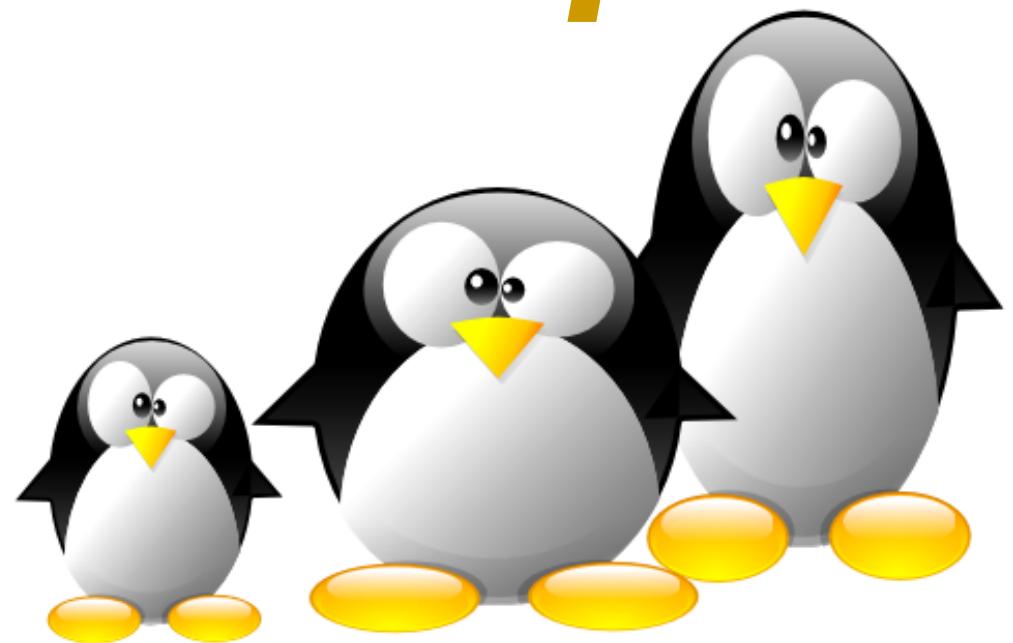


*Part I*

# *Linux Embarqué*



*embedded* **Linux**

# **Chapitre 1:** Fonctionnalités des OS (Le cas de Linux)

# Plan

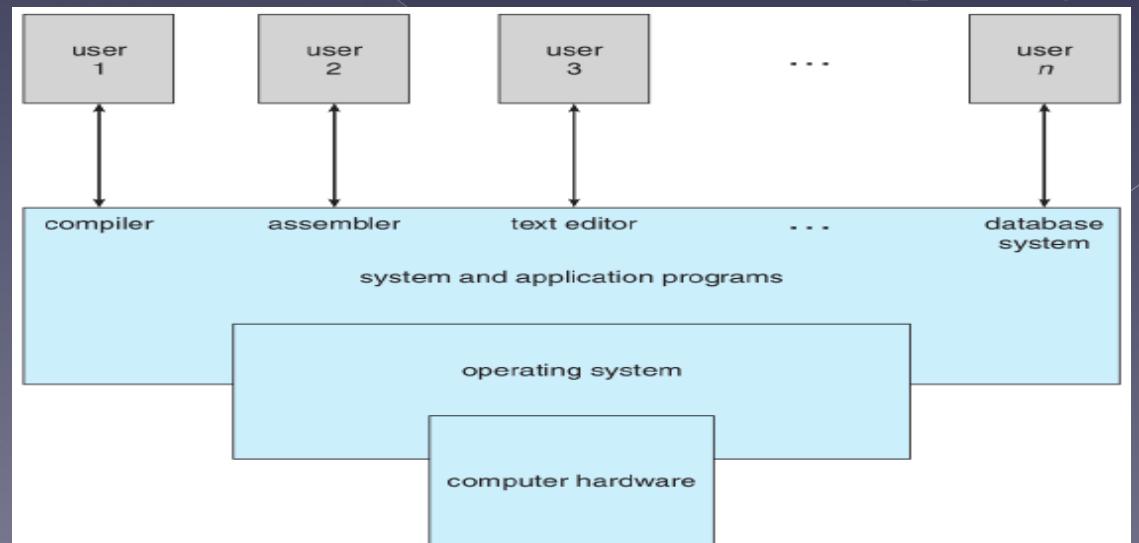
- Généralités sur les OS
- Histoire de GNU Linux
- Architecture du système UNIX/Linux
- Multitâches et Commutation (**Programmation Système**)
- Interruptions, Exceptions et Appels système

# Généralités sur les OS

Le système d'exploitation (noté **SE** ou **OS**, abréviation du terme anglais **Operating System**), est chargé d'assurer la liaison entre les ressources matérielles, l'utilisateur et les applications.

