal).

- **Délivrance d'un signal** (par le processus émetteur) = mise à 1 du bit correspondant

- **Association d'une action à un signal** (par le processus récepteur) :

A chaque bit est associée l'@ d'une fonction qui sera exécutée quand le signal arrivera. Lorsqu'un processus est créé, le SE place des @ associées à des fonctions standard. Certaines font quelque chose d'autres rien.

Si le processus récepteur est actif, quand le bit est mis à 1 cela provoque :

1. La mise à 0 du bit
2. L'exécution immédiate de l'action associée (appel de fonction classique)

Sinon, lorsque le processus récepteur sera relancé, il découvrira que le bit est à 1 et exécutera les étapes 1 et 2 ci-dessus.

# Les signaux (exemple d'UNIX)

Par exemple sous UNIX quand on tape la commande : **kill -sS p** on envoie le signal de nom **S** au processus de pid **p**.

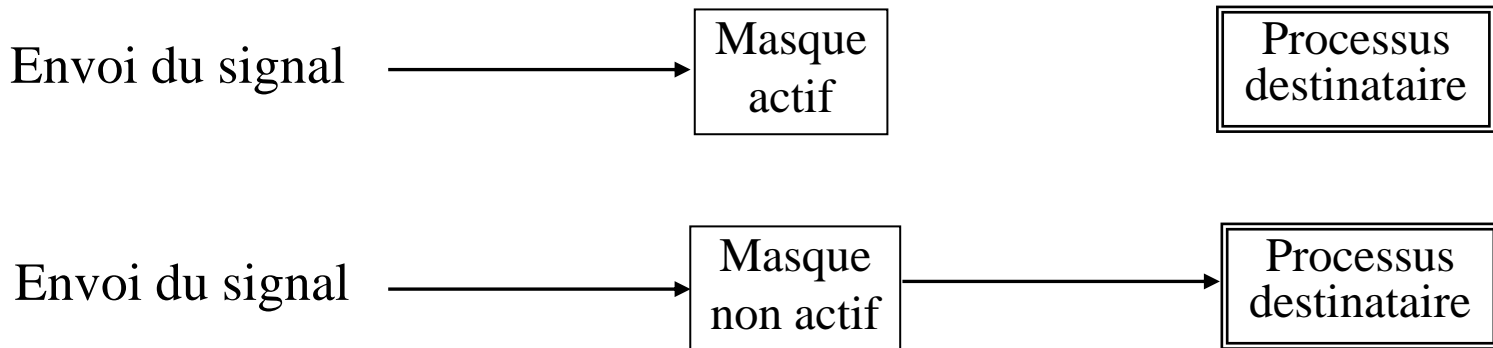
Certains signaux comme **SIGKILL** (9) ont une fonction associée qui tue le processus. D'autres comme **SIGUSR1** ne font rien.

Le processus peut, pour la plupart des signaux, **changer l'@ de la fonction associée** pour y mettre une fonction à lui (pas **SIGKILL**).

Exemple : quand on tape une touche au clavier on a une IT physique qui regarde le caractère tapé. Si c'est une touche normale elle est mise dans un buffer pour lecture ultérieure par `scanf` ou `getchar`. Si c'est Ctrl C le SE délivre le signal **SIGINT** au processus en cours. La fonction normalement associée tue le processus mais ce dernier peut l'avoir détournée.

# Masquage d'un signal

A chaque **bit de signal** est associé un autre bit (le masque). Si ce bit est positionné le signal **n'est pas pris en compte**.



C'est utile si un processus veut éviter de traiter certains signaux à certains moments.

# Communication par signaux

2 étapes :

- **Envoi** du signal par un processus
- **Traitement** de ce signal par un autre processus

La durée entre les 2 étapes est **imprévisible** car le processus récepteur ne prendra en compte le signal que quand il sera exécuté par un CPU

La fonction UNIX **pause** permet à un processus de se mettre en sommeil jusqu'à ce qu'il reçoive un signal.

La fonction UNIX **alarm** permet à un processus de programmer une temporisation pour qu'elle lui envoie un signal **SIGALRM** dans  $n$  secondes.

*Remarque* : Si le signal **SIGALRM** n'est pas masqué ou intercepté, sa réception terminera le processus.

# Les signaux sous UNIX

- Envoi d'un signal par **kill**(pid, n° de signal)
- Masquage d'un signal par **sigprocmask**(opération, nouveau\_masque, ancien\_masque)
- Association d'une fonction à un signal par **signal**(n° de signal, fonction )

Remarque : La norme POSIX définit 1 seule primitive pour tout faire :

**sigaction**(n° de signal, action, ancienne action)

Où action est une structure de données permettant de définir aussi bien un masque qu'une fonction associée.

# Les Systèmes d'Exploitation

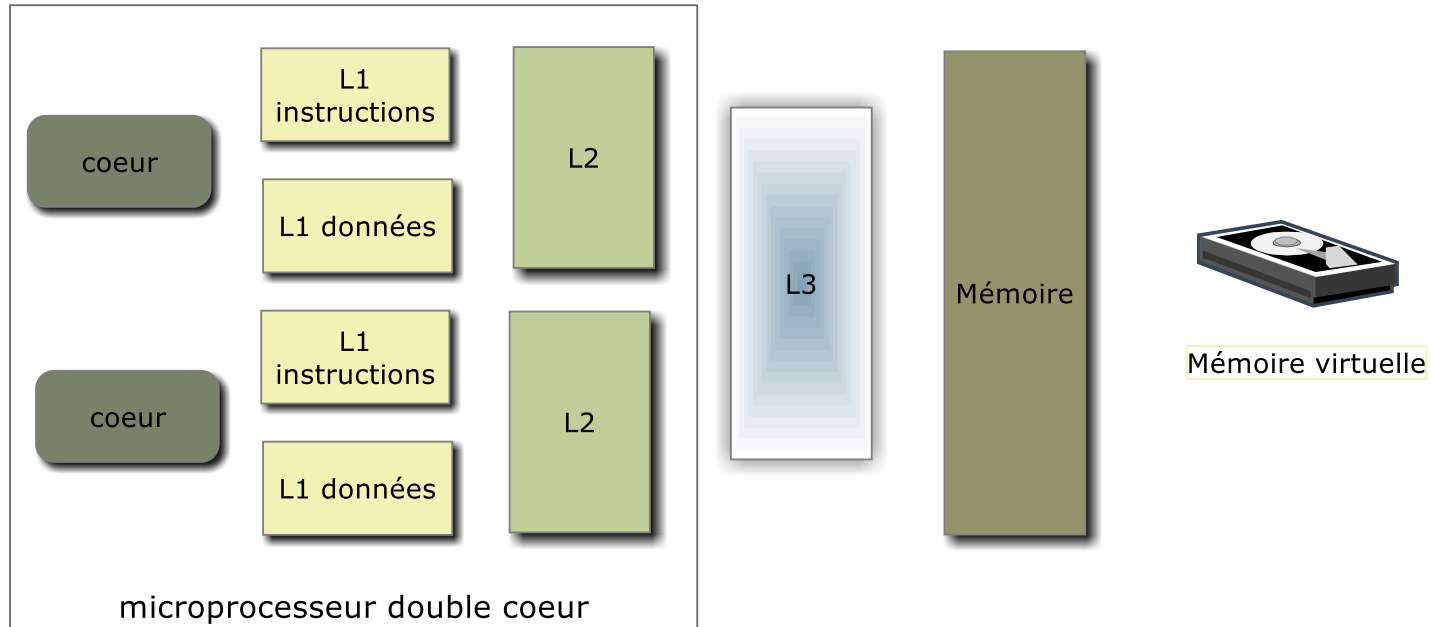
1. Introduction
2. Les Processus
3. Les Threads
4. Concurrency entre processus
5. Communication entre processus
6. Les interruptions logicielles (signaux)
- 7. Gestion du CPU et de la mémoire**
8. Ordonnancement
9. Les Entrées/Sorties
10. Le système de fichiers
11. Démarrage d'un système
12. Interblocage
13. Sécurité
14. Systèmes multiprocesseurs

# Gestion du CPU

- L'horloge du CPU est réglable en fréquence entre  $F_{min}$  et  $F_{max}$ .
- La consommation du CPU et sa température sont proportionnelles à cette fréquence. Au-delà d'une certaine valeur ( $F_{seuil}$ ) la température augmente dangereusement.
- Le SE règle la fréquence d'horloge en fonction de la charge du CPU (économie d'énergie). Lorsque le CPU est très chargé la fréquence d'horloge peut dépasser  $F_{seuil}$ .
- Tant que la fréquence d'horloge est entre  $F_{min}$  et  $F_{seuil}$  le CPU fonctionne normalement, au-delà de  $F_{seuil}$  il est en mode Turbo mais ne peut pas y rester trop longtemps.
- Le SE surveille la température du CPU (capteur intégré) et baisse temporairement la fréquence d'horloge si cette température dépasse un seuil critique (70 à 80°) jusqu'à qu'elle retrouve une valeur correcte.
- Dans un CPU multicœur l'horloge est la même pour tous les cœurs.



# Hiérarchie de mémoire



Chaque cœur :

- Dispose de ses caches de niveau L1 (2 x quelque dizaines de Ko) et de niveau L2 (quelques Mo)
- Partage le cache de niveau L3 (quelque dizaines de Mo) et la mémoire (quelques Go)

La mémoire virtuelle est sur le disque (quelques dizaines de Go)

# Caches et mémoire virtuelle

- Propriétés :
  - **Localité spatiale** : l'accès à une donnée située à une adresse  $A$  a de fortes chances d'être suivi d'un accès à une donnée située à une adresse très proche de  $A$ .
  - **Localité temporelle** : l'accès à une zone mémoire à un instant donné a de fortes chances de se reproduire rapidement
- Les **caches** sont gérés par du matériel
  - Le SE peut configurer ce matériel (zones à ne pas mettre en cache par exemple pour les sémaphores)
- La **mémoire virtuelle** est gérée par le SE
  - Détection d'absence
  - Choix des zones de mémoire à remplacer
  - Copies disque  $\leftrightarrow$  mémoire

# Gestion de la mémoire

Le SE doit gérer la **distribution de la mémoire** entre les différents processus ainsi que la **mémoire virtuelle**.

Remarque : Pour que ça marche il faut qu'un processus puisse s'exécuter n'importe où en mémoire.

Normalement, quand on compile un programme, les noms de variables et d'étiquettes sont remplacés par des **adresses fixes** donc il faudrait recompiler le programme selon l'adresse où on le met et celle où on met ses variables

Les microprocesseurs modernes permettent d'éviter ça en utilisant des **registres de segment ou de base**. Toute adresse est définie **relativement au contenu de ces registres**

Il suffit alors au SE d'**initialiser ces registres** selon l'endroit de la mémoire où il met le code du processus et ses variables.

# Gestion de l'espace mémoire

On divise la mémoire en **blocs** de tailles différentes (**unités d'allocation**).

Quand un processus démarre on lui **alloue** un bloc qui lui suffit.  
S'il n'y en a pas il **attend**.

Le SE gère l'utilisation de la mémoire. Il y a 2 façons de faire :

- Par **table de bits** : chaque **unité d'allocation** a 1 bit dans cette table qui indique si elle est ou non allouée
- par **liste chaînée** : chaque élément de liste correspond à une **unité d'allocation** il indique si elle est libre ou pas et pointe sur la suivante.

# Choix de la zone à allouer à un nouveau processus

On a 3 algorithmes :

- **Algo de la 1<sup>ère</sup> zone libre** : on recherche la **1<sup>ère</sup> unité libre de taille suffisante**.  
On peut améliorer cet algorithme en reprenant la recherche à partir du point où on s'était arrêté la fois précédente et non du début.
- **Algo de meilleur ajustement** : on cherche l'**unité libre dont la taille est la plus proche** de celle demandée. Ça défavorise les petits processus mais c'est compensé par le fait qu'ils peuvent être choisis pour des unités petites alors que les autres ne peuvent pas. Certains SE donnent du bonus aux processus qui n'ont pas démarré pour cette raison => quand ils ont accumulé assez de bonus ils démarrent même s'ils utilisent une unité peu adéquate.
- **Algo rapide** : on gère des listes d'unités libres selon leur taille => **on parcourt directement la bonne liste et on prend la 1<sup>ère</sup> unité** même si sa taille n'est pas optimale.

# La mémoire virtuelle

Les adresses manipulées par les instructions correspondent à des **adresses virtuelles**. La **MMU** (Memory Management Unit) se charge de transformer ces adresses en **adresses physiques**.

On dispose de **beaucoup plus** d'adresses virtuelles qu'il n'y a de mémoire physique.

Lors de cette correspondance on peut détecter qu'une adresse virtuelle ne correspond à **aucune adresse physique**, dans ce cas la MMU **génère une IT** qui sera prise par le SE pour gérer cette absence. Le SE utilise **le disque** pour compléter la mémoire :

Le SE va :

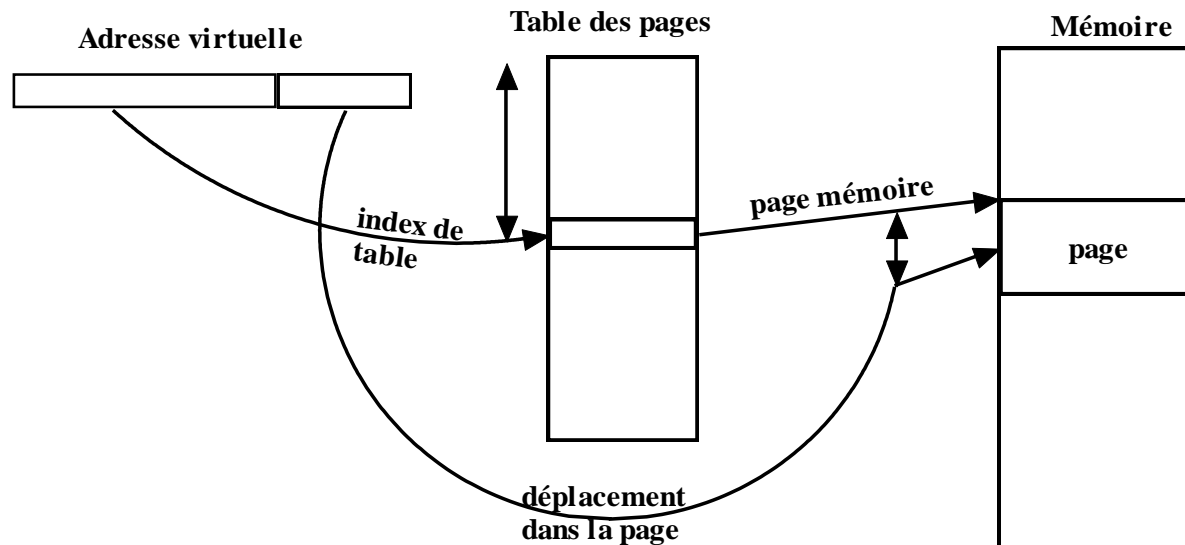
1. choisir une page de mémoire à libérer
2. transférer le contenu de cette page sur le disque
3. transférer dans la page libérée la zone disque correspondant à l'adresse virtuelle demandée.

# Table des pages

Les adresses manipulées par les instructions correspondent à des **adresses virtuelles**.  
La **MMU** (Memory Management Unit) se charge de transformer ces **adresses virtuelles** en **adresses physiques** (adresses de la mémoire).

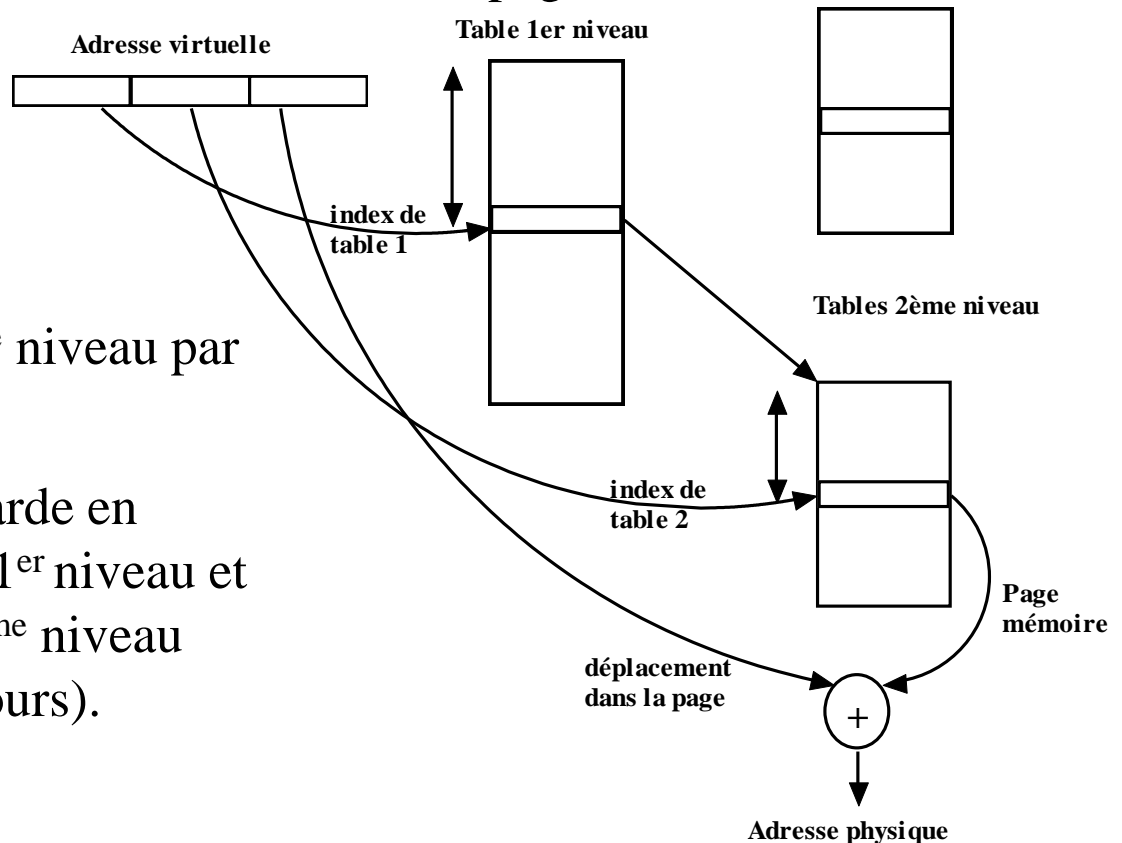
On dispose de **beaucoup plus** d'adresses virtuelles qu'il n'y a de mémoire physique.  
Il faut gérer une **table des pages** pour faire la correspondance

@ **virtuelle** → @ **physique** avec l'éventuel transfert **disque ↔ mémoire** si la page contenant cette adresse n'est pas présente en mémoire.



# Table des pages

La table des pages peut devenir très grande => difficile à garder en mémoire.  
On résoud ce problème en utilisant une table des pages à 2 niveaux :



On crée une table de 2<sup>ème</sup> niveau par processus.

L'intérêt c'est qu'on ne garde en mémoire que la table de 1<sup>er</sup> niveau et les quelques tables de 2<sup>ème</sup> niveau utilisées (processus en cours).



# Description de la table des pages

Dans **chaque entrée d'une table de pages** de 2<sup>ème</sup> niveau on trouve :

- Un **bit de présence** (la page est ou pas présente en mémoire)
- L'**adresse de la page en mémoire physique** (si elle est présente)
- Des bits de protection (lecture, écriture, exécution)
- 2 bits correspondant à l'utilisation de la page (**modifiée, référencée**). Ils serviront lors des changements de page (modifiée => à écrire sur disque avant libération, référencée => à ne pas libérer si possible).
- Un bit pour **inhiber les caches**, permet de ne pas utiliser la mémoire cache pour cette page (sémaphores par exemple).

Remarque : pour accélérer le processus de passage @ virtuelle → @ physique le matériel (MMU) propose en général une petite mémoire associative appelée **TLB** (Translation Lookaside Buffer) qui conserve les entrées récemment utilisées des tables de pages. Si on ne trouve pas dans la TLB on cherche dans les tables de pages **et on met à jour la TLB**.

# Problème de taille de la table des pages

Lorsque la taille des @ virtuelles augmente **la taille et le nombre** de tables de pages augmente aussi.

Exemple : @ virtuelle sur 64 bits si on fait des pages mémoire de 4 Ko on en a  $2^{52}$  même en admettant que chaque entrée de table de page soit sur 8 bits les tables occuperaient **4500 tera octets !!!!**

Certes **toutes les @ virtuelles possibles ne sont pas utilisées** et donc toutes les tables de 2<sup>ème</sup> niveau ne sont pas présentes en mémoire mais le problème de la place occupée reste important.

**Solution** : L'une des approches actuelles de ce problème est de faire des **tables inversées** c'est à dire qu'il existe une entrée de table par page de mémoire physique. La mémoire physique étant plus petite la table reste de taille raisonnable (max = taille mémoire / taille page)

Problème : la transformation @ virtuelle → @ physique suppose de parcourir la table des pages inversée jusqu'à trouver la bonne entrée ou constater qu'elle n'y est pas => c'est moins rapide (**d'où l'intérêt de la TLB**).

# Remplacement de page

**Le SE a en charge de remplacer les pages de la mémoire physique.**

On utilise les bits '**modifiée**' et '**référéncée**' de la table des pages.

Ces bits sont mis à jour lors des accès à la page : (référéncée  $\leftarrow$  1 et modifiée  $\leftarrow$  1 si écriture). On trouve plusieurs algorithmes de remplacement des pages :

**NRU** (Non Récemment Utilisé) : Le bit référencé est remis à 0 périodiquement par le SE (c'est le R de NRU).

NRU recherche une page **NON référencé ET NON modifiée** (00) et l'enlève (une au hasard s'il y en a plusieurs).

S'il n'en trouve pas il cherche **NON référencé ET modifiée** (01) puis **référéncée ET NON modifiée** (10) puis **référéncée ET modifiée** (11).

**FIFO** : On classe les pages par ordre d'allocation et on cherche la page la plus ancienne non référencée (càd ayant le bit référencée à 0).

Pendant cette recherche toutes les pages qui la précèdent (donc plus anciennes mais référencées) sont mises en fin de liste (comme si elles étaient récentes) mais marquées comme non référencées

Cette méthode est appelée **seconde chance** c'est à dire qu'on considère que la page est remise en jeu et pourra rester à condition qu'elle soit à nouveau référencée.

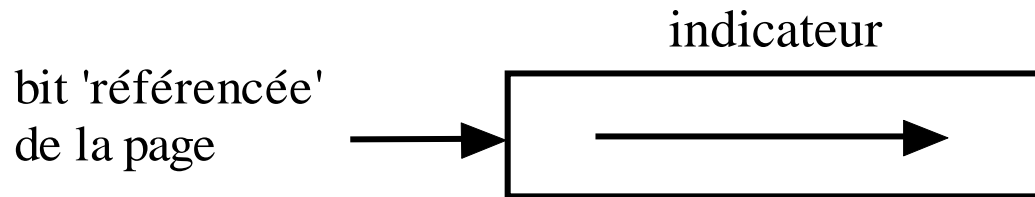
# Remplacement de page

**LRU** (Least Recently Used): remplace la page qui n'a pas été référencée **depuis le plus longtemps**.

Il faut trouver un algorithme pour savoir laquelle c'est.

On utilise un indicateur par page initialisé à 0 quand la page est nouvelle.

Périodiquement le SE effectue l'opération suivante :



Ainsi cet indicateur contient d'autant plus **de bits à 1 en fort poids** que sa page a été plus récemment utilisée donc plus son contenu représente un entier faible plus la page est candidate à être remplacée.

# Optimisation de la gestion de la mémoire

Quand un processus **démarre** il ne dispose pas en mémoire physique des pages contenant ses variables, sa pile et son code => il va créer de nombreux **défauts de pages**.

Au bout d'un moment il aura tout ce qu'il faut en mémoire et fonctionnera sans problèmes.

Ceci est normal au démarrage d'un processus mais quand un processus est **suspendu (bloqué)** les pages qu'il utilisait vont, petit à petit, être **virées** de la mémoire pour laisser de la place aux autres.

Quand il va **redémarrer (débloqué)** il va à nouveau créer des **défauts de pages**.

Les SE essayent d'éviter ça en chargeant d'avance, quand le processus redémarre, **l'ensemble de son espace de travail** tel qu'il était quand le processus a été suspendu.

# Le rôle du SE dans la gestion des pages

Il y a 4 moments clés :

## **Création d'un processus**

- Déterminer la taille du programme et de ses données
- Allouer de la mémoire physique
- Créer une table de pages de niveau 2 pour lui
- Prévoir de la place sur le disque pour ses pages

## **Exécution d'un processus**

- Charger en mémoire tout ou partie de l'espace de travail de ce processus
- Mettre à jour la TLB pour ce processus

## **Fin d'un processus**

- Supprimer sa table des pages de niveau 2
- Libérer les pages en mémoire physique qu'il utilisait
- Libérer la place sur le disque prise par ses pages

## **Défaut de page**

- Localiser sur disque la page correspondant à l'@ virtuelle ayant causé le défaut
- Choisir une page en mémoire physique libre ou en libérer une
- Copier sur disque la page libérée si elle a été modifiée
- Charger la page du disque dans cette page de mémoire libre
- Reculer le CO du processus pour qu'il ré exécute l'instruction ayant provoqué le défaut de page (IT).

# Allocation de mémoire par un processus

On a vu que quand un processus démarre il se voit **allouer de la mémoire** pour :

- son code
- sa pile
- ses variables

Mais souvent un programme a besoin de plus de mémoire dont la taille est dynamiquement définie => il doit pouvoir **demander au SE de la mémoire supplémentaire.**

La demande de mémoire se traduit dans les langages à objets (C++) par **new** et sa libération par **free**

# Exemple d'UNIX

**UNIX offre les primitives pour demander de la mémoire:**

**malloc** accepte un paramètre qui est la taille demandée et retourne un pointeur sur la zone allouée. Ce pointeur peut être NULL si l'allocation a échoué.

**calloc** fait pareil mais initialise la zone allouée à 0

**realloc** accepte un paramètre supplémentaire qui est un pointeur sur une zone déjà allouée. Son rôle est de modifier la taille de cette zone sans en perdre le contenu

**free** accepte en paramètre un pointeur sur une zone déjà allouée et la libère.

**Problème de la taille de mémoire à demander** : En général un programme alloue de la mémoire pour y mettre une variable et il ne connaît pas la taille en octets de cette variable mais il sait de quel type elle est.

C propose une fonction **sizeof(type)** qui donne la taille en octets d'un type.



# Les Systèmes d'Exploitation

1. Introduction
2. Les Processus
3. Les Threads
4. Concurrency entre processus
5. Communication entre processus
6. Les interruptions logicielles (signaux)
7. Gestion du CPU et de la mémoire
- 8. Ordonnancement**
9. Les Entrées/Sorties
10. Le système de fichiers
11. Démarrage d'un système
12. Interblocage
13. Sécurité
14. Systèmes multiprocesseurs

# Ordonnancement

Objectif : **Choisir le prochain processus à exécuter de façon optimale.**

Un **changement de processus** suppose :

- D'enregistrer le contexte du processus actuel (registres, config mémoire, ...)
- De charger le contexte du prochain processus (registres, config mémoire, ...)

En raison de l'utilisation de caches mémoire on aura souvent des défauts de présence

- ⇒ Transferts cache ↔ mémoire. Même chose pour la mémoire virtuelle.
- ⇒ Cela crée une perte de temps importante
- ⇒ Il faut le faire à bon escient.

Un processus fait :

- Du traitement (CPU + mémoire)
- Des E/S (UE)

Avec les progrès sur les CPU le traitement devient moins critique :

- Les E/S sont bloquantes => on **change de processus**
- Le traitement n'est pas bloquant => on utilise le **temps partagé**

On parle **d'ordonnancement préemptif.**

# Critères essentiels de l'ordonnancement

- Attribuer à chaque processus un **temps équitable**
- Essayer d'**occuper au mieux toutes les ressources** de la machine
- Pour les **systèmes interactifs** : répondre rapidement aux actions de l'utilisateur
- Pour les **systèmes temps réel** : respecter les délais
- Pour les **systèmes multimédias** : prévisibilité c'est à dire régularité d'exécution (par exemple 25 images / s pour une vidéo)

Certains de ces critères sont **contradictaires** par exemple : utiliser au mieux les ressources de la machine peut se faire en faisant tourner 2 processus (un qui fait beaucoup de calculs et un qui fait beaucoup d'E/S) mais on ne garantit pas l'équité pour les autres.

# Ordonnancement sur les systèmes classiques

Plusieurs approches sont possibles :

## 1. Tourniquet (round robin)

On a une liste de processus, on passe de l'un à l'autre en tournant et on change :

- soit par **blocage** du processus (E/S, sémaphore, sleep, ...)
- soit par **fin de son quota de temps**

Problème : choix du quantum de temps :

- **Trop court**  $\Rightarrow$  beaucoup de changements  $\Rightarrow$  perte importante de temps en changements
- **Trop long**  $\Rightarrow$  temps de réponse long (pas bon pour interactif ou multimédia)

On choisit, en général, un quantum de **20 à 50 ms** mais ça dépend du type d'utilisation (120 ms sur serveurs Windows  $\Rightarrow$  limite la perte de temps car pas d'interactivité).

Remarque : un cœur de processeur moderne effectue 5 milliards d'instructions par seconde (100 à 250 millions d'instructions par quota : **ça avance !**)

# Ordonnancement sur les systèmes classiques

**Plusieurs approches sont possibles :**

## **2. Priorités**

On associe des priorités aux processus selon leur **importance**.

Les processus de plus haute priorité sont **traités en premier**.

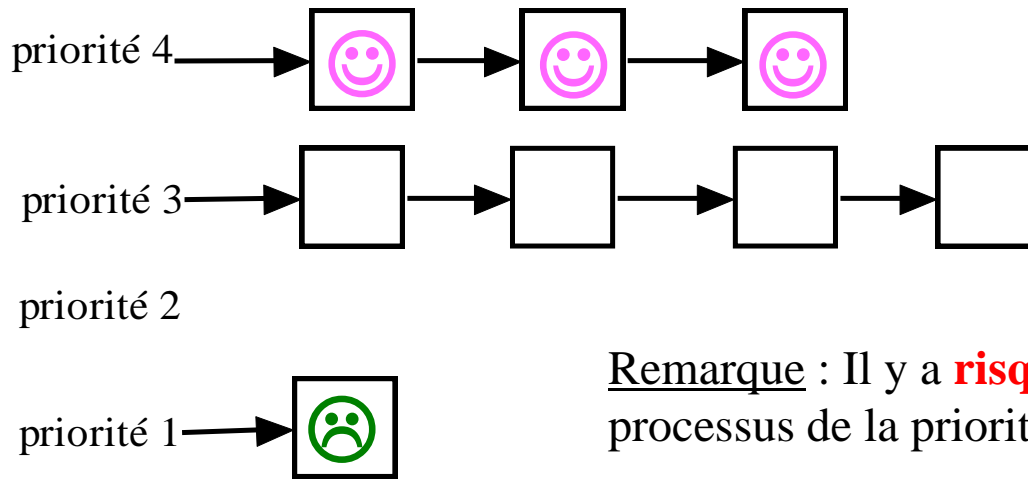
Un processus de priorité N ne sera exécuté **que s'il n'y a aucun** processus de priorité supérieure à N exécutable

La priorité d'un processus peut être **modifiée** à chaque fin de quantum par le SE

On peut calculer la priorité en fonction de la **charge** qu'induit un processus

Par exemple un processus qui n'utilise que 10% de son quantum de temps parce qu'il fait beaucoup d'E/S pourra avoir une priorité élevée puisqu'il ne bloquera pas le CPU trop longtemps.

# Ordonnancement avec priorités



On va faire tourner le **round robin** sur les processus de priorité 4, quand il n'y en a plus (et s'il n'en est pas arrivé de nouveaux) on passera à ceux de priorité 3 etc.

Remarque : Il y a **risque de famine** pour les processus de la priorité la plus basse

On peut résoudre ce problème en **changeant les priorités des processus** au fur et à mesure de leur exécution et/ou en modifiant la durée pour chaque priorité.

## Exemple :

Les processus de priorité 4 utilisent 1 quantum

Les processus de priorité 3 utilisent 2 quanta

Les processus de priorité 2 utilisent 4 quanta

etc

Quand un processus tourne depuis longtemps il **baisse en priorité**.

# Exemple d'UNIX

Le chiffre associé à la priorité est d'autant plus élevé que la priorité est plus faible.

Unix classe les processus dans des **files** correspondant à leur priorité. Les priorités les plus élevées sont négatives : elles sont réservées aux processus système.

UNIX fait du **round robin** sur chaque file mais ne passe à la file de priorité < que s'il n'y a plus rien dans celles de priorité >.

Ceci risquerait de créer de la **famine** pour les processus de priorité basse => UNIX modifie la priorité de chaque processus chaque seconde. La nouvelle priorité est calculée par :

$$\text{priorité} = \text{tpsCPU} + \text{nice} + \text{base}$$

**tpsCPU** : nombre de cycles d'horloge qui ont été attribués au processus dans les dernières secondes. Donc si un proc n'a pas eu beaucoup de tps CPU sa priorité monte.

**nice** : partie de la priorité laissée à l'utilisateur (commande **nice**) on peut aller de -20 à +20 mais les utilisateurs ne vont que de 0 à 20, par défaut c'est 0 càd la meilleure valeur => un utilisateur ne peut que baisser la priorité de ses processus.

**base** : utilisé pour augmenter la priorité des processus qui ont fait une E/S. C'est une valeur négative qui dépend du type d'E/S demandé, dans l'ordre descendant :

E/S disque	E/S clavier	affichage	attente de proc fils
------------	-------------	-----------	----------------------

# Exemple d'UNIX

Certains UNIX (LINUX) distinguent 3 classes de processus :

1. les processus **non préemptibles** sauf par un de la même classe
2. les processus **préemptibles** en temps partagé
3. les **threads** préemptibles en temps partagé

L'idée est que :

- La 1<sup>ère</sup> classe correspond à des **processus temps réel**
- Les 2 autres à des **processus ou threads temps partagé** quand un processus (thread) démarre il dispose d'un quantum de temps dépendant de sa priorité mais elle diminue au fur et à mesure qu'il consomme du temps CPU

ATTENTION : bien que désignés par "temps réel" les procs de la 1<sup>ère</sup> classe ne le sont pas car on n'a aucune garantie. Tout ce que l'on sait c'est qu'ils sont plus prioritaires que les autres donc ont plus de chance de s'exécuter plus souvent

**Ce n'est pas du temps réel.**



# Exemple de Windows

Les priorités de Windows sont :

A l'inverse d'Unix, le chiffre associé à la priorité est d'autant plus élevé que la priorité est plus forte.

Pour les **processus**, (6 valeurs) :

- real time
- high
- above normal
- normal
- below normal
- idle

Pour les **threads**, (7 valeurs) :

- time critical
- highest
- above normal
- normal
- below normal
- lowest
- idle

7 priorités pour un thread et 6 pour le processus qui le contient => 42 priorités différentes.

Mais windows ne gère que **32 priorités différentes** ce qui fait que la priorité finale d'un thread peut être la même dans des cas différents.

# Priorités Windows

Windows ne tient pas compte de la priorité des processus mais de celle des threads composée avec celle de leur processus :

Thread Processus	Idle	Lowest	Below Normal	Normal	Above Normal	Highest	Time Critical
Idle	1	2	3	4	5	6	15
Below Normal	1	4	5	6	7	8	15
Normal	1	6	7	8	9	10	15
Above Normal	1	8	9	10	11	12	15
Hight	1	11	12	13	14	15	15
Real time	16	22	23	24	25	26	31

Windows gère donc 32 listes et recherche un thread exécutable en commençant par la liste de priorité max (31) non vide avec un **round robin**.

# Priorités Windows (suite)

- Les priorités 16 à 31 sont appelées **temps réel** et sont réservées au **système** puisqu'elles ne sont obtenues que par des threads inclus dans des processus de priorité **real time**. **En fait ça n'est pas temps réel du tout !**
- Les priorités 1 à 15 sont réservées aux **utilisateurs**.
- La priorité 0 est réservée à un **processus de RAZ des pages mémoire inutilisées** (*sécurité contre la récupération de données*).
- La priorité -1 est réservée à un **processus de fond** (*processus inactif du système*).

Comme UNIX, Windows **augmente la priorité** d'un thread qui a fait une E/S (sans dépasser 15). Par exemple de :

- 1 pour un disque
- 6 pour le clavier
- 8 pour la carte son
- 2 pour une attente sur sémaphore
- etc

Ces bonus sont **diminués de 1** chaque fois que le processus a épuisé son quantum jusqu'à ce qu'il retrouve sa priorité initiale

Windows utilise un truc pour éviter la **famine** : quand le temps depuis lequel un processus n'a pas eu le CPU dépasse un certain seuil, il le met en priorité 15 pendant 2 quanta. Après quoi il retrouve sa priorité initiale.

# Ordonnancement sur les systèmes temps réel

On distingue en général 2 grandes catégories :

- À tolérance nulle (**temps réel dur**)
- Avec tolérance (**temps réel mou**)

Les algos sont dits **statiques** s'ils assignent à chaque processus une priorité fixe et **dynamiques** si cette priorité peut changer.

- Algo **RMS** (Rate Monotonic System) **statique**  
On donne une priorité qui dépend de la fréquence de l'événement que traite le processus.  
Un processus de priorité plus élevée préempte un processus actif de priorité < dès qu'il doit s'exécuter (principe des ITs et des priorités d'IT).
- Algo **EDF** (Earliest Deadline First) **dynamique**  
Chaque processus indique à quelle échéance il doit s'être exécuté. Celui dont l'échéance est la plus proche devient le plus prioritaire.

# Les Systèmes d'Exploitation

1. Introduction
2. Les Processus
3. Les Threads
4. Concurrency entre processus
5. Communication entre processus
6. Les interruptions logicielles (signaux)
7. Gestion du CPU et de la mémoire
8. Ordonnancement
- 9. Les Entrées/Sorties**
10. Le système de fichiers
11. Démarrage d'un système
12. Interblocage
13. Sécurité
14. Systèmes multiprocesseurs

# Les Entrées/Sorties

- Le SE contrôle tous les organes d'E/S : il en récupère les **interruptions** et fournit les **fonctions d'accès**.

## *Aspects matériels*

- **Contrôleur de périphérique**

C'est un dispositif physique que l'on programme par ses **registres internes** :

- commandes (opération à exécuter et paramètres de cette opération, ...)
- états (opération en cours, opération terminée, erreurs, ...)
- données (valeurs allant vers ou venant du périphérique)

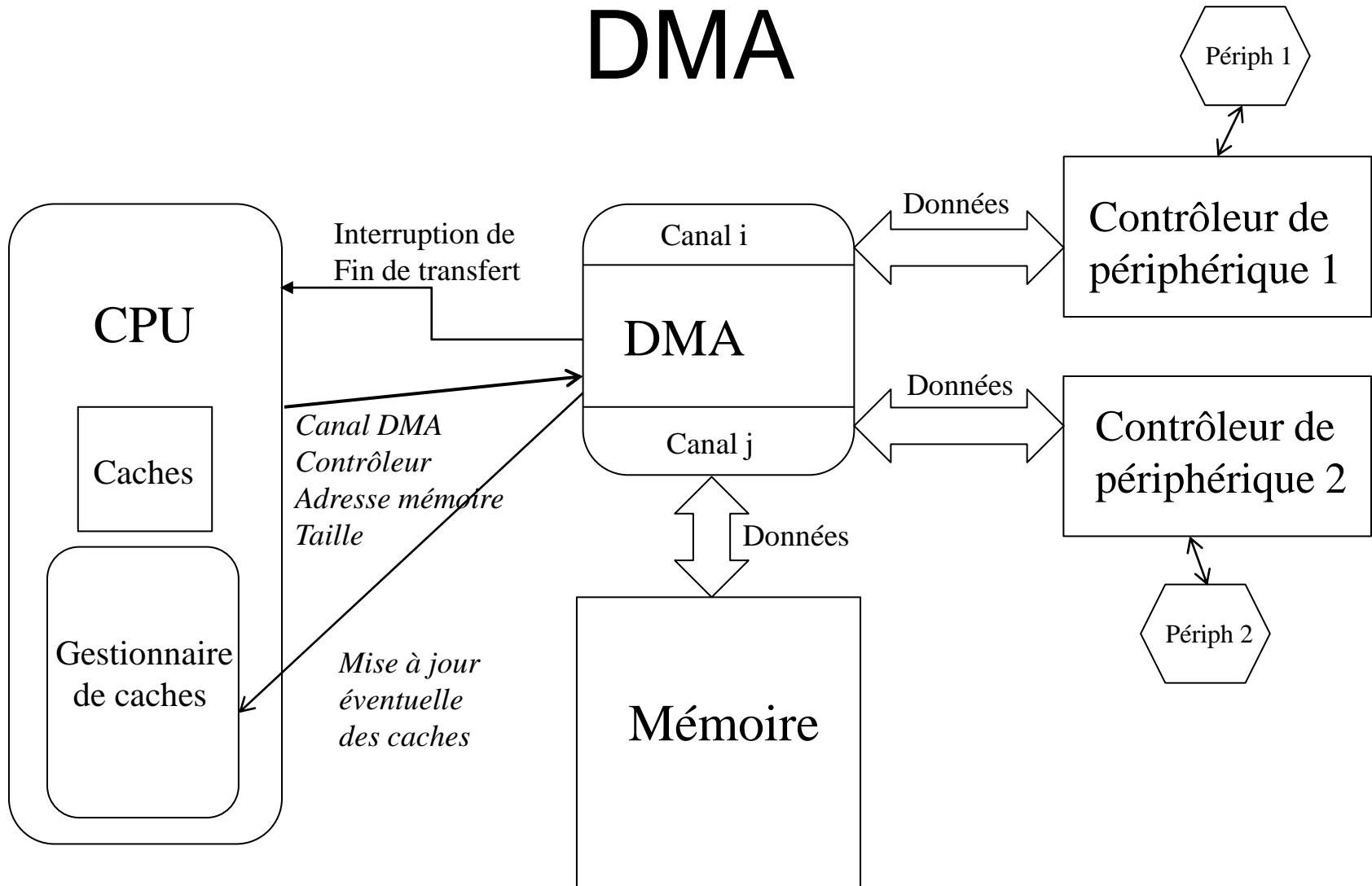
- **DMA**

Pour faciliter les transferts d'info avec les périphériques et soulager le CPU on utilise un **DMA** (Direct Memory Access). Il s'agit d'un dispositif physique auquel le CPU peut demander de faire un transfert entre registres de données du contrôleur de périphériques et mémoire dans un sens ou dans l'autre.

C'est alors le DMA qui **règle sa vitesse sur celle du périphérique** et génère une **IT** vers le CPU quand le transfert est terminé (pendant le transfert le CPU est libre)

Un contrôleur de DMA offre plusieurs **canaux** (une dizaine) càd qu'il peut faire plusieurs transferts en même temps avec plusieurs périphériques.

# DMA



# Les Entrées/Sorties

## *Aspects matériels*

### Les interruptions

Le fonctionnement est le suivant :

- génération d'un **signal physique** sur une ligne spéciale unique (IRQ)
- émission d'un **n° d'IT** (le n° dépend de l'origine de l'IT).

Le SE a assigné un n° à chaque contrôleur de périphérique, au DMA , à la MMU, etc. lors de leur initialisation pour éviter les n°s en double.

Remarque : certaines ITs ne sont pas liées à un contrôleur (division par 0, ...) et ont un n° fixe.

Quand le CPU prend en compte l'IT il :

- **suspend le processus en cours** et sauvegarde son **contexte**
- récupère l'@ où aller dans une **table de vecteurs d'ITs** (table initialisée par le SE au démarrage) en utilisant le n° d'IT comme index
- **exécute la procédure prévue** qui acquitte l'IT auprès du contrôleur

Quand c'est terminé il reprend le **contexte du processus** interrompu et **le continue** sauf si l'action associée à cette IT provoque un autre comportement (par exemple lors d'une IT de fin de quantum de temps).



# Les Entrées/Sorties

## *Aspects matériels*

### **Les interruptions sur un multicœur**

- Il existe un dispositif matériel **APIC** (Advanced Programmable Interrupt Controller) qui redirige le signal physique d'interruption et le n° vers l'un des cœurs du processeur.
- Au démarrage le premier cœur reçoit toutes les interruptions
- L'APIC est programmable par le SE qui peut choisir :
  - D'envoyer toutes les IT sur le même cœur
  - De distribuer les ITs selon leur origine sur différents cœurs

Le SE modifie dynamiquement cette programmation en fonction de la charge des cœurs. Unix possède un fichier `/proc/interrupts` qui indique, pour chaque interruption, combien de fois elle a traitée par chacun des cœurs.