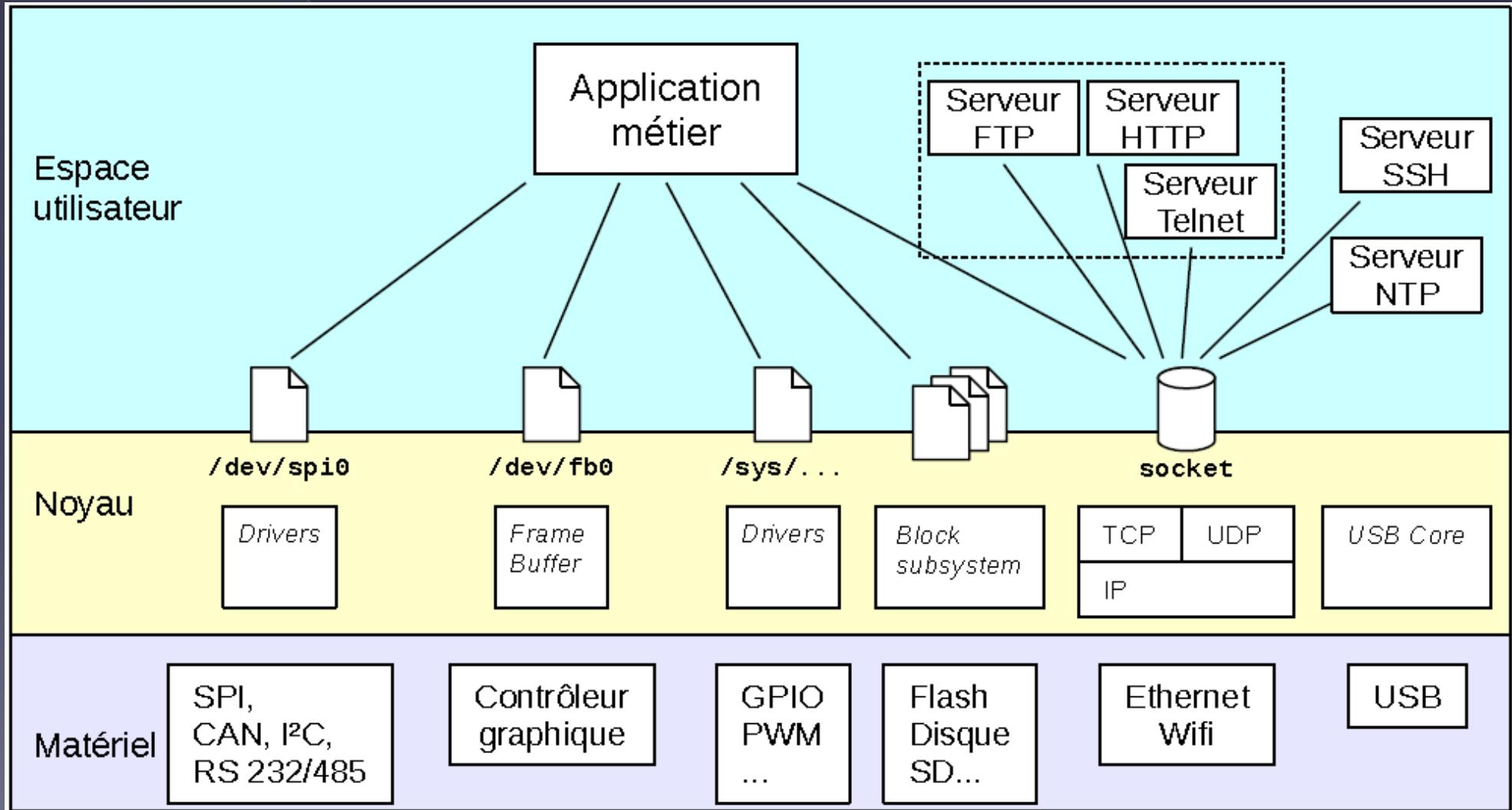
Deux tâches :

- Fournir à l'utilisateur une machine étendue ou virtuelle, plus simple à programmer.
- Gestion des ressources. Deux dimensions du partage (multiplexage) :
  - Temps
  - Espace



# Généralités sur les OS

## Abstraction des périphériques



Masquer la complexité matérielle / simplifier les accès au matériel

# Généralités sur les OS

## Les systèmes d'exploitation permettent:

- De gérer les ressources matérielles en assurant leurs partages entre les différents utilisateurs.
- De présenter une interface homogène et générique(en abstrayant la complexité matérielle) mieux adaptée aux utilisateurs.
- Contrôle de processus sans (ou à faible) contrainte temporelle  
→ **Systèmes à temps partagé**
- Garantir le partage équitable du temps et des ressources

# Généralités sur les OS

## Pourquoi un système d'exploitation pour l'embarqué ?

- Affranchir le développeur de logiciel embarqué de bien connaître le matériel
  - gain en temps de développement  
Les applications doivent avoir un accès aux services de l'OS
    - via des APIs (réutilisabilité du code, interopérabilité, portabilité, maintenance aisée)
- Possibilité de bénéficier des mêmes avancées technologiques que les applications classiques (TCP/IP, HTTP, etc.)
- Environnement de développement plus performant
- Contrôle de processus avec contrainte temps réel
  - Systèmes temps réel: Garantir les temps de réponse

# Généralités sur les OS

## Les latences

**Définition:** différence entre le moment où une tâche doit débuter (ou finir) et le moment où elle débute réellement.

- Elles sont dues:
  - Aux propriétés temporelles des processeurs, des bus mémoire et d'autres périphériques
  - Aux propriétés des politiques d'ordonnancement
  - À la préemptivité du noyau
  - À la charge du système
  - Au changement de contexte

# Histoire de GNU Linux

## Logiciel libre, concepts et licences

La licence est un document autorisant l'utilisation d'un logiciel sous certaines conditions ; elle constitue un contrat entre l'éditeur et l'utilisateur. Une licence libre ajoute trois libertés fondamentales :

- Utiliser le logiciel (même commercialement) ;
- Etudier et modifier le code source ;
- Distribuer la version modifiée.

Le logiciel libre ne doit pas être confondu avec le *freeware*, dont le code source n'est pas disponible et la licence pas forcément compatible avec le modèle libre.

# Histoire de GNU Linux

## Le logiciel libre ?

- Un programme est considéré comme libre lorsque sa licence offre à tous les utilisateurs les quatre points de liberté suivantes:
  - La liberté d'exécuter le logiciel pour n'importe quel but
  - La liberté d'étudier le logiciel et de le modifier
  - La liberté de redistribuer des copies
  - La liberté de distribuer des copies de versions modifiées
- Ces libertés sont accordées pour une utilisation commerciale et non commerciale
- Elles impliquent la disponibilité du code source, le logiciel peut être modifié et distribué aux clients
  - **Bon match pour les systèmes embarqués !**

# Histoire de GNU Linux

## Mais qu'est ce que l'Open source?

L'**open source** repose sur les principes du logiciel libre, mais est né d'une scission avec la **FSF** (Free Software Foundation) vers 1998 et la création de l'**OSI** (Open Source Initiative) par **Eric Raymond**.