# Les Entrées/Sorties

## *Aspects logiciels*

Le principe de fonctionnement est **asynchrone** :

1. Un processus P demande un transfert (par exemple une lecture sur disque) => appelle une primitive du SE (read)
2. Cette primitive programme le contrôleur de périphérique pour l'opération demandée et le DMA pour le transfert
3. Elle met le processus en état bloqué et ajuste sa priorité (bonus)
4. L'ordonnanceur cherche un autre processus prêt et l'exécute
5. L'ordonnancement des processus se poursuit comme d'habitude
6. L'IT de fin de transfert arrive
7. Le processus en cours P' est suspendu et l'IT est traitée
8. Ce traitement d'IT consiste en particulier à rendre le processus P prêt
9. Le processus suspendu P' reprend
10. Plus tard l'ordonnanceur trouvera le processus P prêt, il le rendra actif et P pourra se poursuivre

Remarque : vu du processus le transfert est **bloquant** mais vu du CPU il ne l'est pas.

# Comment le SE gère les E/S ?

Ça met en jeu 4 niveaux :

1. Le gestionnaire d'interruptions
  - Aspect matériel : prise en compte de l'IT par n° et table d'adresses
2. Les pilotes de périphériques
  - Gestion du périphérique et de ses ITs : inclus ou ajoutés au SE
3. La partie non tributaire du périphérique
  - Vue du périphérique par le SE : par exemple une clé USB est vue comme un disque mais un téléphone peut être vu soit comme un disque soit comme une machine (débogage)
4. Les fonctions offertes aux programmeurs
  - Fonctions d'utilisation du périphérique

# Le gestionnaire d'ITs

La **prise en compte d'une IT** est relativement complexe car il faut :

1. **Sauvegarder les registres du CPU** dans la pile (programme interrompu)  
*ATTENTION* : On ne peut pas sauvegarder les registres dans la pile du processus en cours car on pourrait tomber sur un défaut de page  $\Rightarrow$  une nouvelle IT pour charger la page !!!  
 $\Rightarrow$  On utilise la pile du SE qui est dans une page fixe toujours en mémoire (exclue de l'allocation de mémoire).
2. **Enregistrer le contexte du processus actif** dans la table des processus
3. **Trouver l'adresse de la procédure d'IT (table + n° d'IT)**
4. **Charger les tables de mémoire virtuelle** pour la procédure d'IT
5. **Lancer la procédure d'IT**
6. **Acquitter l'IT** vis-à-vis de son émetteur
7. **Traiter l'IT** : par le code contenu dans le pilote du périphérique

**Conclusion** : les SE classiques ne sont pas faits pour traiter des ITs très rapidement  $\Rightarrow$  **non temps réel**

# Les pilotes de périphériques

- Chaque périphérique a un programme associé : **le pilote** (driver) éventuellement livré avec le périphérique et adapté au SE
- Un pilote peut contrôler **plusieurs périphériques** (plusieurs disques par exemple)
- Le pilote fait partie du **noyau du SE**, il contient en particulier la **procédure d'IT**
- Les SE définissent une interface standard (propre au SE) c'est à dire un **ensemble de fonctions** que le pilote doit fournir au SE pour piloter les périphériques (stratégie).
- Le pilote est **chargé dynamiquement** par le SE quand il démarre

Il est indispensable que le pilote corresponde  
au périphérique **ET** au système d'exploitation

# Les pilotes de périphériques

Ce que fait le pilote :

- vérifier que les **paramètres** de la requête soient valides
- si le périphérique est occupé, la requête est **mise en attente** sinon elle est exécutée
- **exécuter la requête** en accédant aux registres du contrôleur de périphérique (souvent il faut plusieurs étapes) + programmer le DMA si nécessaire.
- quand la requête est lancée, le pilote **se bloque** en attendant que la requête soit terminée.
- quand l'IT de fin le **réveille** il transmet **les états** au logiciel de la couche supérieure (non tributaire du périphérique) et regarde s'il y a d'**autres requêtes en attente** si oui il lance la suivante sinon il se termine

Remarque : les pilotes doivent être **ré-entrants** c'est à dire qu'un pilote doit pouvoir traiter une IT alors qu'il faisait autre chose.

Par exemple quand on retire une clé USB alors que le pilote était en train de lire ou écrire dessus  $\Rightarrow$  stopper la requête en cours et enlever celles en attente.

**Ecrire un driver n'est pas simple !**

# La couche indépendante du périphérique

Son rôle :

- **interfaçage uniforme** pour tous les pilotes
- mise en **mémoire tampon**
- rapport d'**erreurs**
- **allocation et libération** des périphériques
- fournir une **taille d'information** indépendante du matériel.

# La couche indépendante du périphérique

## Interfaçage uniforme :

- Les **fonctions offertes** par le pilote diffèrent d'un pilote à l'autre
- Les **fonctions du noyau** dont le pilote a besoin diffèrent d'un pilote à l'autre
- Cette couche se charge d'**interfacer le pilote avec le noyau et avec les utilisateurs.**

Exemple : les noms de périphériques

sous UNIX                /dev/dsk0

sous windows          C:

Sous UNIX à un nom de périphérique sont associées 2 valeurs :

**major number** : qui sert à localiser le pilote

**minor number** : qui sert à désigner l'unité pour ce pilote.

Exemple de lignes du répertoire UNIX /dev :

brw-r-----	1	root	disk	8,	1	août 26 2006	sda1
brw-r-----	1	root	disk	8,	2	août 26 2006	sda2
crw-rw----	1	root	tty	4,	10	août 26 2006	tty10
crw-rw----	1	root	tty	4,	11	août 26 2006	tty11



# La couche indépendante du périphérique

## Mise en mémoire tampon (buffering)

Pour éviter que le processus ne soit réveillé à chaque information lue sur le périphérique on peut utiliser un **tampon (buffer)**

⇒ on ne le réveille que quand le **tampon est plein**

⇒ on gère un second tampon pour mettre les infos qui arrivent quand le 1<sup>er</sup> est plein et avant que le processus ne l'ait vidé.

Le DMA sait gérer 2 tampons et basculer de l'un à l'autre

L'utilisation de tampons permet au SE de mettre en place des **stratégies** adaptées aux périphs.

# La couche indépendante du périphérique

## Exemple de stratégies pour les disques :

- **Lecture anticipée sur disque :**

1. Un processus demande la lecture d'un enregistrement (secteur) sur disque
2. Le contrôleur de périphérique déplace la tête et attend que l'enregistrement passe
3. Pendant cette attente plutôt que de ne rien faire on peut lire les enregistrements qui passent et les mettre dans un tampon  $\Rightarrow$  si plus tard on nous demande un de ces enregistrements on l'aura déjà  $\Rightarrow$  réponse immédiate au processus demandeur.

- **Stratégie d'écriture sur disque :**

Parmi les tampons en attente d'écriture sur le disque on peut choisir d'écrire celui qui est le plus accessible c'est à dire qui demande le moins de déplacement de la tête.

- **Stratégie d'ascenseur :**

Pour optimiser les déplacements des têtes entre les lectures et les écritures on peut les traiter dans l'ordre des pistes croissant puis décroissant  $\Rightarrow$  la tête se déplace vers le centre en traitant toutes les lectures et écritures en attente puis repart vers le bord en traitant toutes les lectures et écritures en attente etc.

- **RAID** (Redundant Array of Inexpensive Disks) :

On répartit les données sur plusieurs disques qui sont vus par les utilisateurs comme un seul.

- RAID1 : n disques pour répartir les données + n disques en copie en cas de panne
- RAID 4 et 5 : un disque est utilisé pour le contrôle d'erreurs des autres  $\Rightarrow$  en cas de panne d'un disque on peut reconstituer les informations perdues donc reconstituer le contenu du disque planté (même de celui de contrôle).
- RAID 6 : amélioration de RAID 5 qui accepte 2 pannes simultanées

# La couche indépendante du périphérique

## Mise en mémoire tampon (buffering)

- **L'avantage des mémoires tampons** c'est que quand un processus veut écrire des données elles sont copiées dans un tampon et le **processus peut continuer** (il n'est pas bloqué au-delà de ce temps de copie)
  - de son point de vue les données sont écrites
  - en réalité elles ne le sont pas (pas encore)
- **L'inconvénient des mémoires tampons** c'est qu'en cas de problème certaines données **ne sont pas réellement écrites**.

Exemple : enlever une clé USB sauvagement  $\Rightarrow$  risque que certains fichiers soient incorrects alors que normalement ils étaient OK (du point de vue de l'utilisateur).

C'est pour ça qu'il y a des **procédures particulières** pour :

- **déconnecter** un périphérique amovible
- **fermer** une session
- **éteindre** la machine.

# La couche indépendante du périphérique

## Rapports d'erreurs

Lors d'une **erreur** on peut faire différentes choses :

- tenter de **refaire l'opération** si c'est possible (par ex pour une écriture ou une lecture disque OK mais pour une gravure sur DVD !)
- **planter le processus** demandeur
- **renvoyer un message** d'erreur au processus (s'il peut faire quelque chose)

## Allocation et libération des périphs

Certains périphériques (graveur par ex) ne peuvent pas être partagés  $\Rightarrow$  utiliser des **sémaphores** pour mettre en attente les processus qui veulent y accéder jusqu'à ce qu'ils se libèrent

## Fournir une taille de données indépendante du périphérique

Par exemple la taille réelle des blocs (secteurs) des supports de stockage est cachée et on fournit une taille de bloc (**cluster**) identique pour tous (disques, clé USB).

# Les fonctions offertes aux utilisateurs

Le SE offre des **fonctions d'utilisation** des périphériques.

Par exemple en C sous UNIX :

**printf** , **scanf** , **open** , **write** , **read** ....

Certaines de fonctions fonctionnent pour plusieurs types de périphériques (par exemple read sur un fichier, une socket, un pipe, ...)

Quand un périphérique est non partageable le SE peut proposer un **spooleur** (spool = Simultaneous Peripheral Operation On Line) c'est un processus toujours présent (**démon**) qui gère l'accès à ce périphérique (par exemple : imprimante).

# Un périphérique particulier : Timer

**Dispositif matériel** = compteur physique décrémente à intervalles réguliers

- Programmable en **mode répétitif**  $\Rightarrow$  une IT chaque fois qu'il arrive à 0 et le compteur est automatiquement rechargé à la valeur contenue dans un registre d'initialisation du Timer.
- Programmable en **mode mono coup**  $\Rightarrow$  une IT quand le compteur arrive à 0 puis il s'arrête jusqu'à ce qu'on le relance en réinitialisant le compteur.
- Utilisé en mode répétitif (chaque 20 à 30 ms) pour le **temps partagé**
- Utilisé en mode mono coup pour des surveillances (**watchdog timer**). Par exemple pour arrêter les disques au bout d'un certain temps de non utilisation.

Remarque : le SE gère aussi des **timers logiciels** càd des variables qu'il décrémente à intervalles réguliers (à chaque IT d'un timer répétitif physique).

C'est moins précis mais ça permet aux utilisateurs de se définir autant de timers logiciels qu'ils veulent

Exemples : UNIX propose la fonction **alarm** qui permet à un processus de demander à recevoir un signal **SIGALRM** dans un délai donné.

Et aussi les fonctions **sleep** et **usleep** qui mettent un processus en sommeil pour un délai donné.

# Les Systèmes d'Exploitation

1. Introduction
2. Les Processus
3. Les Threads
4. Concurrency entre processus
5. Communication entre processus
6. Les interruptions logicielles (signaux)
7. Gestion du CPU et de la mémoire
8. Ordonnancement
9. Les Entrées/Sorties
- 10. Le système de fichiers**
11. Démarrage d'un système
12. Sécurité
13. Interblocage
14. Systèmes multiprocesseurs

# *Organisation du système de fichiers*

- La structure des fichiers est vue comme un **arbre** (répertoires, sous-répertoires, fichiers) mais ce n'est pas la réalité (liens).
- Secteur 0 du disque = **MRB** (Master Boot Record) contient la **table des partitions** dont une est marquée **active**.

Chaque partition contient :

- Un **secteur de boot**
- Les informations relatives au **système de fichiers**
- Les informations sur les **espaces libres**
- Les informations sur les **espaces occupés** par les fichiers
- Et les **données**.



# Implantation des fichiers

**Généralement un fichier n'occupe pas une zone consécutive sur disque mais un ensemble ordonné de blocs disjoints. Il faut savoir les retrouver :**

- **Liste chaînée**

Pour chaque fichier on a une **liste chaînée des blocs** qu'il occupe. Mais c'est vite très lent.

- **FAT** (File Allocation table)

Pour chaque fichier on a une **table désignant les blocs** qu'il utilise. Mais la table devient vite très grande.

- **Inodes**

C'est un mélange des 2 solutions :

On fait **une table** de type FAT pour chaque fichier mais de taille limitée et si la table ne suffit pas on utilise certaines entrées pour désigner **d'autres tables**.

# Exemple d'UNIX

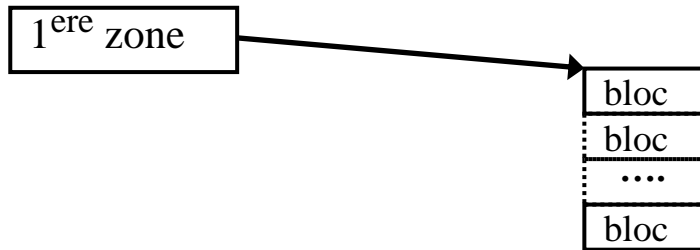
A chaque fichier est associée une **entrée dans la table des inodes**. Dans cette entrée on trouve :

- type de fichier
- bits de protection
- nombre de liens sur ce fichier
- propriétaire UID et GID
- taille du fichier
- adresses des 10 1<sup>ers</sup> blocs occupés par le fichier
- dates et heures du dernier accès au fichier et de la dernière modification du fichier et de l'inode
- 3 zones d'indirection.

# Exemple d'UNIX

Les 3 **zones d'indirection** marchent comme suit :

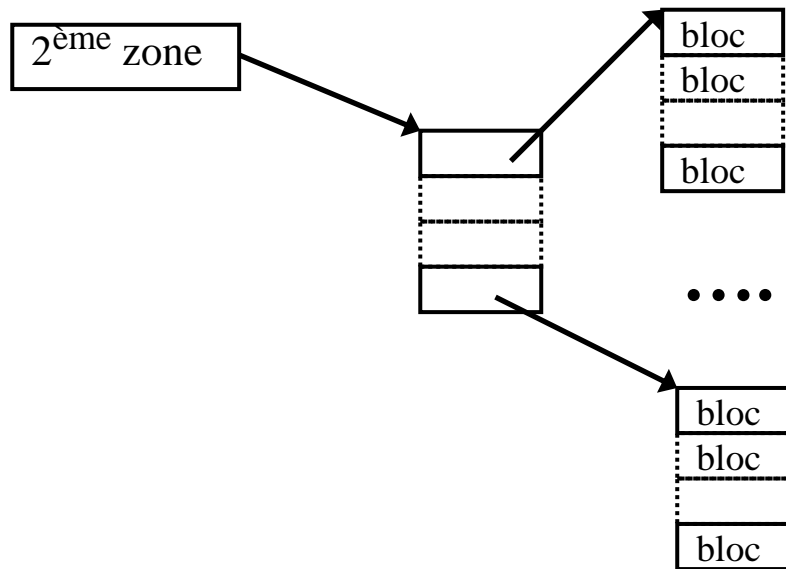
- 1<sup>ère</sup> zone pointe sur une table contenant les 256 blocs suivants du fichier



# Exemple d'UNIX

Les 3 **zones d'indirection** marchent comme suit :

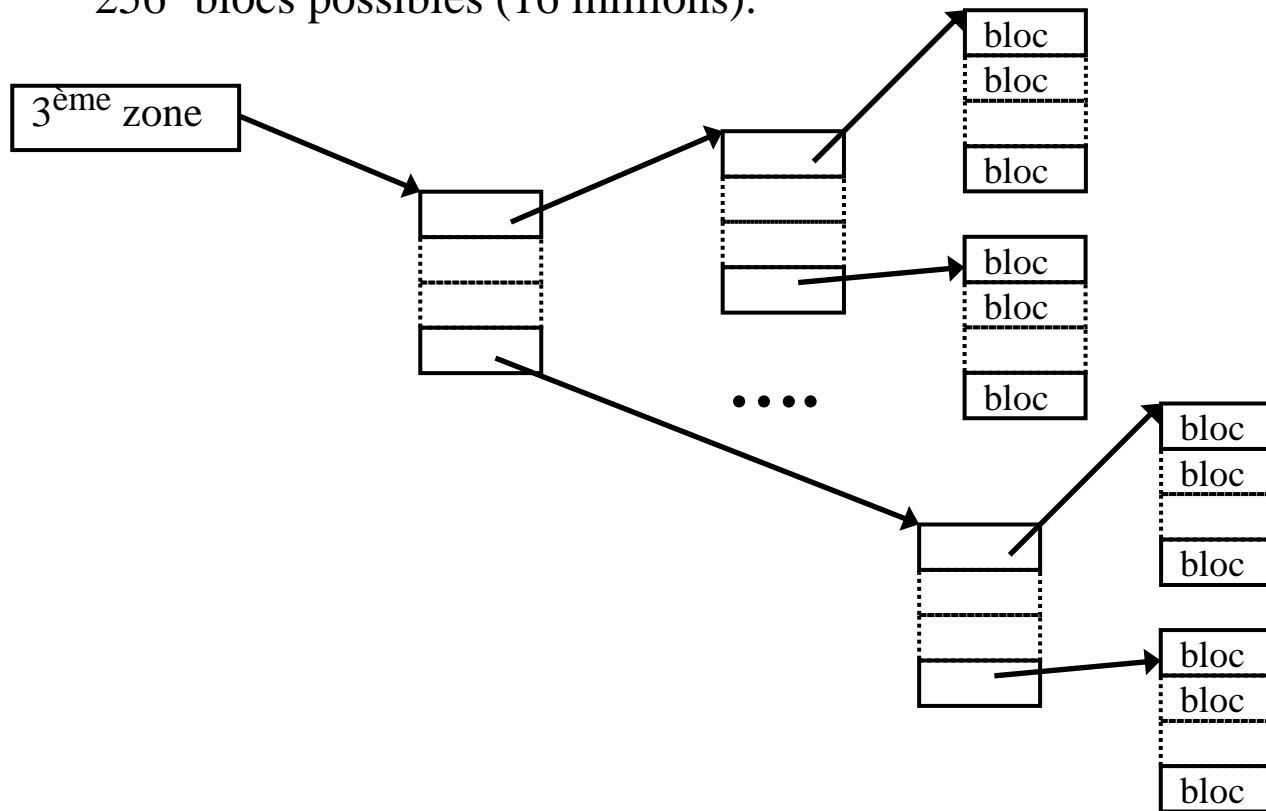
- 2<sup>ème</sup> zone utilisée si la 1<sup>ère</sup> ne suffit pas, elle désigne une table dont chaque entrée désigne une table de blocs suivants. On a donc au max 256 tables désignant 256 blocs chacune  $\Rightarrow 256^2$  blocs possibles (65000).



# Exemple d'UNIX

Les 3 **zones d'indirection** marchent comme suit :

- 3<sup>ème</sup> zone utilisée si ça ne suffit encore pas. Elle désigne une table dont chaque entrée désigne une table du même type que celle désignée par la 2<sup>ème</sup> zone  $\Rightarrow$   $256^3$  blocs possibles (16 millions).



# Exemple de Windows NTFS

Il existe une table appelée **MFT** (Master File Table) dont chaque élément fait 1Ko et décrit 1 fichier par son nom, ses attributs et la liste des blocs qu'il utilise

Cette liste est constituée d'éléments du type (**bloc début – bloc fin**) c'est à dire que si le fichier est en 3 morceaux sur le disque on n'a que 3 éléments dans la liste quelle que soit la taille du fichier

Si cette liste devient trop grande certaines entrées de la liste de blocs sont utilisées pour **désigner d'autres éléments de la MFT** servant à contenir la suite de cette liste

Si le fichier est très petit son contenu peut être **directement** mis dans la MFT ( $\Rightarrow$  gain de place et de temps).

# Gestion des blocs libres

Le SE doit savoir quels sont les **blocs libres** dans le système de fichiers. On peut utiliser :

- une **liste chaînée** des blocs libres (**UNIX**)
- une **table de bits** (1 bit par bloc = libre/occupé) (**WINDOWS**)

Cas des CD/DVD-ROM : c'est plus simple parce que tous les fichiers sont contigus.

# Les Systèmes d'Exploitation

1. Introduction
2. Les Processus
3. Les Threads
4. Concurrency entre processus
5. Communication entre processus
6. Les interruptions logicielles (signaux)
7. Gestion du CPU et de la mémoire
8. Ordonnancement
9. Les Entrées/Sorties
10. Le système de fichiers
- 11. Démarrage d'un système**
12. Sécurité
13. Interblocage
14. Systèmes multiprocesseurs



# Démarrage d'un système

Le démarrage se fait en plusieurs étapes :

- À l'allumage, le processeur lance l'exécution d'un programme qui est en mémoire Flash (**BIOS** = Basic Input-Output System).
- Ce programme reconnaît et initialise les contrôleurs de périphériques présents dans la machine et nécessaires au démarrage.
- Il "*boote*" ensuite sur l'un des périphériques présents grâce à son secteur 0 : **MRB** (Master Boot Record) qui contient la table des partitions dont une est marquée active et contient un secteur de boot.
- Ceci amène le chargement et l'exécution du **noyau** d'un système d'exploitation (Unix, Windows,...)

# Démarrage d'un système (exemple d'UNIX)

- Le noyau reconnaît les périphériques et démarre les pilotes correspondants (les *drivers*)
- il **monte** l'arborescence racine en utilisant le système de fichiers situé sur une des partitions du disque
- puis le noyau lance le programme **/sbin/init** qui devient le processus numéro 1 (celui qui adopte les zombies)
- Le rôle du noyau s'arrête là en ce qui concerne le démarrage du système. C'est **init** qui se charge de démarrer les services.

# Démarrage d'un système (exemple de LINUX)

Le processus **sbin/init** démarre les services (démons) en se basant sur :

- Le contenu d'un fichier de configuration **/etc/inittab**
- Un "niveau de démarrage" qui lui a éventuellement été indiqué (0 = halt, 1 = mono utilisateur, 2 = multi-utilisateur sans NFS, 3 = multi-utilisateur avec NFS, 5 = multi-utilisateur + environnement graphique, 6 = redémarrage)
  - Ce démarrage, lance le script **/etc/init.d/rcS**
  - Puis les scripts contenus dans **/etc/rc.d/rcN.d** où N est le "niveau de démarrage". Les scripts qui commencent par **S** pour le démarrage, ceux qui commencent par **K** pour l'arrêt

# Les Systèmes d'Exploitation

1. Introduction
2. Les Processus
3. Les Threads
4. Concurrency entre processus
5. Communication entre processus
6. Les interruptions logicielles (signaux)
7. Gestion du CPU et de la mémoire
8. Ordonnancement
9. Les Entrées/Sorties
10. Le système de fichiers
11. Démarrage d'un système
- 12. Sécurité**
13. Interblocage
14. Systèmes multiprocesseurs

# Authentification de l'utilisateur

En général par un **mot de passe**. On peut aussi utiliser des systèmes physiques comme la biométrie (empreintes digitales) ou des cartes à puces.

## Fragilité des mots de passe :

De nombreux logiciels permettent de **cracker** les mots de passe.

Les principes qu'ils utilisent sont les suivants :

- Utiliser des **dictionnaires** standards
- Utiliser le contenu du fichier de mots de passe pour générer des mots de passe potentiels, comme le login
- Utiliser le nom de l'utilisateur et toutes ses données associées
- Utiliser un langage de description de modifications permettant de définir les transformations des mots de passe potentiels
- Construire des statistiques à propos de la constitution des mots de passe déjà cassés.

# Fragilité des mots de passe

Temps de craquage par force brute  
(40M essais/seconde)

	Longueur du mot de passe	6	7	8	10
Type de mot de passe	Lettres seules (maj et min)	8 mn	7 h	15 j	114 ans
	Lettres (maj et min) et chiffres	24 mn	1 j	63 j	665 ans
	Plus caractères spéciaux (+28)	3 h 41 mn	19 j	2,5 ans	27600 ans

*Remarque :* Les mots les plus probables sont testés en premier  $\Rightarrow$  temps beaucoup plus court

*Remarque :* Ces temps peuvent être divisés en utilisant des machines en réseau ou des circuits spécialisés (GPU, DSP) ou un superordinateur (plusieurs millions de fois plus rapide 27600 ans  $\rightarrow$  quelques heures).

# Authentification de l'utilisateur

## Sécurité des mots de passe sous Windows

- Les mots de passe sont placés dans le fichier **SAM** (Security Accounts Manager)
- Il sont cryptés par deux algorithmes différents (LANMAN et NTLM)
- Le 1<sup>er</sup> crypte par DES le mot de passe mis en majuscules et complété par des 0 pour atteindre 14 caractères puis séparé en 2 moitiés
- Le 2<sup>ème</sup> crypte le mot de passe exprimé en UNICODE (le résultat est sur 16 octets).

## Sécurité des mots de passe sous LINUX

- A la création on ajoute au mot de passe un code sur 12 bits (*salt*) aléatoire
- Le mdp concaténé au *salt* sont cryptés
- Ces codes et les *salts* sont placés dans le fichier **/etc/shadow** qui n'est accessible que par root

# Sécurité du système de fichiers

Le contrôle d'accès est **discrétionnaire** en ce sens que le propriétaire d'un objet (fichier, répertoire ...) peut en modifier les permissions d'accès.

Principe de **droits** :

- **UNIX** utilisait rwx (read/write/execute) avec les notions d'utilisateur / groupe / autres
- **MSDOS** et **Windows 3.1** utilisaient (read/write/caché) sans notion d'utilisateur ni de groupe
- **Windows NT** définit des droits pour chaque utilisateur et/ou chaque groupe. Mais un utilisateur peut apparaître en tant que tel puis en tant qu'appartenant à plusieurs groupes. Dans ce cas Windows **combine les droits** => on a les droits de chacun des groupes + ceux de l'utilisateur

Exception : si dans l'un des groupes l'utilisateur n'a **aucun droit** c'est ce qui prend le dessus.



# Principes de sécurité par ACL (Windows et Linux)

La liste de contrôle d'accès **ACL** (Access Control List) spécifie pour **chaque fichier** qui, quels utilisateurs et groupes ont accès à ce fichier.

Chaque entrée d'une ACL assigne à un utilisateur ou à un groupe un ou plusieurs des niveaux d'accès suivants aux fichiers :

- **Aucun** n'accorde aucun accès au fichier.
- **Lire** autorise l'affichage des données du fichier
- **Écrire** autorise la modification des données du fichier
- **Exécuter** autorise l'exécution du fichier programme.
- **Supprimer** autorise la suppression du fichier.
- **Modifier les autorisations** autorise le changement des autorisations sur le fichier.
- **Appropriation** autorise l'appropriation du fichier.

# Principes de sécurité par ACL

La liste de contrôle d'accès **ACL** (Access Control List) spécifie pour **chaque répertoire** qui, quels utilisateurs et groupes ont accès à ce répertoire.

Un jeu de privilèges similaire est défini sur les répertoires :

- **Aucun** n'accorde aucun accès au répertoire.
- **Lire** autorise l'affichage des noms de fichiers et de sous-répertoires
- **Écrire** autorise l'ajout de fichiers et de sous-répertoires
- **Exécuter** autorise la modification des sous-répertoires
- **Supprimer** autorise la suppression de sous-répertoires.
- **Modifier les autorisations** autorise le changement des autorisations du répertoire
- **Appropriation** autorise l'appropriation du répertoire.

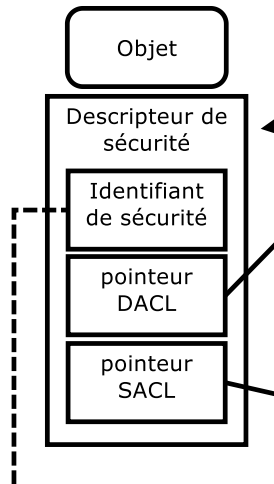
# Modèle de sécurité de Windows

Sous Windows, toutes les ressources sont gérées sous forme d'**objets** (les fichiers, les processus, les clés de la base d'enregistrement ... ).

Chaque objet du système est protégé par un **descripteur de sécurité (SD)** (Security Descriptor), qui définit quels types d'accès sont autorisés de la part de quelles entités.

Chaque processus dispose d'un contexte de sécurité lui permettant d'accéder ou non à des objets. Cette structure, attachée à tout processus, porte le nom de **jeton d'accès (AC)** (Access Token).

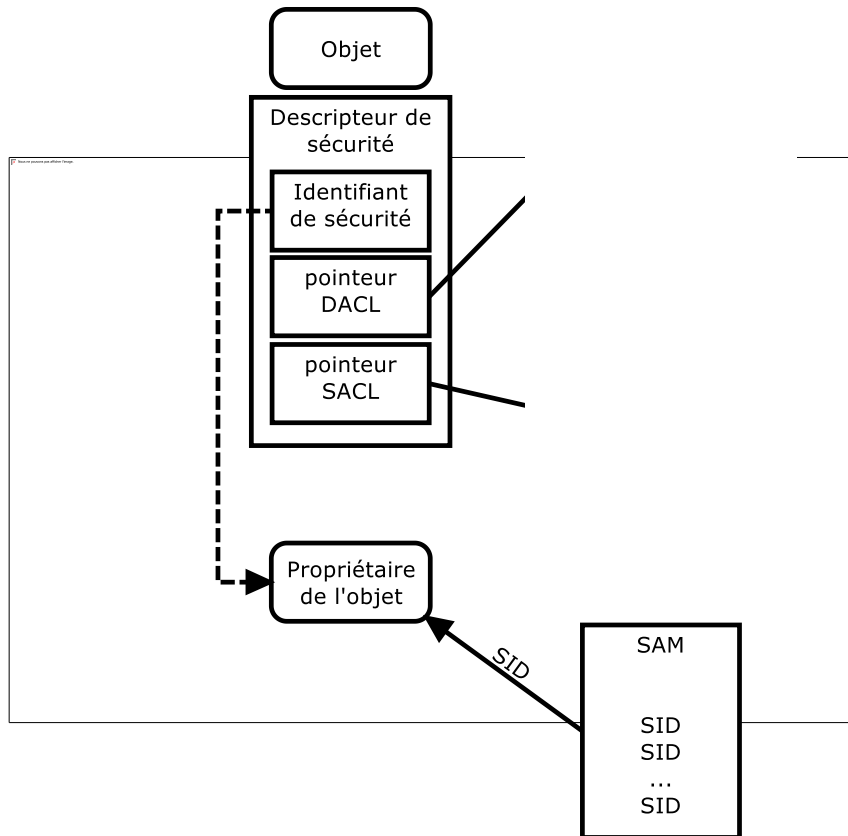
# Modèle de sécurité de Windows



A tout objet est associé un **descripteur de sécurité** (SD), qui contient trois informations principales :

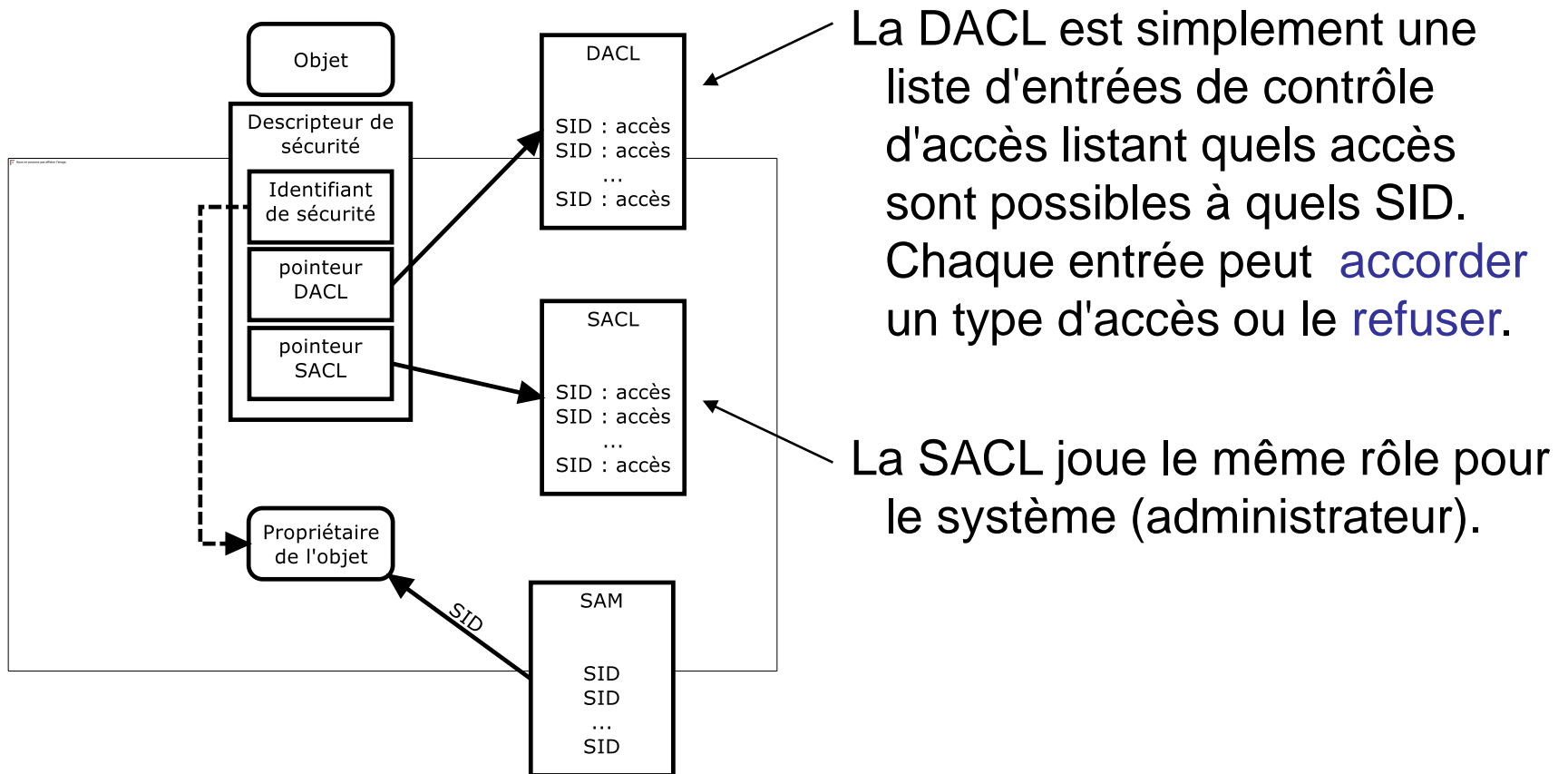
- l'identifiant de sécurité (SID) du propriétaire de l'objet,
- sa liste de contrôle d'accès discrétionnaire (DACL)
- sa liste de contrôle d'accès système (administrateur) (SACL)

# Modèle de sécurité de Windows

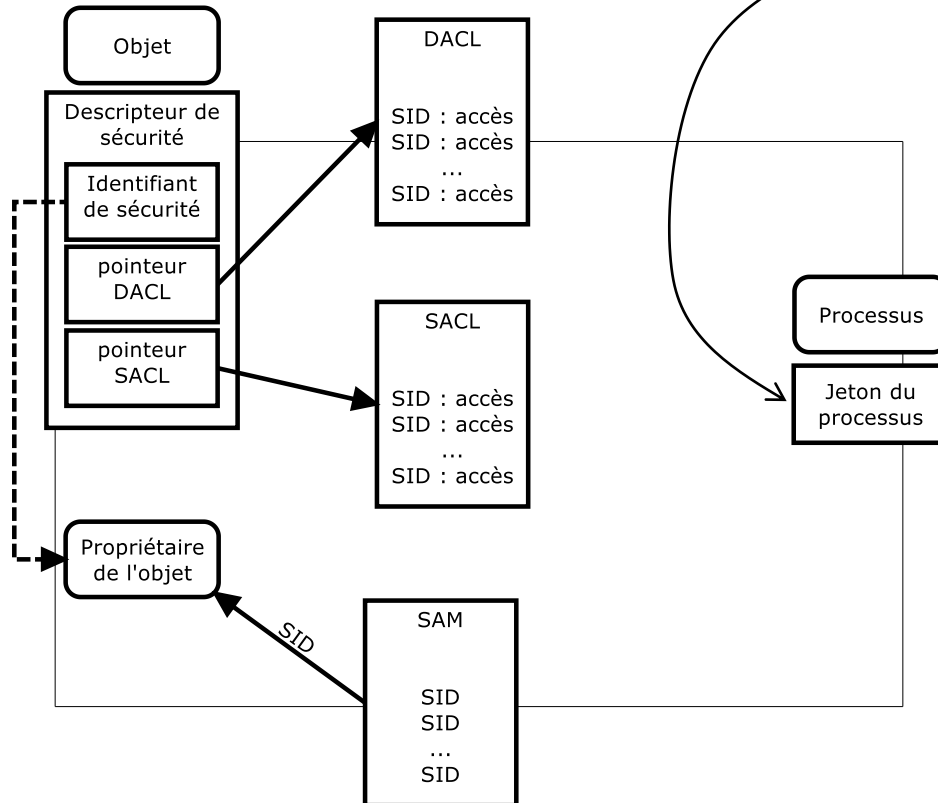


Un propriétaire est identifié par son SID placé dans la base de données de sécurité du système **SAM** (Security Account Manager) qui fait partie de la base d'enregistrement (base de registre : `HKEY_LOCAL_MACHINE` )

# Modèle de sécurité de Windows



# Modèle de sécurité de Windows



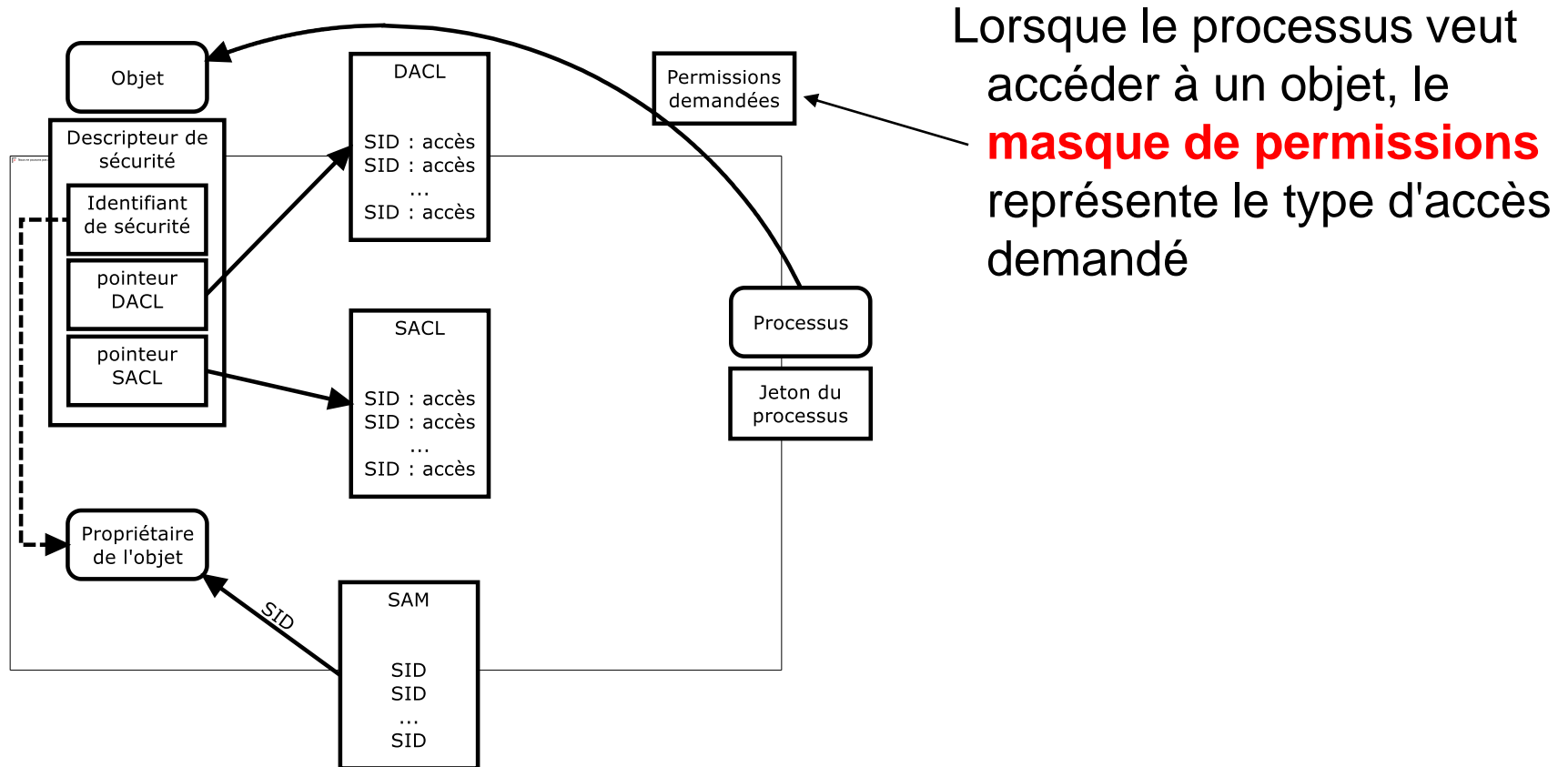
Un **jeton d'accès (AC)** est attaché à chaque processus et définit son contexte de sécurité. En particulier par :

–identité et attributs d'autorisation :

- SID du propriétaire du processus
- Liste des SID des groupes locaux (au niveau du système local) dont le propriétaire fait partie.
- Liste des SID des groupes globaux (au niveau du domaine)
- SID indiquant le type de session de connexion (par exemple, interactive)
- SID identifiant la session de connexion.

–sécurité par défaut pour les nouveaux objets créés

# Modèle de sécurité de Windows





# Modèle de sécurité de Windows

L'algorithme utilisé pour le contrôle d'accès utilise trois éléments :

- le **masque de permissions** représentant le type d'accès demandé
- le **descripteur de sécurité** représentant la protection de l'objet
- le **jeton du processus**, contenant les attributs d'autorisation

L'algorithme utilisé est le suivant :

- Si les permissions implicites sont suffisantes, **l'accès est autorisé**
- Sinon chaque entrée de la DACL est examinée. Si elle autorise certaines permissions d'accès pour l'un des SID contenu dans le jeton, ces permissions sont ajoutées.
- L'algorithme se poursuit jusqu'à ce que l'un des trois événements suivants se produise :
  - Une entrée de la DACL refusant un accès à un SID contenu dans le jeton est rencontrée : **l'accès est refusé**
  - La fin de la DACL est atteinte sans que toutes les permissions aient été accumulées : **l'accès est refusé.**
  - La fin de la DACL est atteinte et toutes les permissions présentes dans le masque de permissions spécifié lors de l'accès ont été accumulées : **l'accès est autorisé**

# Virus

- Définition :
  - « Programme capable “d’infecter” d’autres programmes en les modifiant de façon à inclure une copie de lui-même, potentiellement évoluée » *Frederick B. Cohen 1972*
- Généralement les virus sont **cachés** dans un logiciel parfaitement fonctionnel
- Actuellement, l’objectif est de **gagner de l’argent** :
  - Vol d’informations bancaires
  - Appel de n°s de téléphone
  - Fishing : provoquer une action volontaire de l’utilisateur
  - Bots (robots) pour le déni de service
  - Rançon (virus police, virus hadopi, cryptage, ...)

# Les types de virus

On distingue les types de virus suivants :

- Les vers (worms) : se copient eux-mêmes et de se propagent à travers un réseau
- Les chevaux de Troie (trojans) : installent d'autres applications afin de pouvoir contrôler l'ordinateur qu'ils infectent depuis l'extérieur.
- Les bombes logiques : se déclenchent suite à un événement particulier (date système, activation distante, ...)
- Les Rootkits : dissimulent des objets sur les ordinateurs (processus, fichiers ...). Ils sont souvent utilisés par d'autres virus pour se cacher
- Les Exploits : exploitent une faille de sécurité dans un logiciel, un système d'exploitation ou un protocole de communication
- Les Adwares : affichent des publicités
- Les Spywares : collectent et transfèrent des informations

# Les formes de virus

On distingue cinq formes de virus :

- **Les virus mutants** : virus réécrits afin d'en modifier le comportement ou la signature (presque tous)
- **Les virus polymorphes** : virus dotés de fonctions de chiffrement et de déchiffrement de leur signature
- **Les rétrovirus** : virus ayant la capacité de modifier les signatures des antivirus afin de les rendre inopérants
- **Les virus de secteur d'amorçage** : virus capables d'infecter le secteur de démarrage d'un disque (MBR master boot record)
- **Les virus trans-applicatifs (virus macros)** : virus situés à l'intérieur d'un banal document et exécutant une portion de code à l'ouverture de celui-ci.

# Les antivirus

Un **antivirus** est un programme capable de détecter la présence de virus et, dans la mesure du possible, de désinfecter le programme porteur. On parle alors d'**éradication** de virus pour désigner la procédure de nettoyage de l'ordinateur.

Il existe plusieurs méthodes d'éradication :

- La **suppression du code** correspondant au virus dans le fichier infecté (pas toujours possible)
- La **suppression du fichier** infecté
- La **mise en quarantaine** du fichier infecté : le déplacer dans un emplacement où il ne pourra pas être exécuté (permet de le restaurer si on est sûr qu'il n'est pas dangereux, fausse alerte).