Selon **Richard Stallman**, la différence fondamentale entre les deux concepts réside dans leur philosophie :

*« L'open source est une méthodologie de développement, alors que le logiciel libre est un mouvement social. »*

# Histoire de GNU Linux

## Mais qu'est ce que l'Open source?

1. **Libre redistribution:** en tant que composant d'une distribution  
→ pas de droit d'auteur.
2. **Inclusion du code source:** code source doit être accessible sans frais supplémentaires.
3. **Autorisation de travaux dérivés:** modification et travaux dérivés et leur redistribution
4. **Intégrité du code source de l'utilisateur**
5. **Pas de discrimination entre les personnes ou les groupes**
6. **Pas de discrimination entre les domaines d'applications.**
7. **Distribution systématique de la licence**
8. **La licence ne doit pas être spécifique à un produit**
9. **La licence ne doit pas contaminer d'autres logiciels**

# Histoire de GNU Linux

## Naissance du logiciel libre

- **1983**, Richard Stallman, Projet GNU et le concept de logiciel libre. Début du développement de gcc, gdb, glibc et d'autres outils importants
- **1991**, Linus Torvalds, Linux kernel project, De plus avec les logiciels de GNU et de nombreux autres composants open-source:
  - Un système d'exploitation entièrement libre, **GNU / Linux**
- **1995**, Linux est de plus en plus devient populaire sur les systèmes serveur.
- **2000**, Linux est de plus en plus devient populaire sur les systèmes embarqués
- **2008**, Linux devient le plus populaire sur les appareils mobiles
- **2010**, Linux devient le plus populaire sur les téléphones

# Histoire de GNU Linux

## Licence GPL

Les logiciels libres sont pour la plupart régis par des licences très structurées, dont une des plus célèbres est la **GPL** (**G**eneral **P**ublic **L**icense) produite par la Free Software Foundation de Richard Stallman dans le cadre du projet **GNU**.

La GPL est basée sur la notion de *copyleft* (opposé du *copyright*), qui oblige toute modification d'un logiciel sous GPL à être publiée. Les trois principes de la GPL peuvent s'énoncer comme suit:

- La licence s'applique uniquement en cas de redistribution.
- Un code source basé sur du code GPL est considéré comme du travail dérivé et doit être publié.
- Dans le cas d'un programme exécutable, un lien statique ou dynamique entre des composants propriétaires et des composants sous GPL est illégal.

# Histoire de GNU Linux

## Licence GPL v3

La **GPL v3** est une nouvelle version sortie en 2007 afin de mettre fin (selon Richard Stallman) à la tivoisation de l'industrie. Plus généralement, le problème invoqué est la fourniture du code source (**GPL**), mais sans garantir le fonctionnement du système modifié, car ce dernier utilise forcément des composants non libres.

Contrairement à la **GPL v2**, la **GPL v3** oblige le fabricant à garantir le fonctionnement d'un logiciel libre modifié.

# Histoire de GNU Linux

- **1991:** Le noyau Linux est écrit à partir de zéro (from scratch) en 6 mois par **Linus Torvalds** dans sa chambre de l'université d'Helsinki, afin de contourner les limitations de son **PC 80386**.
- **1991:** Linus distribue son noyau sur Internet. Des programmeurs du monde entier le rejoignent et contribuent au code et aux tests.
- **1992:** Linux est distribué sous la licence *GNU GPL*
- **1994:** Sortie de Linux 1.0



# Histoire de GNU Linux

- **1994:** La société Red Hat est fondée par Bob Young et Marc Ewing, créant ainsi un nouveau modèle économique basé sur une technologie OSS.
- **1995:** GNU/Linux et les logiciels libres se répandent dans les serveurs Internet.
- **2001:** IBM investit 1 milliard de dollars dans Linux
- **2002:** L'adoption massive de GNU/Linux démarre dans de nombreux secteurs de l'industrie.