# Les antivirus

## (détection de virus connus)

- L'antivirus s'appuie sur la **signature** (portion de code exécutable ajoutée par le virus au fichier infecté) propre à chaque virus. Il s'agit de la méthode de **recherche de signature** (*scanning*), la plus ancienne méthode utilisée par les antivirus.

Cette méthode n'est fiable que si l'antivirus possède une **base virale** à jour, c'est-à-dire comportant les signatures de tous les virus connus et de toutes leurs variantes (mutants).

- Cependant les **virus polymorphes**, dotés de capacités de camouflage, peuvent échapper à cette méthode : le code du virus est crypté et s'auto-décrypte au démarrage.

On utilise alors les **méthodes génériques** qui recherchent la partie du code du virus qui n'est pas modifiée (celle qui décrypte le reste) mais ce n'est pas toujours efficace.

# Les antivirus

## (détection de virus non connus)

- Utilisation d'un **contrôleur d'intégrité** pour vérifier si les fichiers ont été modifiés. Le contrôleur d'intégrité construit une base de données contenant des informations sur les fichiers exécutables du système (date de modification, taille et éventuellement un code de contrôle type MD5). Ainsi, lorsqu'un fichier exécutable change de caractéristiques, l'antivirus prévient l'utilisateur de la machine. **Gênant pour les programmeurs**
- **Surveillance du comportement** des processus. Pas exemple un processus qui modifie la base d'enregistrement ou accède à des fichiers système. Mais cela **peut lever de fausses alertes**
- **Méthodes heuristiques** qui consistent à analyser le début du code d'un processus pour détecter soit une auto modification du code soit une tentative de trouver d'autres exécutables (pour les infecter) . Mais cela **peut lever de fausses alertes**
- Méthode du **bac à sable** consiste à exécuter le programme douteux dans un émulateur du système d'exploitation puis à analyser ce qu'il a modifié. L'émulateur évite les risques de modification réelle du système mais c'est **lent**.

# Tester la présence d'un antivirus

Test mis au point par le comité d'experts EICAR (European Institute for Computer Antivirus Research : <http://www.eicar.org/>) spécialistes de la sécurité informatique

- Pour tester qu'un antivirus est actif, créer un fichier texte.
- Tapez-y la ligne suivante :  
**X5O!P%@AP[4\PZX54(P^)7CC)7}\$EICAR-STANDARD-ANTIVIRUS-TEST-FILE!\$H+H\***
- Lors de l'enregistrement du fichier l'antivirus devrait réagir en signalant la présence du virus ***Eicar-Test-Signature*** ou ***Eicar-AV-Test***



# Les autres mécanismes

Relèvent plutôt du réseau :

- Pare-feu (firewall)
  - Filtrage de protocoles
  - Filtrage d'adresses IP
  - Filtrage de ports
  - Filtrage d'applications
  
- Mandataire (proxy)
  - Filtrage de serveurs
  - Filtrage d'URLs

# Les Systèmes d'Exploitation

1. Introduction
2. Les Processus
3. Les Threads
4. Concurrency entre processus
5. Communication entre processus
6. Les interruptions logicielles (signaux)
7. Gestion du CPU et de la mémoire
8. Ordonnancement
9. Les Entrées/Sorties
10. Le système de fichiers
11. Démarrage d'un système
12. Sécurité
- 13. Interblocage**
14. Systèmes multiprocesseurs

# Interblocage

L'**accès concurrent** aux ressources peut provoquer des **interblocages** de processus.

Exemple : Les processus A et B utilisent les ressources R1 et R2

1. A demande l'accès à R1 : **accordé**
2. B demande l'accès à R2 : **accordé**
3. A demande l'accès à R2 : **refusé** car utilisé par B  $\Rightarrow$  **A se bloque**
4. B demande l'accès à R1 : **refusé** car utilisé par A  $\Rightarrow$  **B se bloque**

**Les 2 processus sont bloqués indéfiniment ainsi que les 2 ressources**

Ce phénomène peut se produire dans des quantités de circonstances comme l'accès concurrent à des enregistrements d'une BD qui sont verrouillés par les processus.

# Les ressources

Ce sont des **périphériques** ou des **variables partagées** ou tout ce qui peut faire l'objet d'**accès concurrents**

Certaines ressources peuvent être **retirées** sans problème à un processus qui les possède comme la mémoire par exemple.

D'autres **ne le peuvent pas** comme un graveur de DVD

En général on utilise des **sémaphores** pour contrôler l'accès aux ressources partagées.

# Le problème de l'interblocage

Condition d'**interblocage** il y en a 4 :

1. **Condition d'exclusion mutuelle** : chaque ressource est soit libre soit attribuée à un processus et un seul.
2. **Condition de détention** : les processus ayant déjà obtenu des ressources peuvent en demander d'autres
3. **Condition de non réquisition** : les ressources détenues par un processus ne peuvent pas lui être retirées. Elles ne peuvent être libérées que par ce processus
4. **Condition d'attente circulaire** : il y a un cycle d'au moins 2 processus chacun attendant une ressource détenue par l'autre

Ces 4 conditions **sont nécessaires** pour que se produise un interblocage.

**Mais ça arrive !**

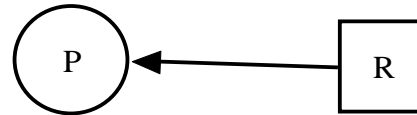
# Modélisation de l'interblocage

On dessine :

- un **processus** par un cercle
- une **ressource** par un carré

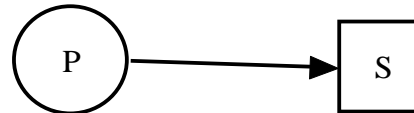
Un arc de ressource vers processus indique que cette ressource **est détenue** par ce processus

**P détient R**



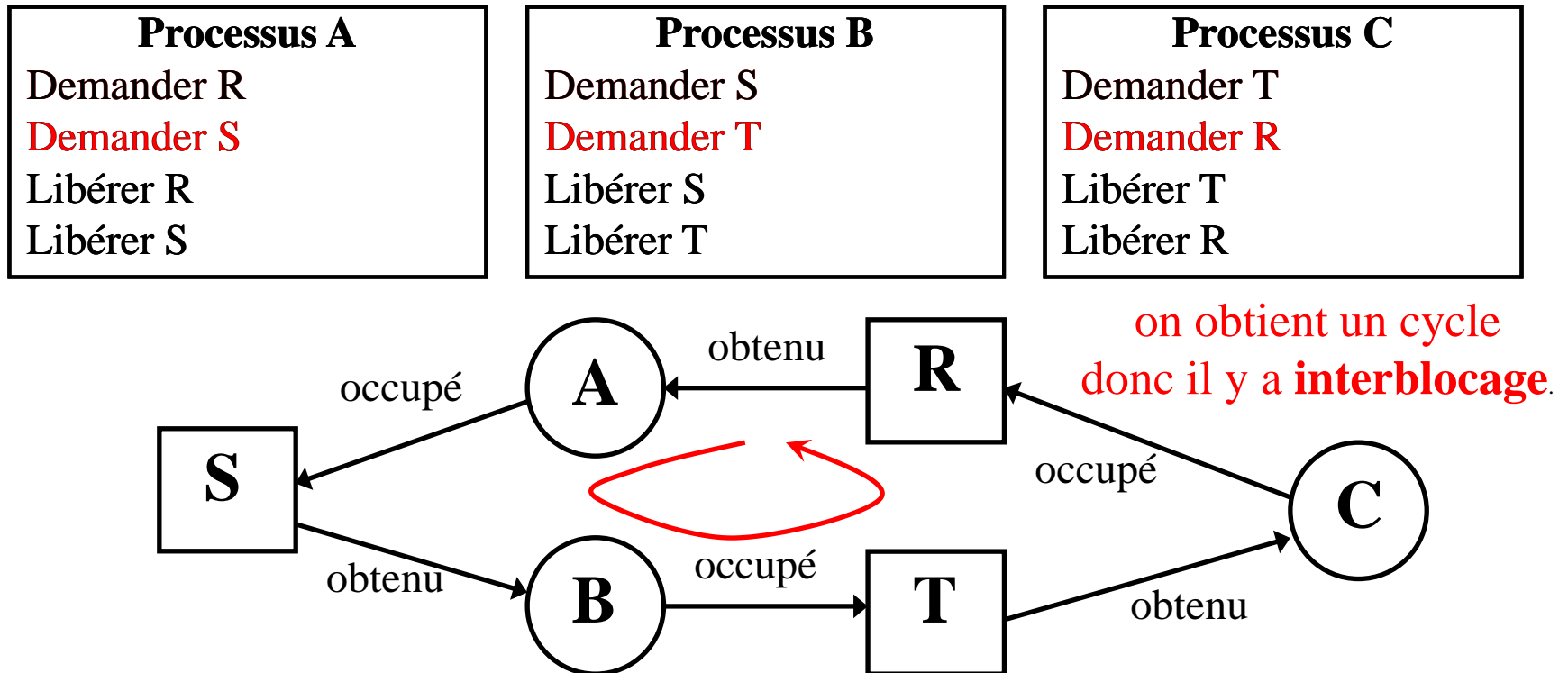
Un arc de processus vers ressource indique que ce processus **demande** cette ressource

**P demande S**



En faisant ce graphe pour tous les processus et toutes les ressources on met en évidence des cycles => des **interblocages**.

# Exemple d'évolution de 3 processus et 3 ressources



Remarque : si les 3 processus s'étaient **exécutés différemment** sans rien changer à leur code l'interblocage aurait pu être évité : par exemple si A avait libéré R avant que C ne le demande.

# Comment traiter le problème de l'interblocage ?

- **Ignorer**

Ignorer le problème en supposant qu'il ne se produit pas souvent : l'utilisateur pourra y remédier (tuer un processus par exemple). **UNIX et Windows font comme ça !**

- **Détecter et reprendre**

- **Détecter** : le problème revient à **trouver un cycle** dans le graphe processus – ressources. On connaît des algorithmes pour ça.
- **Reprendre** : on a 3 façons de résoudre le problème de l'interblocage :
  - **Préemption** : on **retire une ressource** à un processus en général c'est très **difficile voire impossible sans intervention humaine**
  - **rollback** : si les processus prévoient des **points de reprise** comme on le fait en réseau il est possible de ramener les processus à des points de cohérence antérieurs à la demande de ressource qui a fermé le cycle **mais il faut que le programmeur ait prévu ça (difficile)**
  - **suppression de processus** : on choisit un processus du cycle et on le **supprime**. Il est parfois possible de relancer ce processus un peu plus tard sans qu'il y ait de conséquences. Par exemple un processus qui envoie des mails peut être tué puis relancé sans problème (**mais lequel choisir ?**).



# Comment traiter le problème de l'interblocage ?

- **Eviter**

On a vu qu'il faut réunir **4 conditions** pour avoir un interblocage donc si on peut éviter que l'une d'entre elles se produise on a évité le problème.

- **condition d'exclusion mutuelle** : on peut éviter cette condition pour certains périphériques par exemple le **spooleur** d'impression est le seul qui a accès à l'imprimante
- **condition de détention** : on peut exiger que tout processus demande **toutes les ressources dont il aura besoin dès le début**. Ou il les obtient et il se terminera et les libèrera ou il ne les obtient pas et attend sans bloquer de ressource. Mais les processus ne peuvent pas toujours savoir d'avance de quoi ils auront besoin (surtout les processus interactifs). De plus avec ce système les ressources sont très sous employées.
- **condition de non préemption** : c'est en général impossible
- **condition d'attente circulaire** : on peut exiger qu'un processus ne puisse demander **qu'une ressource à la fois**  $\Rightarrow$  il faut qu'il libère celle qu'il a avant d'en demander une autre. En fait on peut alléger cette règle en numérotant les ressources et en imposant la règle : Un processus ne peut demander une ressource de n° k que s'il ne détient aucune ressource de n°  $> k$ . Ceci fait que **l'ordre de blocage des ressources** est toujours le même et que l'interblocage est impossible.

# Les Systèmes d'Exploitation

1. Introduction
2. Les Processus
3. Les Threads
4. Concurrency entre processus
5. Communication entre processus
6. Les interruptions logicielles (signaux)
7. Gestion du CPU et de la mémoire
8. Ordonnancement
9. Les Entrées/Sorties
10. Le système de fichiers
11. Démarrage d'un système
12. Sécurité
13. Interblocage
- 14. Systèmes multiprocesseurs**
15. Les systèmes temps réel

# Hyperthreading / Multi processeur

L'**hyperthreading** consiste à dupliquer H fois (en général 2 fois) les registres du processeur  $\Rightarrow$  quand on passe d'un processus à un autre on n'a plus à sauvegarder/restituer le contexte machine mais seulement à changer de registres (plus rapide).

Le système d'exploitation gère alors le processeur comme H processeurs **mais l'exécution n'est pas réellement parallèle (seulement plus rapide)**.

**Multi-cœur** : placer K cœurs dans la même puce. Ils partagent la mémoire et les caches de niveau 3 mais ont chacun leur CPU et leurs caches de niveau 1 et 2.

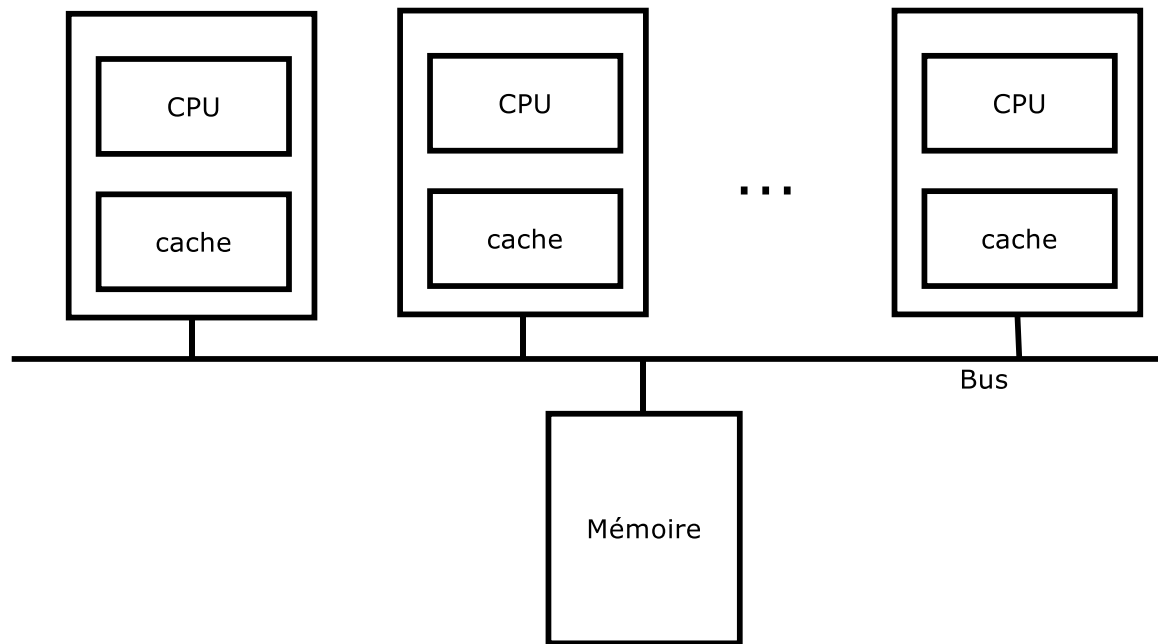
Le système d'exploitation gère alors les K cœurs **et l'exécution est réellement parallèle**

**Multiprocesseur** : placer N processeurs dans la même machine (chacun pouvant être multi-cœur et chaque cœur pouvant faire de l'hyperthreading)

Si on a N processeurs chacun ayant K cœurs implémentant l'hyperthreading (H duplications des registres). Le système gère  $N * K * H$  processeurs et l'exécution est  $N * K$  fois réellement parallèle.

# Architectures à mémoire commune

- Principe



Les processeurs partagent la même mémoire  
Chacun a ses caches propres

# Exemple d'architecture à mémoire commune

## Serveur Power Edge R940 (DELL) :

- 4 CPU Intel Xeon Platinum 8180 à 2,5 GHz ayant 28 cœurs chacun
- Mémoire : 3 To de RAM à 2666 MHz
- Stockage : 92 To (24 disques SSD de 3,84 To à 12 Go/s en RAID 5)
- Réseau : 4 cartes Ethernet 10 Gb/s
- SE : Windows server ou linux (Red Hat ou Ubuntu)
- Alimentation : 2 x 2400 W (redondante)
- Prix : 400 000 €
- Puissance maximale : 8960 G opérations/s

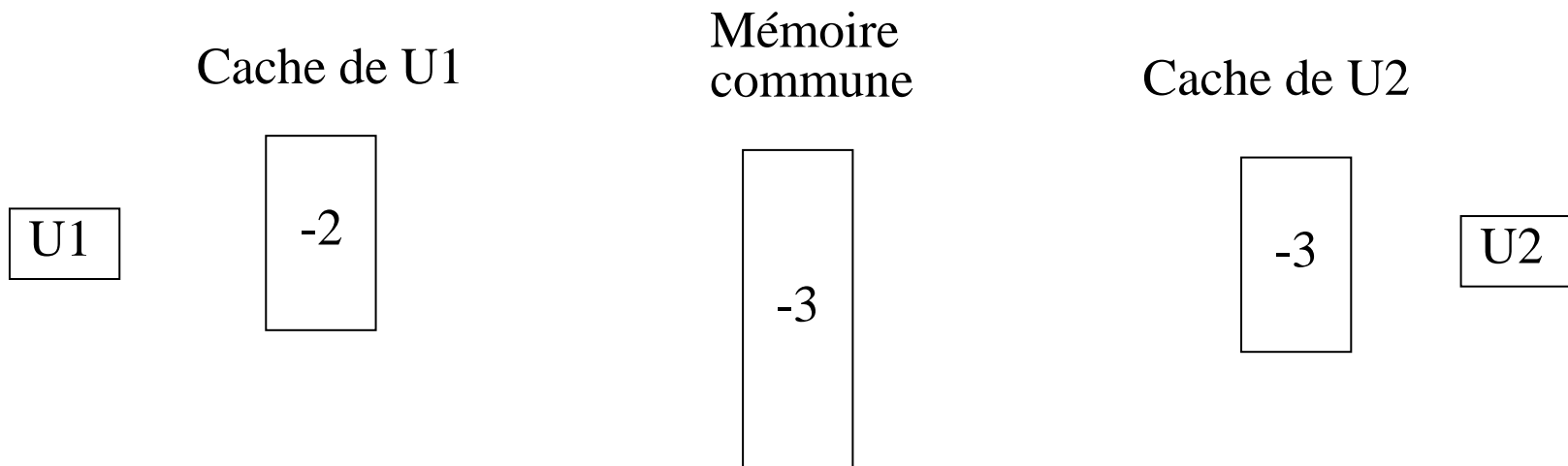


# Problèmes liés à la mémoire commune et aux caches

- Un processus (ou thread) P1 s'exécute sur le processeur U1
- Un second processus (ou thread) P2 s'exécute sur le processeur U2
- Ces 2 processus partagent une variable V qui est dans la mémoire commune de la machine
  - P1 modifie V par exemple : si ( $V < 0$ )  $V \leftarrow V + 1$
  - P2 utilise V par exemple :  $A \leftarrow V / 2$
- U1 et U2 ont des caches de niveau 1 et 2 propres
- Au moment où P1 lit V pour savoir s'il est  $< 0$  cette variable est mise dans le cache de U1 si elle n'y était pas déjà. Donc quand P1 modifie V (exécute  $V \leftarrow V + 1$ ) V est dans le cache de U1
- Au moment où P2 utilise V cette variable est mise dans le cache de U2 si elle n'y était pas déjà
- Plusieurs cas peuvent se produire :

# Problèmes liés à la mémoire commune et aux caches

- 1<sup>er</sup> cas : V est déjà dans le cache de U2

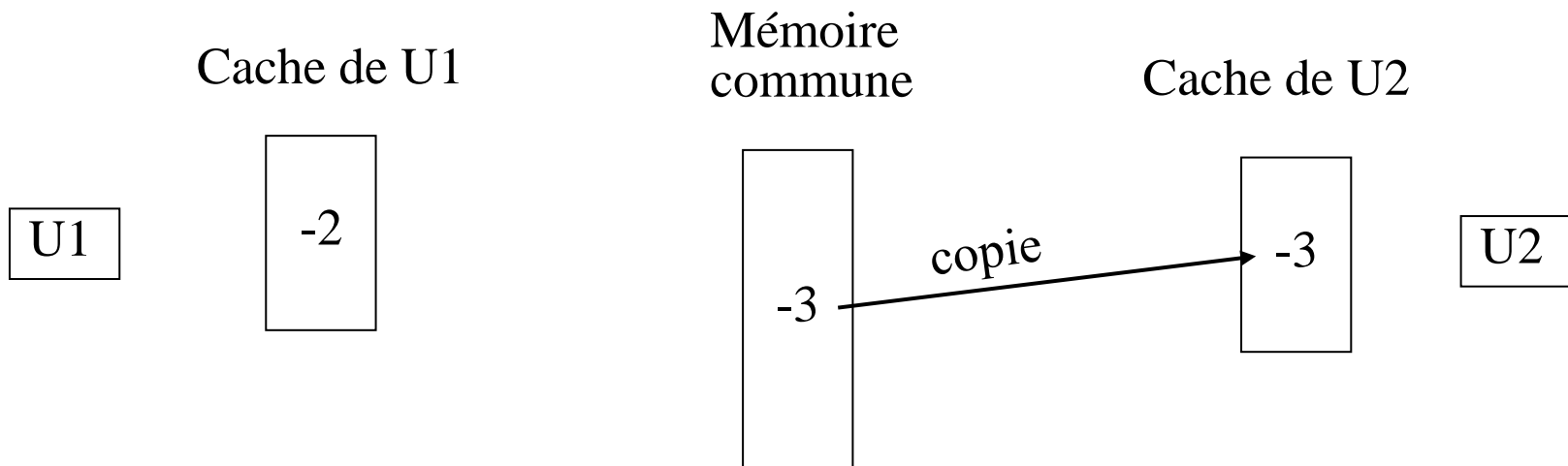


- V étant  $< 0$  P1 modifie V
- Quand P2 exécute  $A \leftarrow V/2$  il obtient  $A = -1,5$