# Les développeurs du noyau GNU Linux officiels



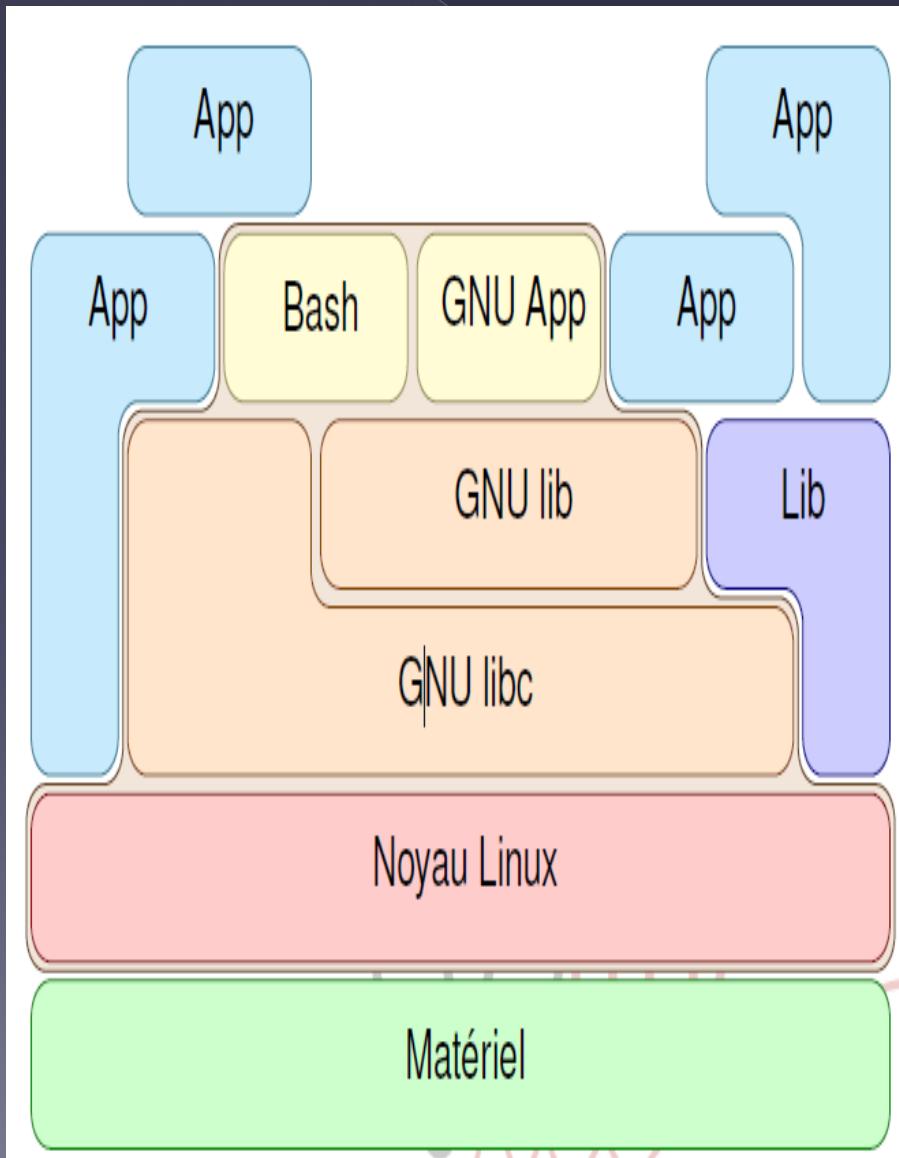
# Histoire de GNU Linux

## Qu'est-ce que Linux ?

- **Linux ne désigne que le noyau**
- Linux est souvent associé aux outils **GNU** d'où le nom de **GNU/Linux**
- Systèmes avec les outils **GNU** mais un noyau différent : **GNU/Hurd, Solaris, etc...**
- Systèmes Linux sans **GNU** : Android

# Histoire de GNU Linux

## GNU/Linux est finalement



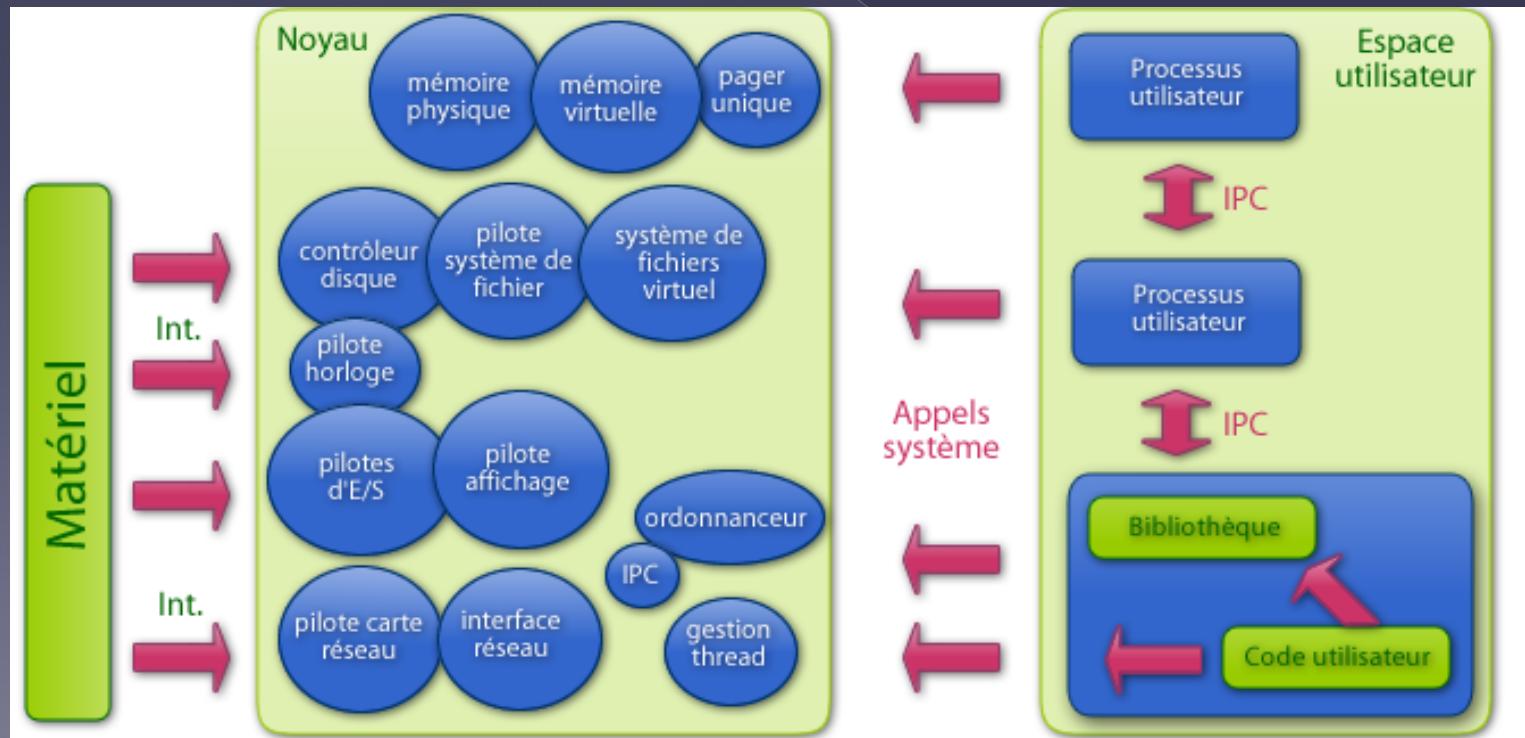
**Noyau (kernel)** : réalise les fonctions essentielles (gestion des tâches et mémoire), interface entre le matériel et les applications (pilotes).

**Lib c** : bibliothèque principale contenant les fonctions de base utilisées par les applicatifs.

**Applications (ou commandes)**: livrées avec le système ou développées pour des besoins spécifiques.

# Architecture du système UNIX/Linux

- Un **noyau (kernel)** réalise les fonctions essentielles comme la gestion des tâches et de la mémoire, ainsi que l'interfaçage entre le matériel et les applicatifs, grâce aux appels systèmes et aux pilotes de périphériques.
- Les exécutables du système sont pour certains indispensables.
- De nombreuses bibliothèques sont utilisées par les applicatifs.



# Architecture du système UNIX/Linux

## Le noyau Linux

Le noyau est l'élément principal du système, et ce pour plusieurs raisons.

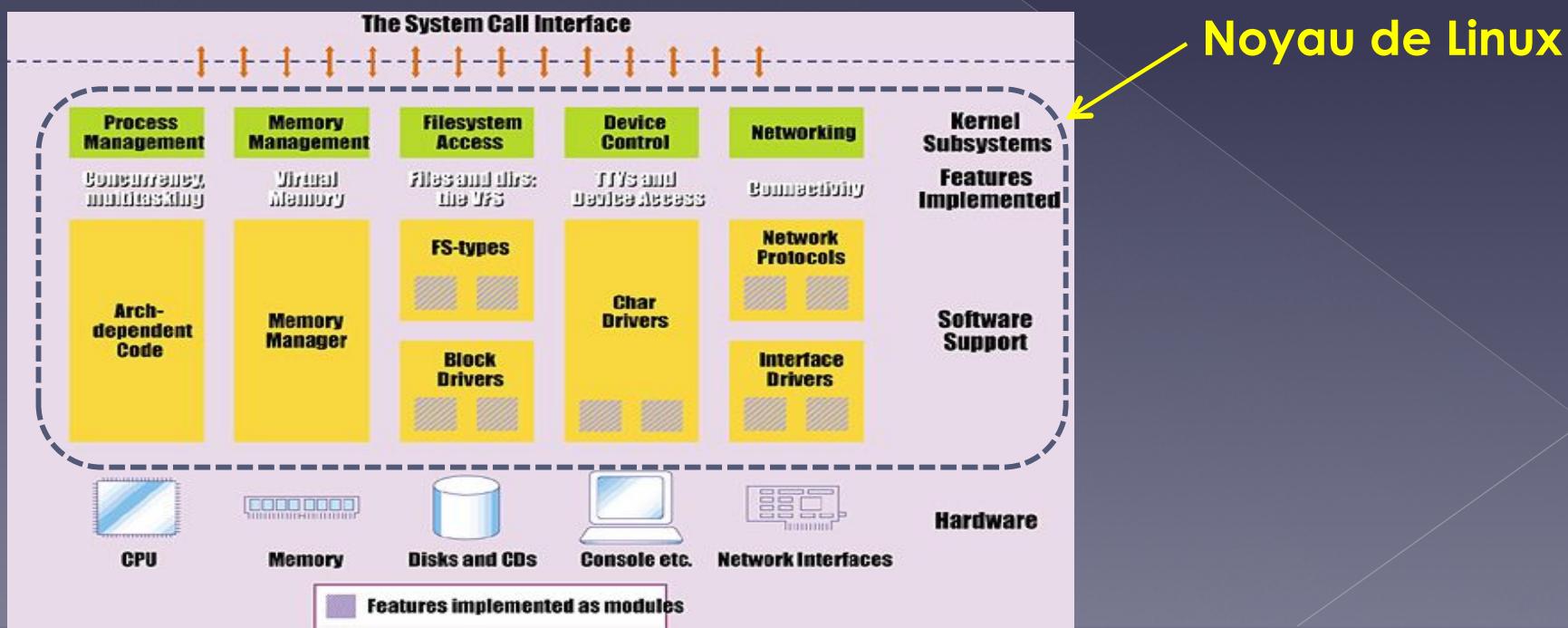
- La première est historique, puisque ce noyau fut initialement conçu par **Linus Torvalds**, le reste du système étant constitué de composants provenant en majorité du projet GNU de Richard Stallman. On parle donc du système d'exploitation **GNU/Linux**.

- La deuxième raison est technique et tient à la structure **monolithique** du noyau, qui en fait l'interface unique entre le système et le matériel.

# Architecture du système UNIX/Linux

## Le noyau Linux

Le noyau Linux est donc un fichier exécutable unique (`vmlinuz`, `vmlinux`, `zImage`, `bzImage`) chargé d'assurer les fonctions essentielles du système comme l'ordonnancement des tâches, la gestion de la mémoire et le pilotage des périphériques, que ceux-ci soient matériels ou bien virtuels comme les systèmes de fichiers (`VFAT`, `EXT3/4`).



# Architecture du système UNIX/Linux

## Système de fichier Linux

L'organisation des fichiers d'un système Linux est définie par un document intitulé dans sa version originale le **Filesystem Hierarchy Standard (FHS)**.