C'est FAUX

# Problèmes liés à la mémoire commune et aux caches

- 2<sup>ème</sup> cas : V n'est pas dans le cache de U2



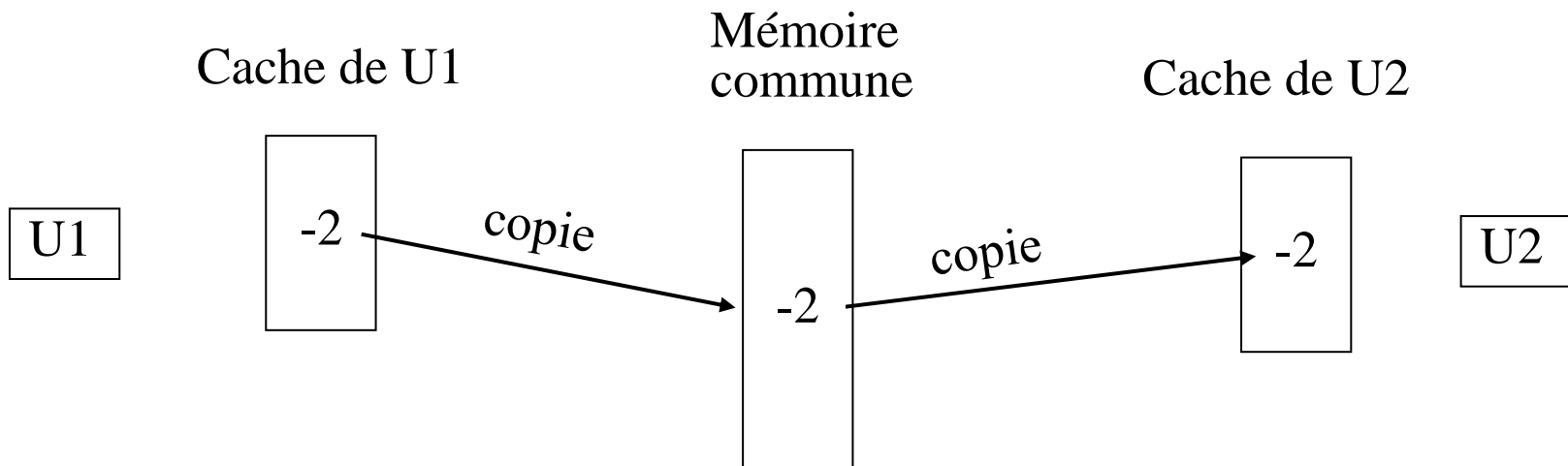
- V étant  $< 0$  P1 modifie V
- Quand P2 veut exécuter  $A \leftarrow V/2$  une copie est faite dans son cache depuis la mémoire
- Le calcul de P2 donne  $A = -1,5$

C'est **FAUX**



# Solution possible

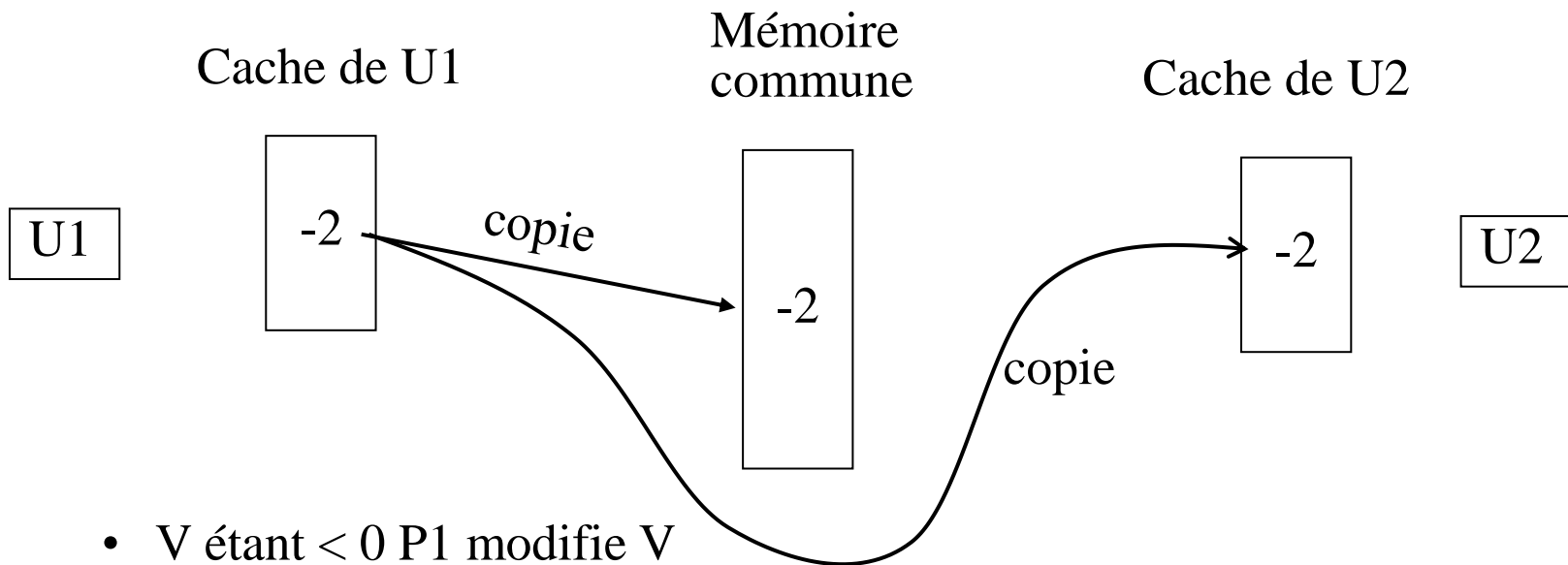
- Synchronisation par la mémoire



- $V \text{ étant } < 0$  P1 modifie  $V$
- Comme  $V$  est partagé la copie en mémoire du cache de U1 est faite
- Le bloc contenant  $V$  dans le cache de U2 est invalidé
- Quand P2 veut exécuter  $A \leftarrow V/2$  une copie est faite dans son cache depuis la mémoire
- **Le calcul de P2 donne bien  $A = -1$**

# Solution optimisée

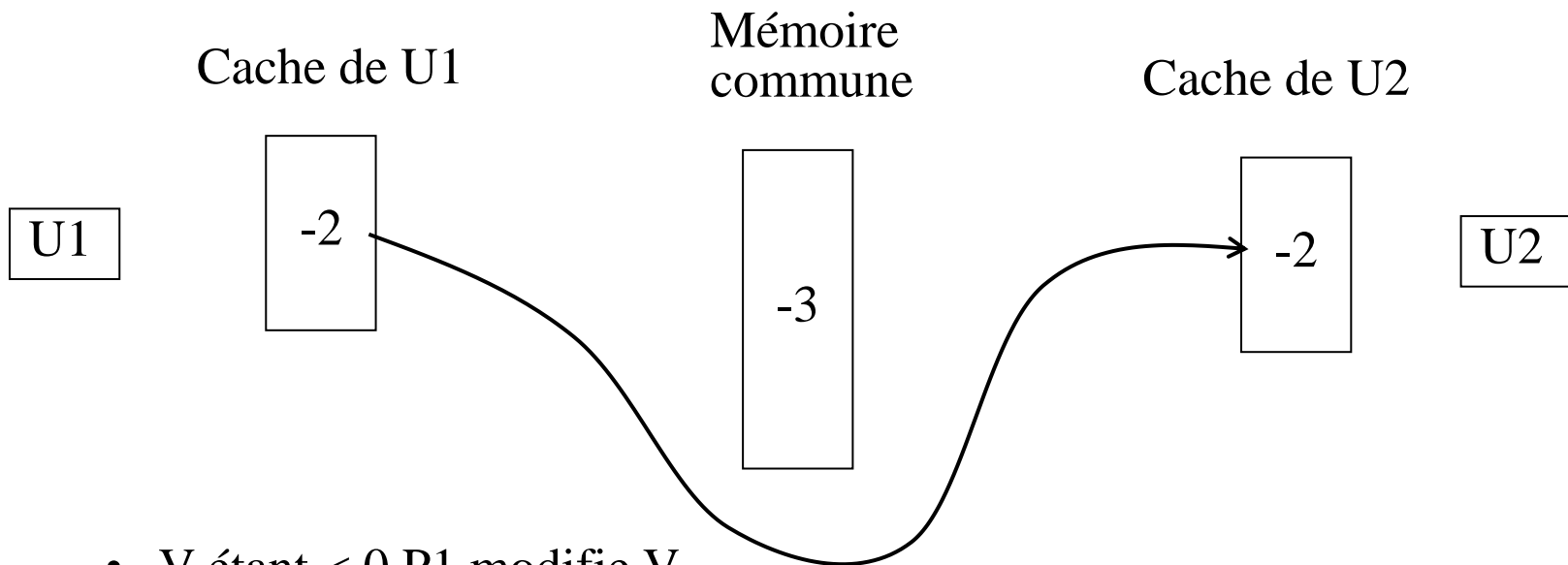
- Synchronisation par la mémoire (version optimisée)



- V étant  $< 0$  P1 modifie V
- Comme V est partagé la copie en mémoire du cache de U1 est faite et le bloc contenant V dans le cache de U2 est mis à jour
- Le calcul de P2 donne bien  $A = -1$

# Autre solution possible

- Synchronisation des caches



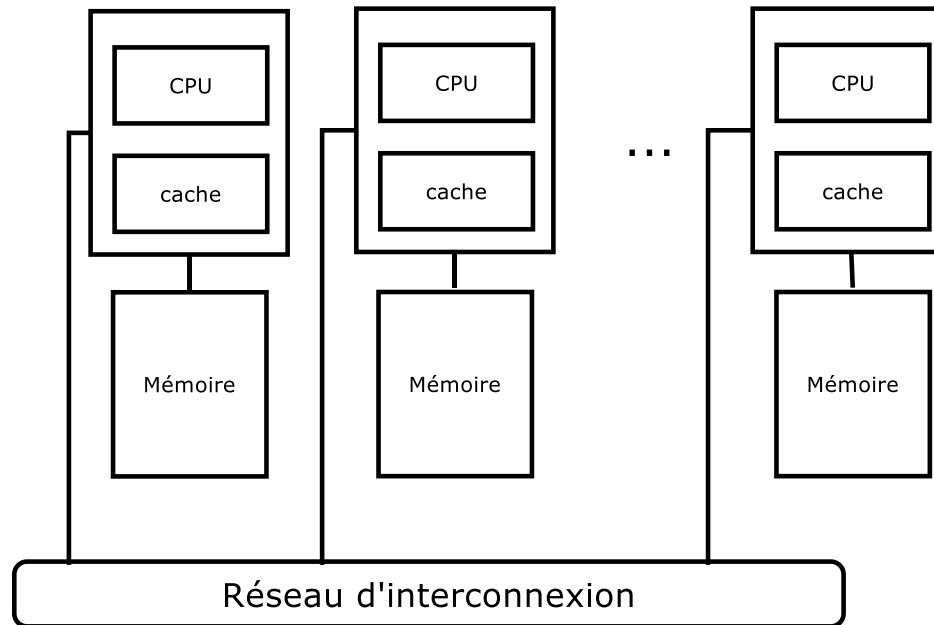
- $V \text{ étant } < 0$  P1 modifie  $V$
- Comme  $V$  est partagé la modification du cache de U1 est reportée sur celui de U2 (pas sur la mémoire)
- Le calcul de P2 donne bien  $A = -1$

# Comment le faire ?

- Un **gestionnaire des caches** qui sait ce que contient chaque cache :
  - Lors d'un transfert mémoire → cache il vérifie si la mémoire doit être mise à jour avant.
  - Lors d'utilisation simultanée de variables il synchronise les caches (copie d'un vers N)
- Optimisation :
  - Lors des transferts de données entre cache et mémoire on peut récupérer ces données sur le bus pour effectuer le transfert vers les caches concernées ⇒ mise à jour simultanée de plusieurs caches ⇒ gain de temps.

# Architectures à mémoires séparées (clusters)

- Principe



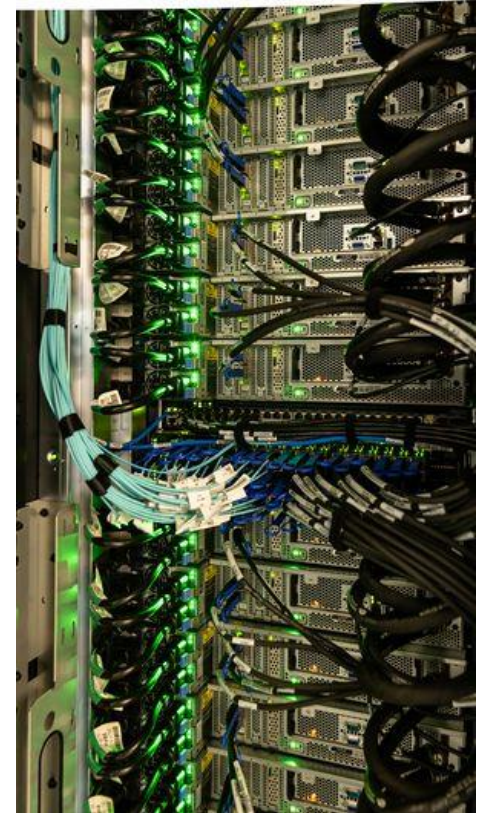
Les processeur ont chacun leur mémoire

Ils sont interconnectés par un réseau rapide (échange de données)

# Exemple de cluster

## IBM Summit (Oak Ridge National Laboratory)

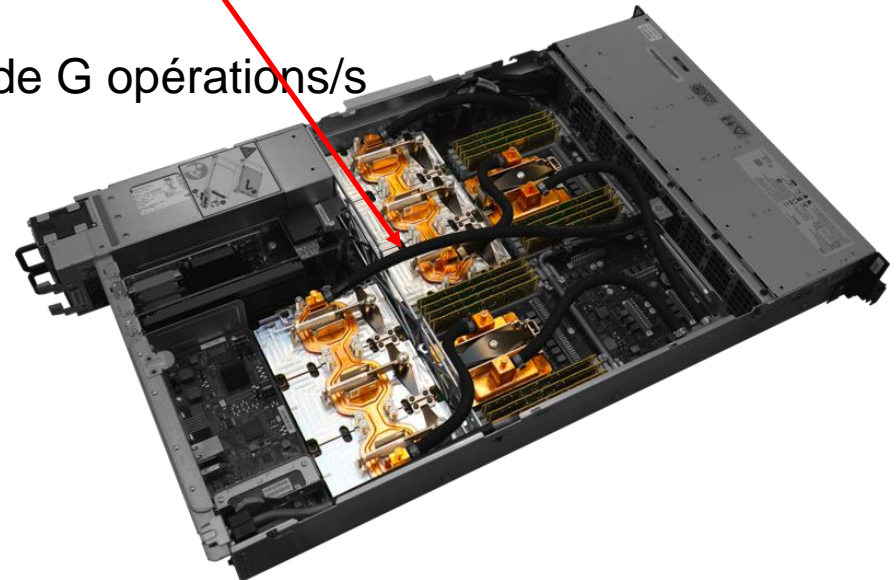
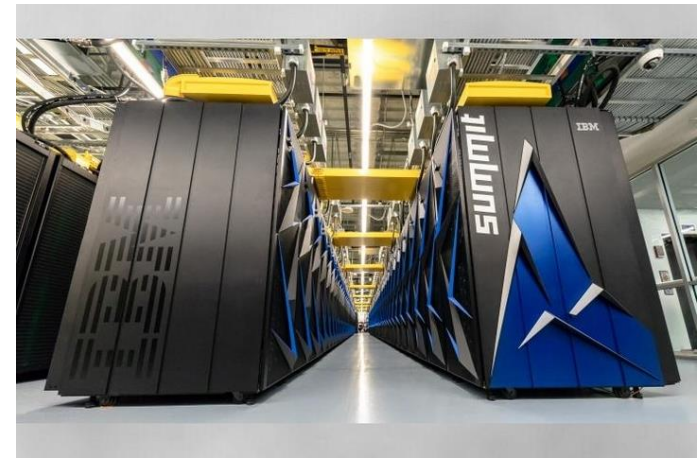
- 4608 nœuds regroupés en 256 racks de 18 nœuds
- Chaque nœud comprend :
  - 2 processeurs POWER 9 à 3 GHz ayant 22 cœurs chacun (202 752 cœurs de CPU au total)
  - 6 processeur de calcul NVIDIA Volta V100s à 80 cœurs chacun (2 211 840 cœurs de calcul au total)
  - 512 Go de RAM + 96 Go de mémoire pour les procs de calcul + 1600 Go de mémoire flash (2360 To RAM + 442 To mémoire de calcul + 7373 To flash au total)



# Exemple de cluster

IBM **Summit** (Oak Ridge National Laboratory)

- Interconnexion : réseau en arbre à 100 Gb/s
- Stockage : 250 millions de Go à 2200 Go/s + 8 serveurs de disques
- Consommation / Poids / Surface : 13 MW, 340 tonnes, 520 m<sup>2</sup>
- Refroidissement par eau : 15 000 litres / minute
- SE : linux Red Hat
- Coût : 240 M d'euros
- Puissance maximale : 200 millions de G opérations/s



# Problèmes liés aux mémoires séparées

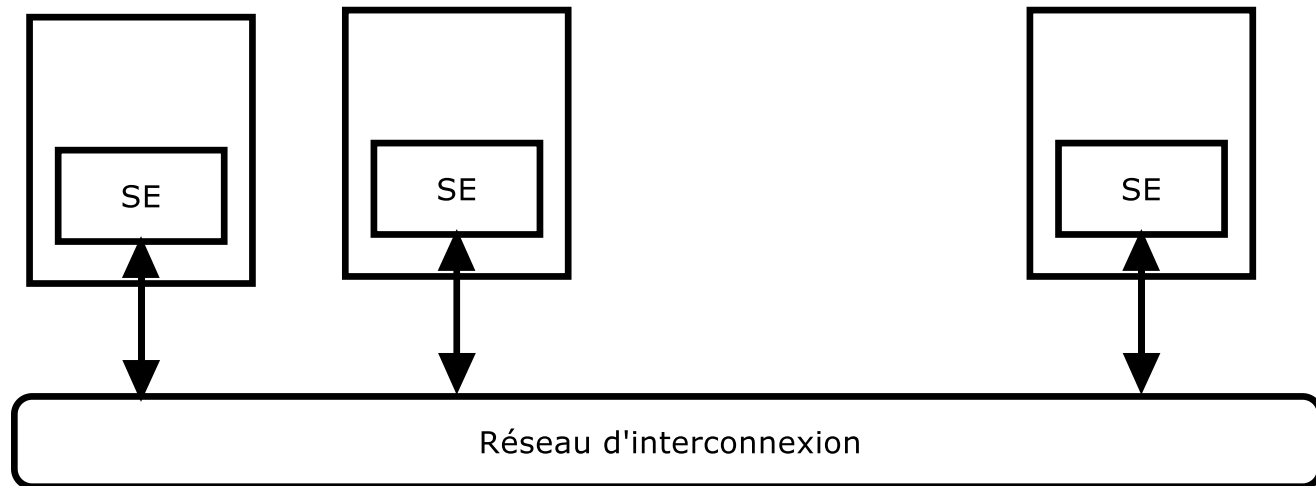
- Distribution initiale des données
  - Certaines données doivent être dupliquées dans les mémoires des processeurs (comme le fait fork)
- Transferts de données par réseau
  - Les processeurs échangent des données par le réseau d'interconnexion (comme on le fait par un pipe)
- Synchronisation des données
  - Il faut s'assurer que les données modifiées en un point et utilisées en un autre soient à jour.
- Communication entre processus
  - Par le réseau d'interconnexion
    - Communication directe ou indirecte (lien entre P1 et P2 direct ou via P3)
    - Synchrone ou asynchrone (attente ou pas d'acquittement)
  - Possibilité d'appel de fonctions à distance (PRC)



# SE des Systèmes multiprocesseurs

Il existe 3 grands types de solutions :

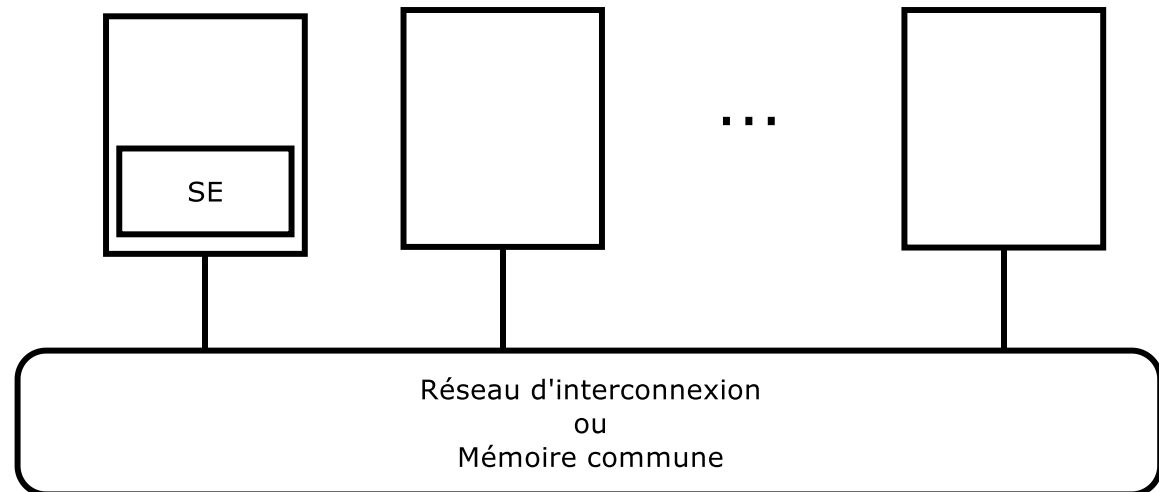
1. **Chaque processeur a son SE** mais dans ce cas il n'y a pas vraiment de coopération possible entre processus sauf à faire comme sur un réseau classique



# SE des Systèmes multiprocesseurs

Il existe 3 grands types de solutions :

- 2. Maître/Esclave** : 1 processeur a le SE, les autres exécutent les processus utilisateurs. A chaque nouveau processus créé le CPU maître choisit le CPU sur lequel il va l'implanter en essayant d'équilibrer la charge.