D'après le FHS, le système de fichiers de Linux est organisé de la manière suivante:

/	Racine du système		
bin	Principales commandes utilisateur		
boot	Noyaux statiques	proc	Système de fichiers virtuel /proc
dev	Pseudo-fichiers (device nodes)	sbin	Principales commandes système
etc	Fichiers de configuration	sys	Système de fichiers virtuel /sys
lib	Bibliothèques partagées	tmp	Répertoire temporaire
media	Points de montage dynamiques	usr	Hiérarchie secondaire
mnt	Points de montage temporaires	var	Données variables
opt	Applications externes		

# Architecture du système UNIX/Linux

## Système de fichier Linux

Les différents systèmes de fichiers situés en dessous de la racine peuvent être :

- **partageables**, que l'on peut utiliser entre plusieurs machines, par exemple à travers un montage NFS (ex. : /opt) ;
- **non partageables**, locaux à une machine (ex. : /etc) ;
- **statiques**, qui ne sont pas modifiés au cours du fonctionnement de la machine (ex. : /usr) ;
- **variables**, qui sont modifiés au cours du fonctionnement de la machine (ex. : /var).

# Architecture du système UNIX/Linux

- **/dev** : Fichiers spéciaux de lien avec les périphériques matériels

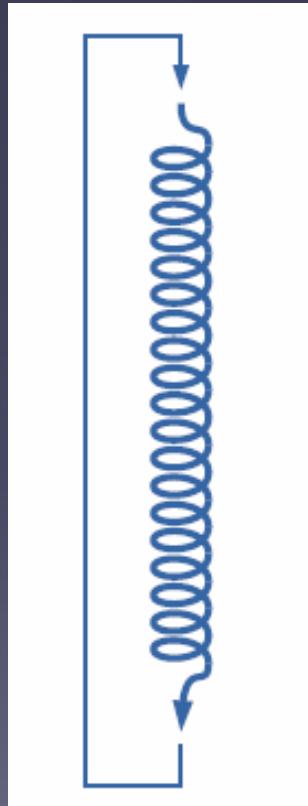


- **Types de bus :**
  - **hd** : Périphériques IDE
  - **sc** : Périphériques SCSI
  - **sd** : Périphériques SATA
- **Exemples :**
  - **/dev/hda1** : Partition 1 sur le 1er disque IDE
  - **/dev/sdb2** : Partition 2 sur le 2ème disque Sata

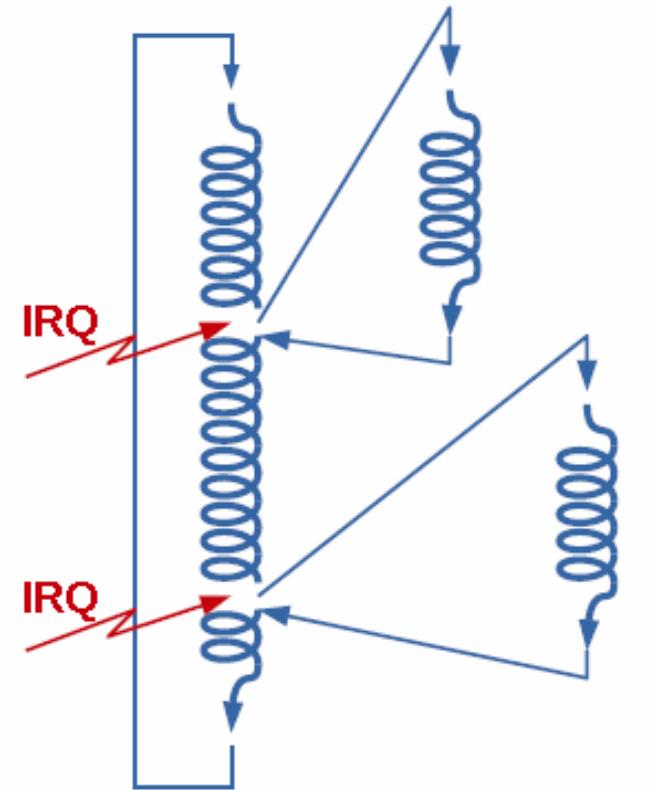
# Multitâches et Commutation

# Multitâches et Commutation

## Exécution de tâches Système monotâche



Superloop

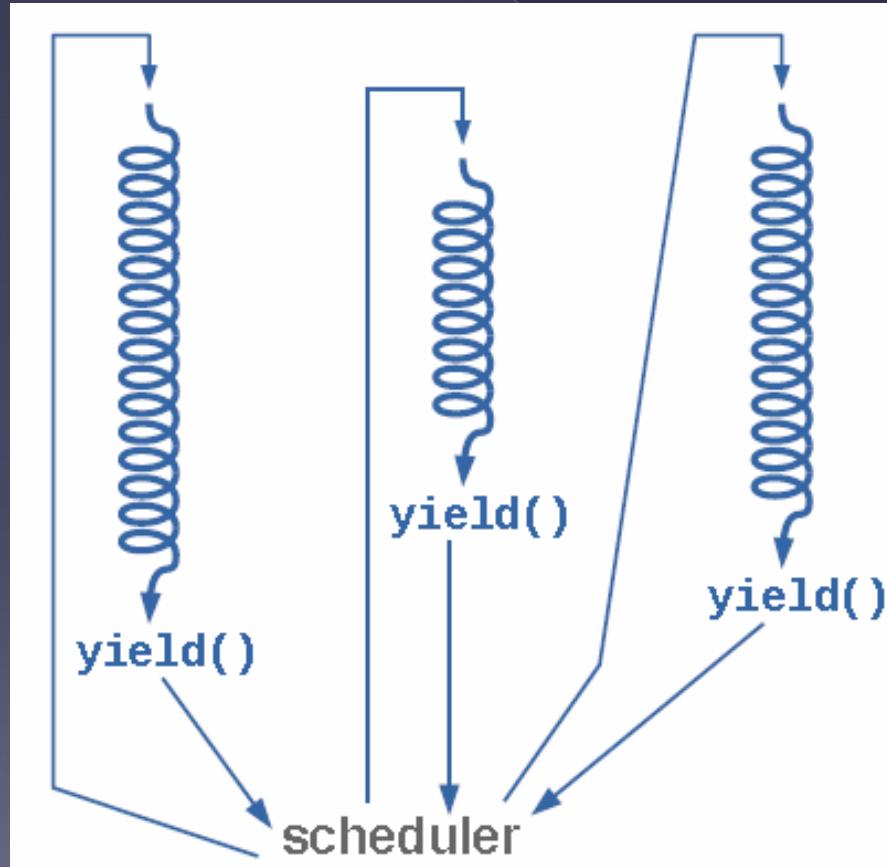


Superloop avec interruptions

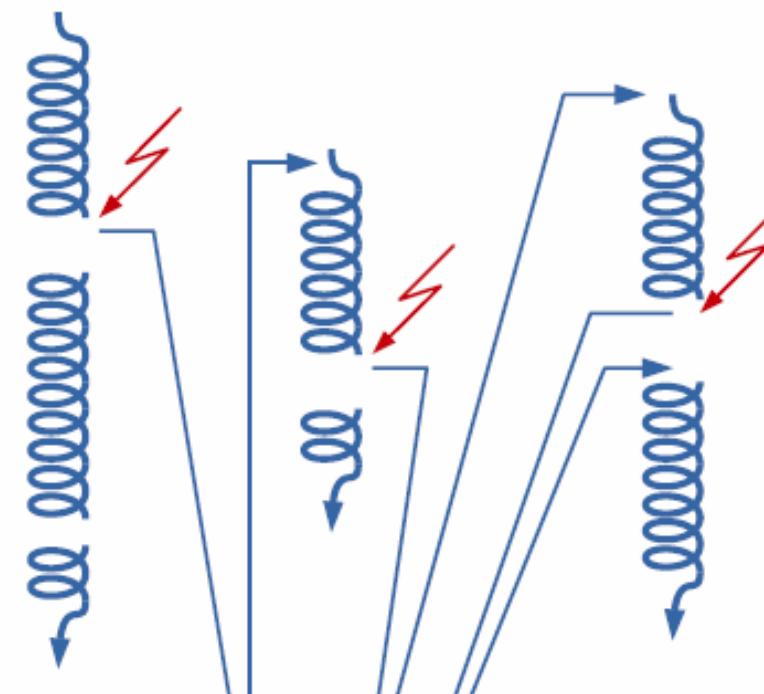
Fonctionnement typique d'un microcontrôleur / API

# Multitâches et Commutation

## Exécution de tâches Système multitâche



Coopératif



Préemptif

L'ordonnanceur (`scheduler`) est une fonctionnalité essentielle des systèmes OS pour exécuter des tâches sur un même processeur.

# Multitâches et Commutation

## Exécution de tâches

### L'ordonnancement

Tâche de sélection d'un processus en attente dans la liste des processus prêts et d'allocation de la CPU pour ce processus

Il existe plusieurs modes d'ordonnancement :

- **Temps partagé (time sharing system)** : comportement par défaut sur les O.S. comme Linux
- **Temps réel (realtime scheduling)** : suivant des algorithmes comme Round Robin ou Fifo basés sur des priorités entre tâches ou Earliest Deadline First utilisant des temps d'expiration des tâches.

# Généralités sur les OS

## Récapitulatif

Critères	Temps partagé	Temps réel
<b>But</b>	Maximiser la capacité de traitement (débit) & utilisation des ressources	Etre prévisible (garantie de temps de réponse)
<b>Temps de réponse</b>	Bon en moyenne	Bon dans le pire des cas / moyenne non importante
<b>Comportement à la charge</b>	Confortable à l'utilisateur	Stabilité et respect des contraintes de temps

# Multitâches et Commutation

## Processus & programme

- **Processus:** Actif & Dynamique
- **Programme:** Passif & Statique



Un processus correspond plus exactement à un espace indépendant de mémoire virtuelle, dans lequel un ou plusieurs threads s'exécutent.