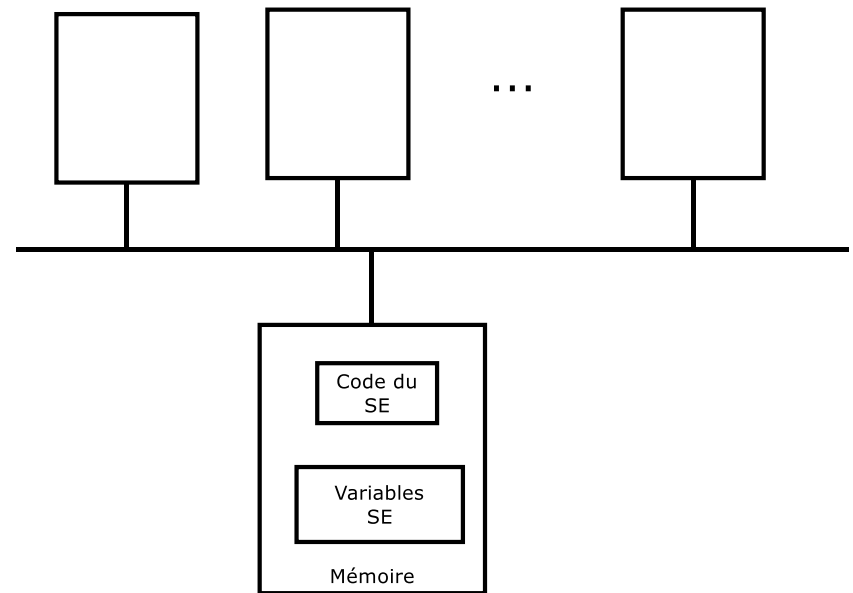


# SE des Systèmes multiprocesseurs

Il existe 3 grands types de solutions :

- 3. Symétrique** : Il n'y a qu'un SE en mémoire mais tous les CPU peuvent l'exécuter (cette mémoire est partagée). On protège alors le code du SE par un **MUTEX** pour qu'un seul CPU l'exécute à la fois

Pour optimiser on met **des MUTEX différents selon les parties du SE** (par ex un CPU peut exécuter l'ordonnanceur pendant qu'un autre exécute une E/S)  $\Rightarrow$  **le SE est divisé en sections critiques** mais il faut éviter les interblocages.



# Ordonnancement des processus

Le système d'exploitation distribue **les processus** sur les différents processeurs :

- Equilibrer les charges des processeurs

Le problème est celui de l'ordonnancement des processus :

Quand un processus démarre ou redémarre sur quel processeur faut-il le mettre ?

*A priori sur le moins chargé mais ...*

# Ordonnancement

- **Cas d'un processus non lié (càd n'ayant aucun lien avec d'autres)**

Choisir le processeur le moins chargé a un **inconvénient** qui est que si un processus A a tourné sur le processeur  $P_1$  puis est repris par le processeur  $P_2$  il y a perte de temps du fait que  $P_1$  avait peut être encore ses pages en mémoire alors que  $P_2$  n'a rien.

On peut éviter ça en choisissant qu'un processus soit **toujours exécuté sur le même CPU** choisi à la création du processus mais la charge n'est plus aussi bien répartie.

# Ordonnancement

- **Cas des processus liés**

Ce sont des **processus qui échangent beaucoup entre eux** (père-fils ou threads d'un même processus). Pour que les échanges se fassent de façon efficace il est préférable que :

- ces processus tournent sur des **CPU différents**
- ces processus tournent en **même temps**

Explication :

- S'ils tournent sur le même CPU il n'y a pas de vrai parallélisme
- S'ils tournent sur des CPU différents mais pas au même moment les échanges sont différés

Solution : créer des **groupes de processus** qui sont rendus actifs en même temps chacun sur un **CPU différent** et ont **des quota de temps synchrones**.

# Systèmes à mémoire commune

Le système d'exploitation distribue les processus sur les différents processeurs en essayant d'équilibrer les charges des processeurs mais en respectant les règles d'ordonnancement suivantes :

- Un processus qui a été suspendu reprendra de préférence **sur le même processeur** pour éviter de recharger la totalité des pages.
- Un nouveau processus (fork) sera assigné au **processeur le moins chargé**.
- Un processus surchargé (exec) pourra être **transféré sur un autre processeur moins chargé**.

# Systèmes à mémoire commune (cas de linux 2.6)

L'ordonnanceur gère un tableau de files d'attente de processus par processeur (une entrée par priorité, 140 niveaux de priorité).

Chaque entrée de ce tableau désigne une *file des processus en attente* et une *file des processus prêts*.

Un mot binaire indique quelles sont les entrées du tableau qui désignent des files de processus en attente *non vides*. Par exemple si on n'a que des processus en priorité 2 et en priorité 4 ce mot aura les bits 2 et 4 à 1 (les autres à 0). Ceci permet de savoir rapidement dans quelle entrée du tableau aller chercher (celui de plus forte priorité non vide donc correspondant au 1<sup>er</sup> bit à un de ce mot binaire).

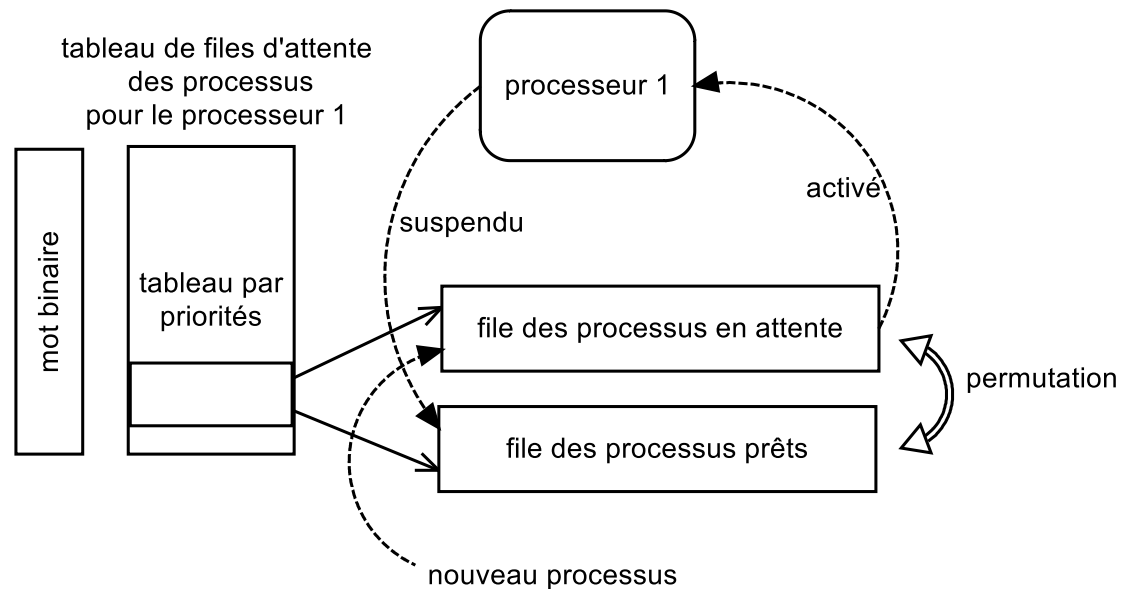
Les processus en attente de même priorité sont exécutés dans l'ordre (parcours de la file des processus en attente = round robin).



# Systèmes à mémoire commune (cas de linux 2.6)

Quand un processus termine son quota de temps :

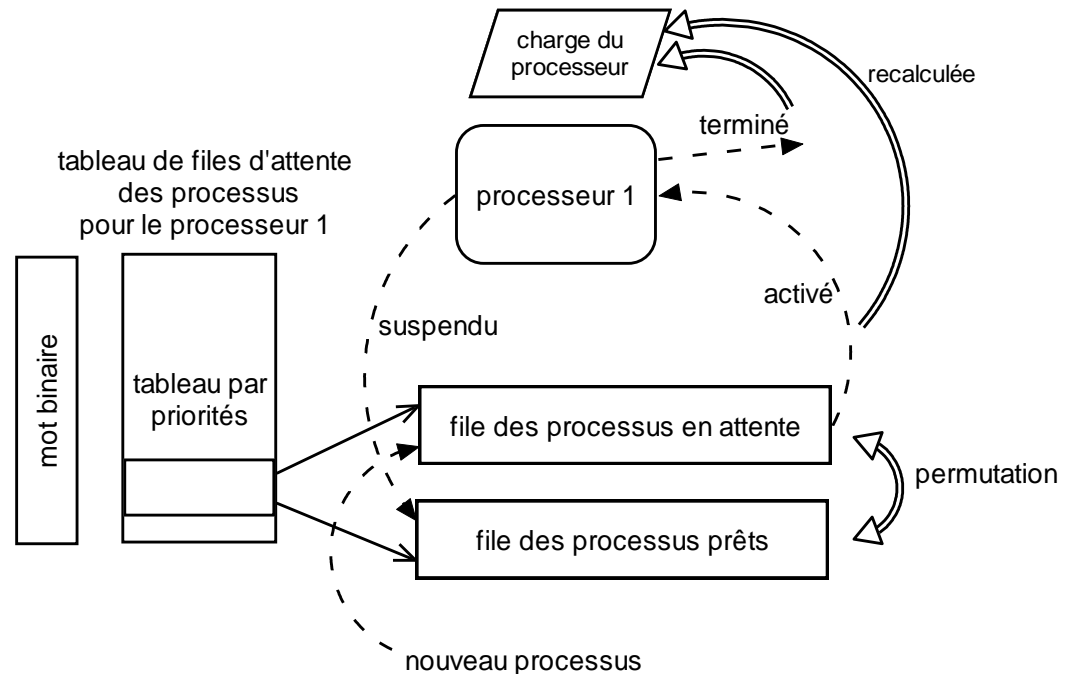
- Un nouveau quota lui est affecté
- Il est déplacé de la file des processus en attente vers celle des processus prêts de même priorité
- S'il était le dernier processus en attente de cette priorité, la file des processus prêts et celle des processus en attente sont permutées.



# Systèmes à mémoire commune (cas de linux 2.6)

Quand un processus démarre :

- Si le processus est nouveau il est affecté au processeur le moins chargé
- La charge du processeur sur lequel il est lancé est recalculée.



# Ordonnancement sur un cluster

Chaque CPU fait **son propre ordonnancement**. Le problème est alors de choisir sur quel CPU on implante un nouveau processus.

On essaye **d'équilibrer la charge et de minimiser les communications** entre CPUs.

1. Si on connaît d'avance les besoins des processus en terme de :

- temps CPU
- mémoire
- communication

On peut faire ça de façon **statique** (c'est le cas pour un gros programme de calcul qui tourne sur un cluster par exemple météo)

2. Sinon on peut le faire **dynamiquement** de 3 façons :

1. Quand un processus en crée un autre, le CPU sur lequel il est **le garde** ou **cherche un CPU peu chargé** qui veuille le prendre
2. Quand un processus se termine sur un CPU celui-ci **consulte les autres CPU** pour voir s'ils sont plus chargés que lui, si c'est le cas il leur demande un processus à exécuter

Remarque : on peut combiner ces 2 méthodes.

3. Les CPUs maintiennent dans un **fichier commun** leur **état d'occupation**. Quand un processus est créé sur un CPU celui-ci consulte ce fichier pour voir quel CPU est le moins chargé.

# Problème de synchronisation

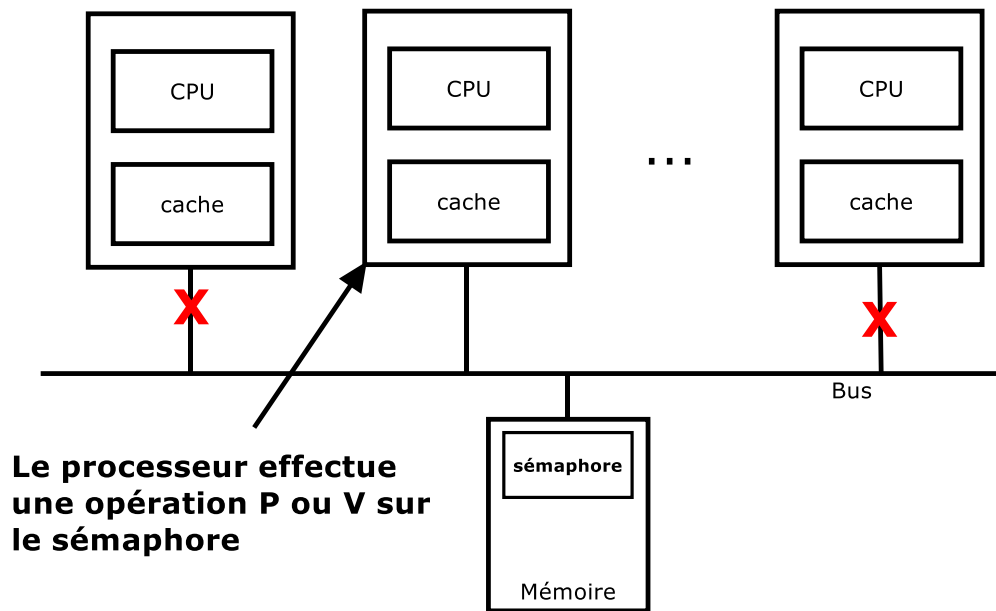
- Sur un monoprocesseur on utilise des opérations atomiques comme **P** et **V** en **bloquant les Its**
- Sur un multiprocesseur **ça ne suffit pas** car les autres CPU continuent à fonctionner

La solution dépend de l'architecture :

- Mémoire commune
- Mémoires distribuées et réseau d'interconnexion

# Synchronisation avec mémoire commune

- Le sémaphore est en mémoire et **n'est pas copié dans les caches** (variable protégée)
- **La mémoire est verrouillée** pendant une opération atomique (TSL) pour éviter qu'un autre processeur y accède.



Les processeurs possèdent **une ligne physique** de verrouillage de la mémoire.

Les **instructions** accédant à la mémoire peuvent provoquer une activation de cette ligne (préfixe **lock** sur les processeurs Intel).

# Synchronisation avec mémoires distribuées

- Le sémaphore S est dans la mémoire de **l'un des processeurs**.
- Les opérations P et V sur ce sémaphore S sont exécutées par **ce processeur seulement**.
- Elles lui sont demandées par réseau (appel de fonction à distance) par les autres processeurs
  - Lors d'une opération P le processus demandeur est bloqué jusqu'à ce que l'opération soit terminée. A la fin de l'opération, selon la valeur de S, soit il reste bloqué sur le sémaphore S soit il continue.
  - Lors d'une opération V le processus demandeur est bloqué jusqu'à ce que l'opération soit terminée. A la fin de l'opération le processus demandeur est débloqué et l'un des processus bloqués sur le sémaphore S (sur l'une des machines) est également débloqué.
- On utilise un protocole réseau (synchrone) et des primitives bloquantes (**send** et **receive**) pour ces opérations.

# Gestion des mémoires distribuées dans un cluster

Sur les clusters on n'a **pas de mémoire partagée** mais le SE peut faire **comme si**.

Chaque CPU utilise des adresses virtuelles qui correspondent à des pages mémoire présentes ou pas. Dans un système classique si une page n'est pas présente **le SE va la chercher sur disque**.

Dans un cluster **le SE peut aller la chercher sur un autre CPU** en demandant au SE de celui-ci de la lui envoyer

⇒ l'adresse virtuelle correspond à adresser toutes les mémoires de tous les CPU du cluster (ou au moins une partie de chacune)

Quand un CPU **modifie une page ainsi partagée** le SE envoie un message aux autres SE pour qu'il marquent cette page comme **non présente chez eux** ⇒ au prochain accès ils feront un transfert.

# Parallélisme

Le système d'exploitation distribue la charge entre processeurs **au niveau des processus** donc :

Si une application est mono-processus elle ne bénéficie du multiprocesseur (ou multicœur) que parce que les autres processus (système, autres utilisateurs ...) s'exécutent sur d'autres processeurs (ou d'autres cœurs).

Pour bénéficier de la puissance d'un multiprocesseur, les applications **doivent être découpées en processus parallèles**.

MAIS :

- En général les langages de programmation ne sont pas adaptés à cela.
- Les programmeurs n'ont pas l'habitude de paralléliser leur code.

***Pour être de bons programmeurs vous devez l'avoir***



# Langages et parallélisme

## Plusieurs approches :

- Détection automatique du parallélisme par le compilateur : limité au parallélisme d'exécution des instructions dans les processeurs (pas de parallélisme de tâches).
- Création de processus ou de threads par le programmeur
  - `fork` et `pthread_create` en C ou C++
  - Classe `Thread` et interface `Runnable` en java
  - Type `Task` en ADA
  - Etc.
- Parallélisation du traitement de données
  - **PLINQ** (Parallel Language-Integrated Query) de Microsoft permet de traiter en parallèle les éléments d'une collection
- Parallélisation de structures de contrôle (langages spécialisés ou extensions de langages classiques)
  - **OCCAM** langage spécialisé pour le parallélisme
  - **OpenMP** (Open Multi-Processing) permet de paralléliser l'exécution de blocs de code, de boucles et d'alternatives en C et C++ par des directives **#pragma**

# Langages et parallélisme (OCCAM)

SEQ

$op_1$

..

$op_N$

Exécute les  $op_i$  en séquence

PAR

$op_1$

..

$op_N$

Exécute les  $op_i$  en parallèle

# Langages et parallélisme (OCCAM)

SEQ

op<sub>1</sub>

PAR

SEQ

op'<sub>1</sub>

...

op'<sub>N</sub>

SEQ

op''<sub>1</sub>

...

op''<sub>K</sub>

op<sub>2</sub>

Exécute :

- op<sub>1</sub>
- puis les 2 séquences op'<sub>1</sub>...op'<sub>N</sub> et op''<sub>1</sub>...op''<sub>K</sub> en parallèle
- puis op<sub>2</sub>

# Langages et parallélisme (Open MP)

## Open MP : exemple simpliste

```
int main() {  
    int tableau[TAILLE];  
    for (int i = 0; i<TAILLE; i++) {  
        // calcul sur les éléments du tableau  
    }  
}
```

Devient :

```
int main() {  
    int tableau[TAILLE];  
    #pragma omp parallel for  
    for (int i = 0; i<TAILLE; i++) {  
        // calcul sur les éléments du tableau  
    }  
}
```

La boucle est éclatée  
en plusieurs threads  
comme vu en TD

# Communication

**MPI (Message Passing Interface)** : bibliothèque réunissant une centaine de fonctions pour les langages C et C++ permettant de faire communiquer des processus sur multiprocesseurs par passage de messages (il existe des extensions pour d'autres langages).

## Organisation des communications :

- **Intracommunicateur** : ensemble de processus pouvant communiquer
- **Intercommunicateur** : lien entre 2 intracommunicateurs

## Fonctions de communication :

- **Communication point-à-point** : permet à deux processus d'échanger une donnée (fonctions *MPI\_Send*, *MPI\_Ssend*, *MPI\_Recv* et *MPI\_Sendrecv*)
- **Communication collective** : permet d'envoyer une même donnée à tous les processus (*MPI\_Bcast*), de découper un tableau entre tous les processus (*MPI\_Scatter*) ou d'effectuer une opération à laquelle chaque processus contribue (*MPI\_Reduce*).

# Les Systèmes d'Exploitation

1. Introduction
2. Les Processus
3. Les Threads
4. Concurrency entre processus
5. Communication entre processus
6. Les interruptions logicielles (signaux)
7. Gestion du CPU et de la mémoire
8. Ordonnancement
9. Les Entrées/Sorties
10. Le système de fichiers
11. Démarrage d'un système
12. Sécurité
13. Interblocage
14. Systèmes multiprocesseurs
- 15. Les systèmes temps réel**

# Systemes temps réel

# Applications temps réel

- Applications dépendant de l'évolution dynamique d'un procédé extérieur
- L'application est correcte si elle produit les bons résultats (les bonnes actions) **et** le fait dans les temps
- Une application temps réel n'est pas forcément une application qui va vite mais une application qui satisfait les contraintes de temps
- On distingue 3 types d'applications temps réel :
  - **Sévère** : Le non respect d'une seule échéance est catastrophique
  - **Ferme** : Le respect des échéances est nécessaire, leur non respect fait que les résultats obtenus sont inutilisables
  - **Souple** : Le respect des échéances est important, leur non respect réduit l'intérêt des résultats obtenus



# Les systèmes temps réel

Un système temps réel offre au programmeur :

- La possibilité de définir des tâches (processus / threads)
- Des mécanismes de synchronisation (sémaphores)
- Des mécanismes de communication (boîte à lettres, pipes, signaux ...)
- **La possibilité de gérer les interruptions**

L'objectif principal est la prise en compte des besoins d'urgence, d'importance et de réactivité. La seule contrainte est que tout soit exécuté *à temps*. C'est donc essentiellement un problème d'ordonnancement des tâches.

- Principes d'ordonnancement des tâches :
  - Choix de l'ordonnancement
    - Statique (à l'avance) / Dynamique (au moment)
  - Priorités
    - Statiques / Dynamiques
  - Algorithmes
    - Préemptifs / Non préemptifs

# Les systèmes temps réel

Types de tâches :

- Normales / Urgentes
- Répétitives / Non répétitives

**Les tâches répétitives et les tâches urgentes sont critiques**

Les tâches répétitives peuvent être :

- Périodiques (à intervalles réguliers)
- Sporadiques (déclenchées par un événement)

De plus les tâches peuvent être :

- Dépendantes / Indépendantes

# Algorithme Rate Monotonic (RMA)

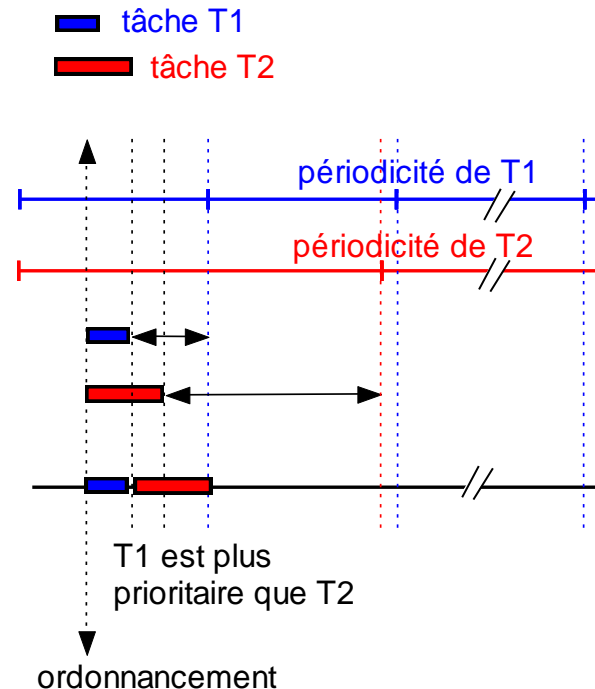
Algorithme à **priorités fixes** adapté aux applications statiques.

- Tâches urgentes :  
Assigner des priorités élevées selon l'urgence
- Tâches non répétitives :  
Assigner les priorités les plus faibles
- Tâches répétitives périodiques :  
Assigner des priorités selon la périodicité de la tâche (plus la période est courte plus la priorité est forte).
- Tâches répétitives sporadiques :  
Assigner des priorités selon l'importance de l'événement.
- Calculer à l'avance si ces choix sont viables dans le cas pire (durée maximale des tâches) : **Parfois difficile à faire en raison des événements non prévisibles.**
- L'ordonnancement est ensuite très simple (statique)

# Algorithme Earliest Deadline First (EDF)

Algorithme à **priorités variables** adapté aux applications dynamiques.

Pour toutes les tâches (périodiques ou non) on calcule une priorité en fonction de leur date d'échéance.



# Systemes embarqués

# Systèmes embarqués

- Si le système est simple il n'y a pas de SE mais un **moniteur** qui :
  - Initialise le matériel (registres du processeur et des contrôleurs de périphériques)
  - Contient les pilotes de périphériques (facultatif)
  - Offre une API minimale (facultatif)
  - Peut communiquer (USB ou JTAG) avec une autre machine qui :
    - Permet d'éditer et compiler les applications
    - Permet de les télécharger sur le système embarqué
    - Permet parfois de les tester (traces, points d'arrêt ...)
    - Permet de mettre à jour le moniteur

Tout le reste est fait par l'application

# Systèmes embarqués

- Lorsqu'il y a un SE, deux approches :
  - **Système léger** :
    1. SE existant épuré pour ne garder que ce qui est nécessaire
      - Linux (Android)
      - IOS (Apple) dérivé de Mac OS X
      - Window Phone (Microsoft) dérivé de Windows
    2. SE développé pour l'embarqué
      - Symbian (Nokia) , QNX (Blackberry) ...
      - VxWork (Wind River) utilisé par la NASA, pSOS (SCG) racheté par Wind River et abandonné, Squawk (Sun) racheté par Oracle et abandonné
  - **Système modulaire** :
    - le SE est divisé en modules (ordonnanceur, gestion de mémoire, ...) et, selon ce dont a besoin l'application, on n'installe que certains modules.
      - RTOS (système temps réel sous forme de bibliothèque)
      - Tiny OS (associé au langage de programmation NesC)

# Exemple : création d'un linux allégé

*(d'après Jalil Boukhobza Université de Bretagne Occidentale)*

## 1. Optimiser le noyau :

- Certaines parties peuvent être supprimées ou transformées en modules installables si besoin
- Des pilotes peuvent être enlevés (pas de carte son, pas d'imprimante, ...)
- Ajuster les options du noyau :
  - Nombre max de processus, de sémaphores, ...
  - Réseau (protocoles inutiles)
  - Support USB, SATA, SCSI ...



# Exemple : création d'un linux allégé

(d'après *JalilBoukhobza Université de Bretagne Occidentale*)

## 2. Répertoires :

- Certains ne servent qu'en multi utilisateurs (/home /mnt /opt ...)  
⇒ on peut les enlever

## 3. Bibliothèques :

- Celles de base sont lourdes (libc), il existe des versions simplifiées (uClibc ou Diet libc)

## 4. Drivers de périphériques :

- Enlever ceux qui ne sont pas disponibles
- Ajouter ceux qui manquent (capteurs)

# Exemple : création d'un linux allégé

*(d'après JalilBoukhobza Université de Bretagne Occidentale)*

## 5. Commandes :

- En enlever (installation, compilation, édition, ...)
- En simplifier (login, comptes, ...)

## 6. Démarrage

- Linux propose 7 niveaux de démarrage certains sont inutiles en embarqué

## 7. Adapter le système de fichiers

- En RAM
- Sur mémoire flash

# SE modulaire



- Exemple de RTOS (Real Time Operating System) SE pour systèmes embarqués
  - Contraintes :
    - Peu de place en mémoire (quelques Ko)
    - Nécessité de traitements temps réel (interruptions, timers)
  - Caractéristiques :
    - Un système embarqué est mono programme mais ce programme peut être multi processus
    - Dans un système embarqué c'est le programmeur qui gère le matériel et les interruptions physiques pas le SE (temps réel)
  - Conséquences :
    - Inutile d'avoir un SE complet dont on n'utilise qu'une partie (par exemple tout le code relatif à la gestion des boîtes à lettres est inutile si on ne s'en sert pas)

# SE modulaire



- RTOS pour systèmes embarqués (solution)
  - Le SE n'est pas résident. Il se présente sous la forme d'une bibliothèque de fonctions
  - Seules les fonctions utilisées par un programme seront présentes dans le code généré
  - Le programmeur :
    - Ecrit les threads (tâches à effectuer)
    - Ecrit les fonctions associées aux interruptions et aux signaux
    - Ecrit un programme principal qui :
      - Initialise le matériel et les variables partagées
      - Crée les mécanismes de synchronisation et de communication (sémaphores, pipes, ...)
      - Associe les fonctions aux interruptions physiques et aux signaux
      - Crée les threads et leur attribue des priorités
      - Devient l'ordonnanceur du système (appel de fonction à boucle infinie)

# SE modulaire



- Le programme embarqué contient :
  - Le code métier : les tâches à réaliser
  - Les initialisations (du matériel, des tâches, des mécanismes de communication et de synchronisation)
  - Les fonctions associées aux interruptions physiques et aux signaux
  - Les morceaux du SE utilisés
- Ce programme est complètement autonome et contient tout ce dont on a besoin pour réaliser l'application (il est mono application mais multitâche)
- Les tests peuvent être faits à l'aide de sondes (JTAG) reliées à un PC pour :
  - Mettre des points d'arrêt
  - Lancer / relancer le programme
  - Voir / modifier les registres du processeur et des contrôleurs de périphériques
  - Voir / modifier les variables en mémoire

# SE modulaire



TinyOS est écrit en nesC qui un langage dérivé du C qui inclut des notions :

- de composants,
- d'interfaces bidirectionnelles,
- de tâches
- de gestion des interruptions.

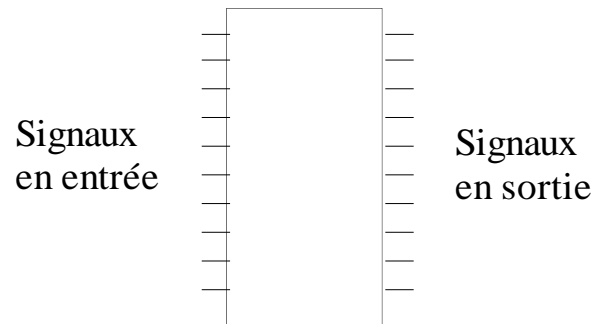
TinyOS et nesC sont liés : les applications pour TinyOS doivent être écrites en nesC.

# Principes de TinyOS

- Un **composant** est constitué d'au moins un **module** utilisant et fournissant des **interfaces**. S'il contient plusieurs modules les liens entre eux sont décrits par un fichier de **configuration**.
- Une application complète est un composant contenant plusieurs modules liés entre eux dont un module **Main** qui permet de démarrer.
- Le SE est constitué d'une centaine de composants que l'on peut utiliser pour écrire des applications. Quand le programme est généré par le compilateur, seuls les composants utilisés (y compris ceux du système) sont présents.
- Le module **Main** est lui-même connecté à certains composants du système (comme l'ordonnanceur par exemple) qui seront donc chargés en mémoire puis lancés par Main

# Application en nesC

- Une application consiste en des composants reliés entre eux exactement comme un montage électronique.
- Un composant électronique a des entrées (des signaux qu'il utilise) et des sorties (des signaux qu'il fournit) :



On fera de même pour un composant logiciel. Les signaux en entrée sont utilisés par ce composant (donc fournis par d'autres) tandis que les signaux en sortie sont fournis par ce composant (donc utilisés par d'autres).

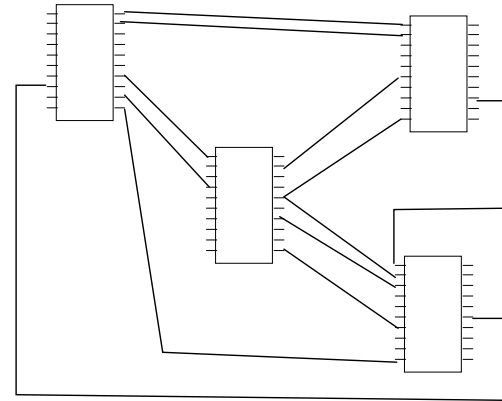


# Application en NESC

On va ensuite interconnecter ces composants entre eux.

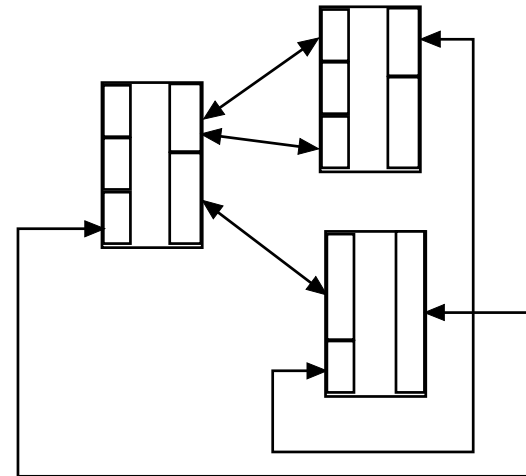
En électronique on a un schéma de câblage du type :

Il faut savoir quelles pattes d'un composant relier à quelles autres.



Pour les composants logiciels on va organiser les pattes (commandes/événements) en groupes (**interfaces**) et relier directement des groupes entres eux.

Toutes les pattes d'un groupe sont reliées à toutes les pattes d'un autre groupe (leur nom permet de savoir lesquelles sont reliées auxquelles).

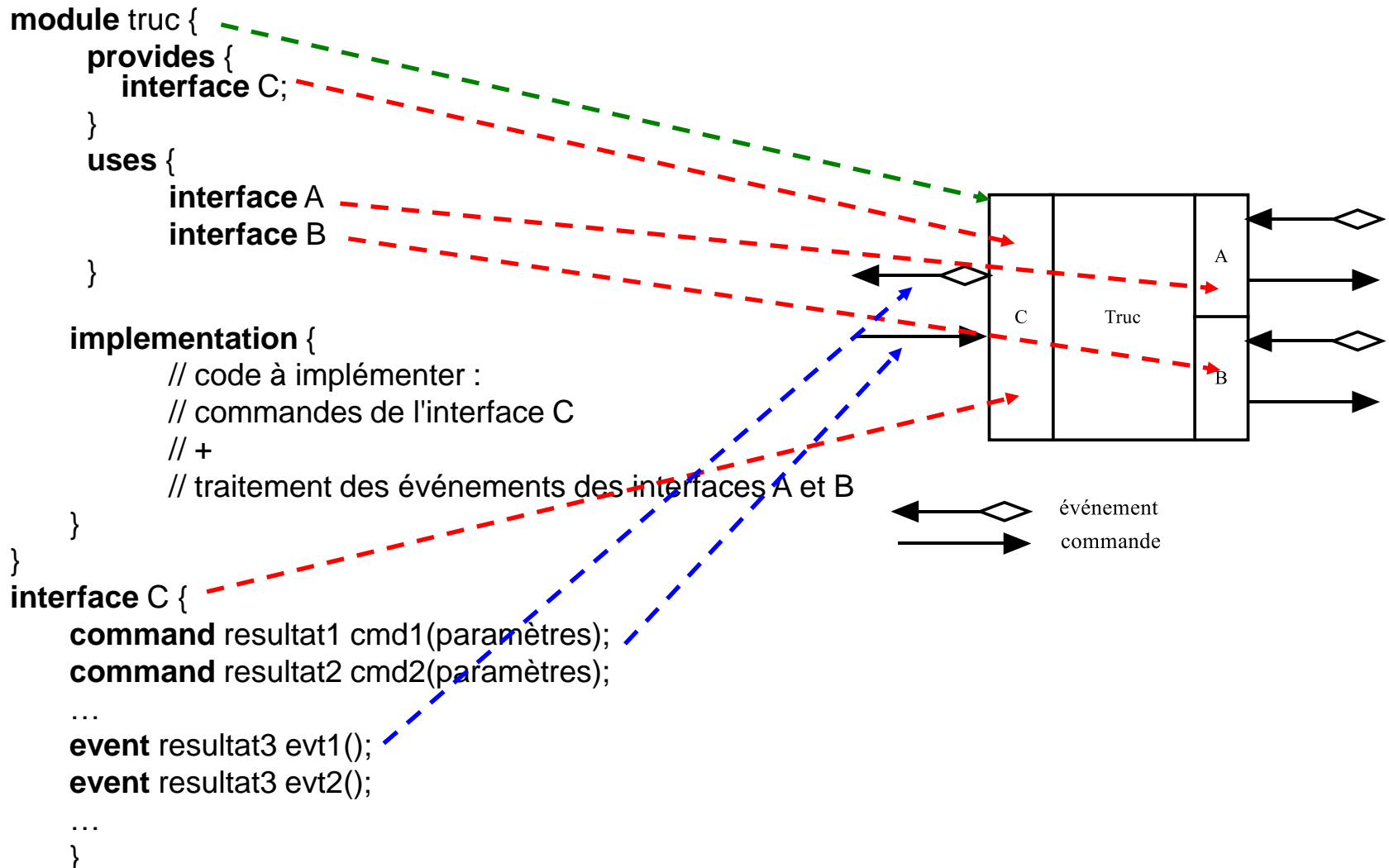


# Modules et interfaces

Les **modules** offrent et utilisent des **interfaces**.

- On distingue dans une interface les **commandes** (sorties) et les **événements** (entrées). Les liaisons établies entre interfaces sont donc **bidirectionnelles** car certaines pattes sont des entrées et d'autres des sorties.
- Une interface déclare un ensemble de fonctions appelées **commands** qui peuvent être utilisées par les autres composants et un ensemble de fonctions appelées **events** qui réagissent à des événements en provenance d'autres composants.
- Les **commandes** seront utilisées par un appel explicite (comme un appel de fonction ou de méthode) tandis que les **événements** provoqueront automatiquement l'exécution du code approprié dans le composant qui les reçoit (comme un signal).

# Modules et interfaces



# Liaisons

Un composant en nesC est constitué de **modules** et d'une **configuration**.

- Les **modules** contiennent le code et implémentent une ou plusieurs interfaces.
- Les **configurations** servent à assembler les composants entre eux en connectant les interfaces utilisées par certains composants à celles fournies par d'autres pour fabriquer de nouveaux composants (composites).
- L'**application** est elle-même un composant n'offrant aucune interface (composant de plus haut niveau).

# Liaisons :

Fabriquer le composant C3 à partir de C1 et C2.

```
configuration C3 {
```

```
  provides {
```

```
    interface A;
```

```
    interface B
```

```
  }
```

```
  implementation {
```

```
    components C1,C2;
```

```
    A = C1.A;
```

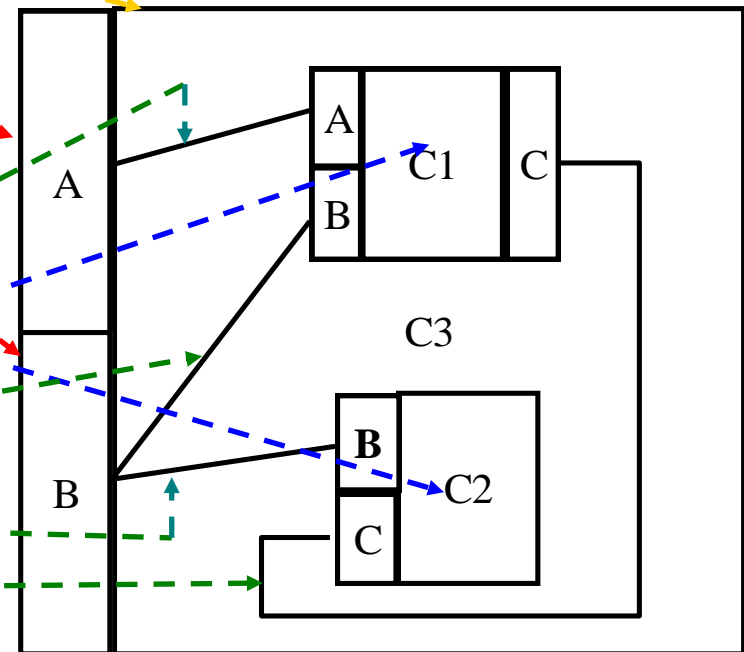
```
    B = C1.B;
```

```
    B = C2.B;
```

```
    C1.C → C2.C;
```

```
  }
```

```
}
```



# Application NESC

- Une application est un assemblage dans lequel apparaît le composant **Main** qui sert à démarrer.  
Elle est décrite par une configuration "de premier niveau".
- L'ensemble des configurations permet de savoir **quels composants sont utiles pour que l'application fonctionne**.

Il suffit de regarder le fichier de configuration de l'application pour savoir quels composants elle utilise puis de faire la même chose récursivement pour chacun de ces composants.

**Les composants du système d'exploitation sont donc installés en fonction des besoins de l'application**

# Virtualisation

# Machine virtuelle

- Si l'on écrit un logiciel qui a pour but de simuler un ordinateur qui a :
  - Un CPU
  - Une mémoire
  - Une unité d'Entrée/Sortie avec des périphériques (dont disque)
  - Un BIOS
- Quand on exécute ce logiciel on se retrouve comme avec un ordinateur nu
- Généralement cet ordinateur aura un matériel différent de celui de l'ordinateur réel (partage des ressources matérielles).
  - CPU avec moins de cœurs
  - Mémoire plus petite ou pas (mémoire virtuelle)
  - Unité d'E/S moins complète



# Machine Virtuelle

- Si l'on installe sur cet ordinateur nu un système d'exploitation, il devient utilisable comme un ordinateur classique.
- Le système s'installe sur le disque (initialement vierge) de cet ordinateur nu.
- Ensuite on peut installer et exécuter des applications sur cet ordinateur
- Ces applications sont exécutées par le(s) processeur(s) de la machine physique et utilisent (partiellement) ses ressources (mémoire, disque, périphériques)

# Avantages

- Meilleure utilisation des ressources physiques (en définissant plusieurs MV)
- Evolutivité (adaptation du matériel)
- Répartition de charge (MV déplaçables)
- Flexibilité (clonage d'une MV)
- Sécurité (les MV sont isolées entre-elles)

# Autres types de MV

- Le logiciel n'a pas pour but de simuler une machine physique différente mais une machine ayant un processeur dont le jeu d'instructions est différent.
- Exemples :
  - MV java : le jeu d'instruction est le byte code généré par le compilateur java
  - MV python : le jeu d'instruction est le byte code généré par le compilateur python
- Avantage : portabilité du code compilé

# Le matériel

- Les constructeurs ont adapté le matériel à l'usage des machines virtuelles :
  - CPU : instructions dédiées et registres supplémentaires
  - Mémoire : zones marquées comme contenant des informations non exécutables (protection)
  - MMU : tables de pages étendues pour tenir compte des zones allouées aux différentes machines virtuelles
  - DMA : partage des canaux et possibilité de redirection d'adresses
  - Unité d'E/S :
    - gestion des interruptions tenant compte des cœurs alloués aux machines virtuelles
    - MMU pour les gestionnaires de périphériques
    - Partage des cœurs des GPU des cartes graphiques
    - Partage des connexions réseau (par redirection des données)

# Hyperviseur

- Quand on exécute plusieurs machines virtuelles sur la même machine se pose le problème du partage des ressources physiques de la machine réelle entre les diverses machines virtuelles.
- C'est le rôle de l'**hyperviseur**.
- L'hyperviseur alloue des ressources physiques (processeurs, mémoire, stockage ...) à chaque machine virtuelle.
- Chaque machine virtuelle reste distincte des autres et n'interfère pas avec les autres.
- Une machine virtuelle peut être démarrée, arrêtée ou migrée à tout moment sans perturber les autres

# Hyperviseur

- Il existe deux principaux types d'hyperviseurs :
  - **Type 1** : l'hyperviseur remplace le système d'exploitation de la machine physique
  - **Type 2** : l'hyperviseur s'exécute sur le système d'exploitation de la machine physique comme une application classique
- L'hyperviseur permet :
  - D'ajouter/enlever des machines virtuelles
  - De les arrêter/démarrer
  - De les déplacer vers une autre machine physique (vers un autre hyperviseur)

# Hyperviseur de type 1

- Utilisés en majorité en entreprise
- Plus efficace car accède directement aux ressources de la machine physique.
- Sa sécurité ne dépend pas de celle du système d'exploitation sous-jacent.
- Il faut en général un logiciel distinct pour administrer les machines virtuelles (parfois même sur une machine différente).
- Exemples :
  - ESXi (Vmware)
  - Hyper-V (Microsoft)
  - Citrix (XenServer)
  - KVM (Linux)

# Hyperviseur de type 2

- Le plus souvent utilisés pour des tests ou sur un PC
- Plus simple à utiliser car s'installe comme une application classique
- On peut installer plusieurs hyperviseurs sur la même machine physique
- Moins performant car doit passer par le système d'exploitation pour accéder aux ressources physiques
- Plus sensibles aux défauts de sécurité du système d'exploitation sous-jacent.
- Exemples :
  - VirtualBox (Oracle)
  - Workstation (Vmware)



# Conteneurs

- **Les conteneurs ne sont pas des machines virtuelles**
- Un conteneur ne virtualise pas un ordinateur mais le système d'exploitation
- Un conteneur contient une application, ses variables d'environnement, ses fichiers de configuration, ses bibliothèques et ses dépendances.
- L'application s'exécute sur le système d'exploitation de la machine physique (elle est donc liée à ce SE)
- Un conteneur peut être créé/démarré/arrêté/déplacé comme une MV
- Ce sont des solutions beaucoup plus légères mais plus limitées
- Exemples :
  - Kubernetes
  - Docker

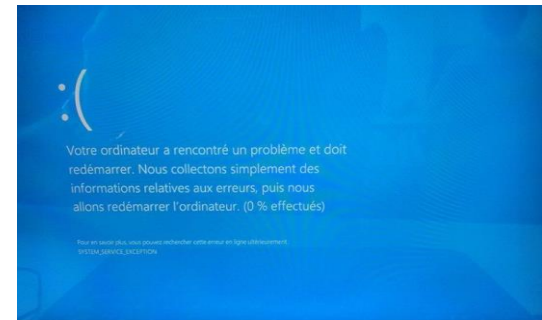
# Pour conclure

Un bon informaticien doit savoir ce que fait un système d'exploitation en termes :

- De gestion du processeur
- De gestion des processus
- De gestion de la mémoire
- De gestion des E/S
- De gestion du système de fichiers
- De gestion des utilisateurs
- De gestion de la sécurité

Pour pouvoir :

- L'installer
- Le configurer
- Résoudre les problèmes



# Pour conclure

Un bon informaticien doit savoir ce qu'offre un système d'exploitation en termes :

- De parallélisme (processus / threads)
- De communication (pipes / signaux / bêt / mémoire partagée ...)
- De synchronisation (sémaphores / signaux)

Pour pouvoir :

- Exploiter les performances de la machine
- Ne pas bloquer une application qui attend quelque chose
- Offrir une meilleure expérience utilisateur

**Il faut y penser dès la conception de l'application**

# Les Systèmes d'Exploitation