# Multitâches et Commutation

## Processus

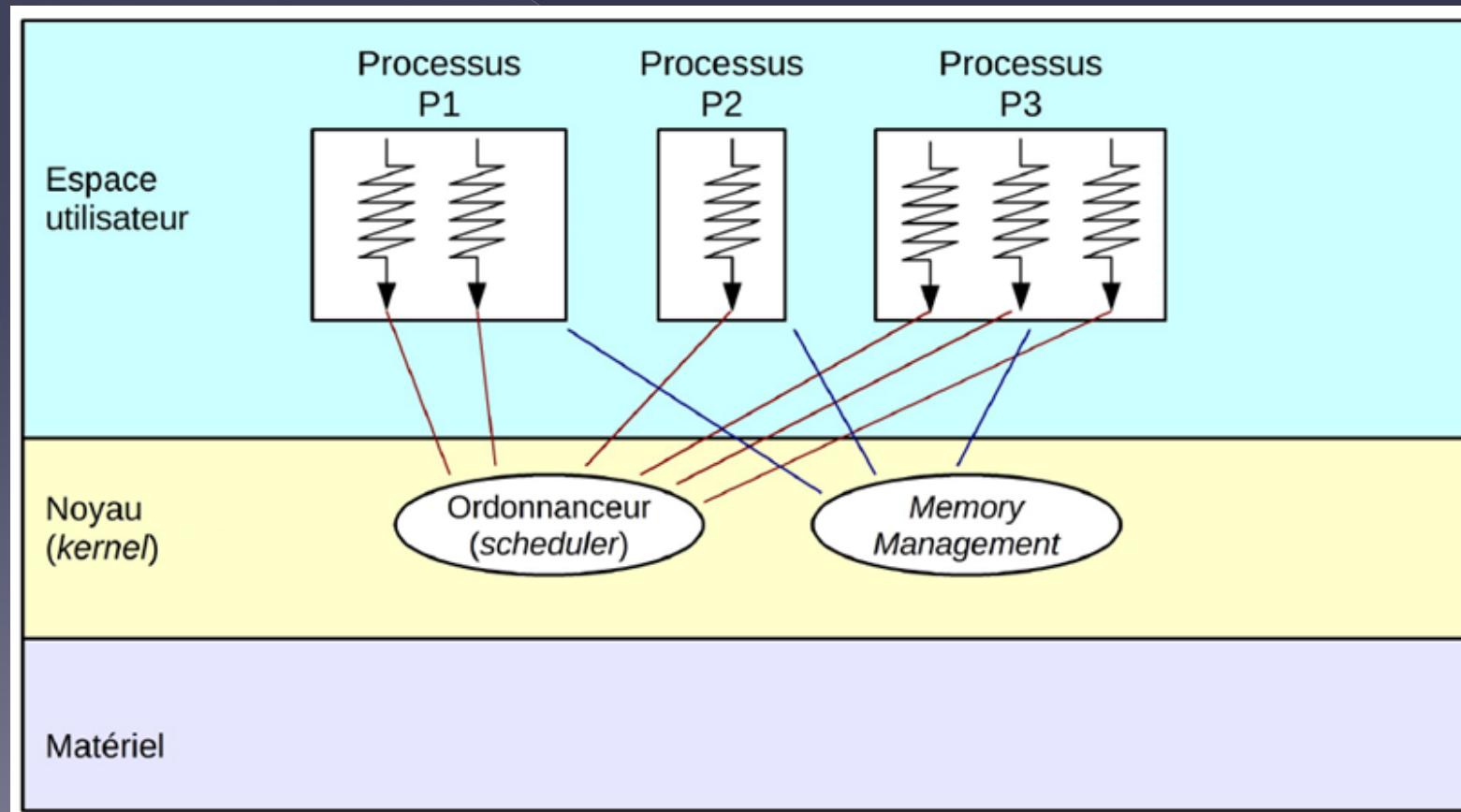
C'est un programme en cours d'exécution. Chaque processus possède :

- **Un espace d'adressage qui contient :**
  - le programme exécutable
  - ses données
  - sa pile
- **Un ensemble de registres dont :**
  - le compteur ordinal
  - le pointeur de pile
- **D'autres registres matériels et informations nécessaires.**

# Multitâches et Commutation

## Processus et threads

Les processus sont des espaces de mémoire disjoints, au sein desquels s'exécutent un ou plusieurs threads.

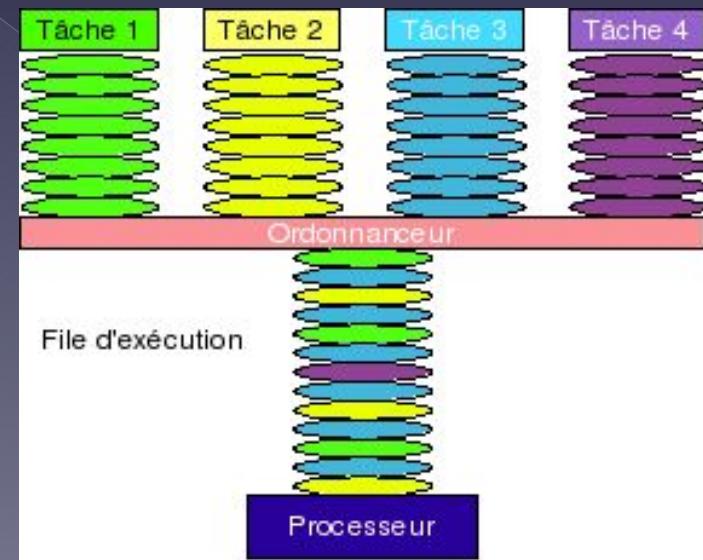
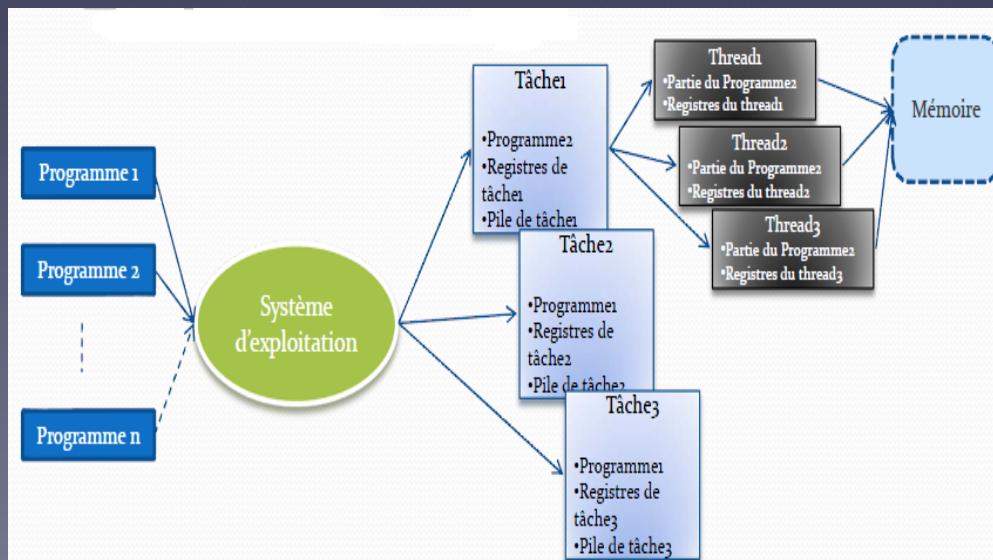


# Multitâches et Commutation

## Threads (OU processus léger OU fil d'exécution)

### - Les threads d'une même tâche:

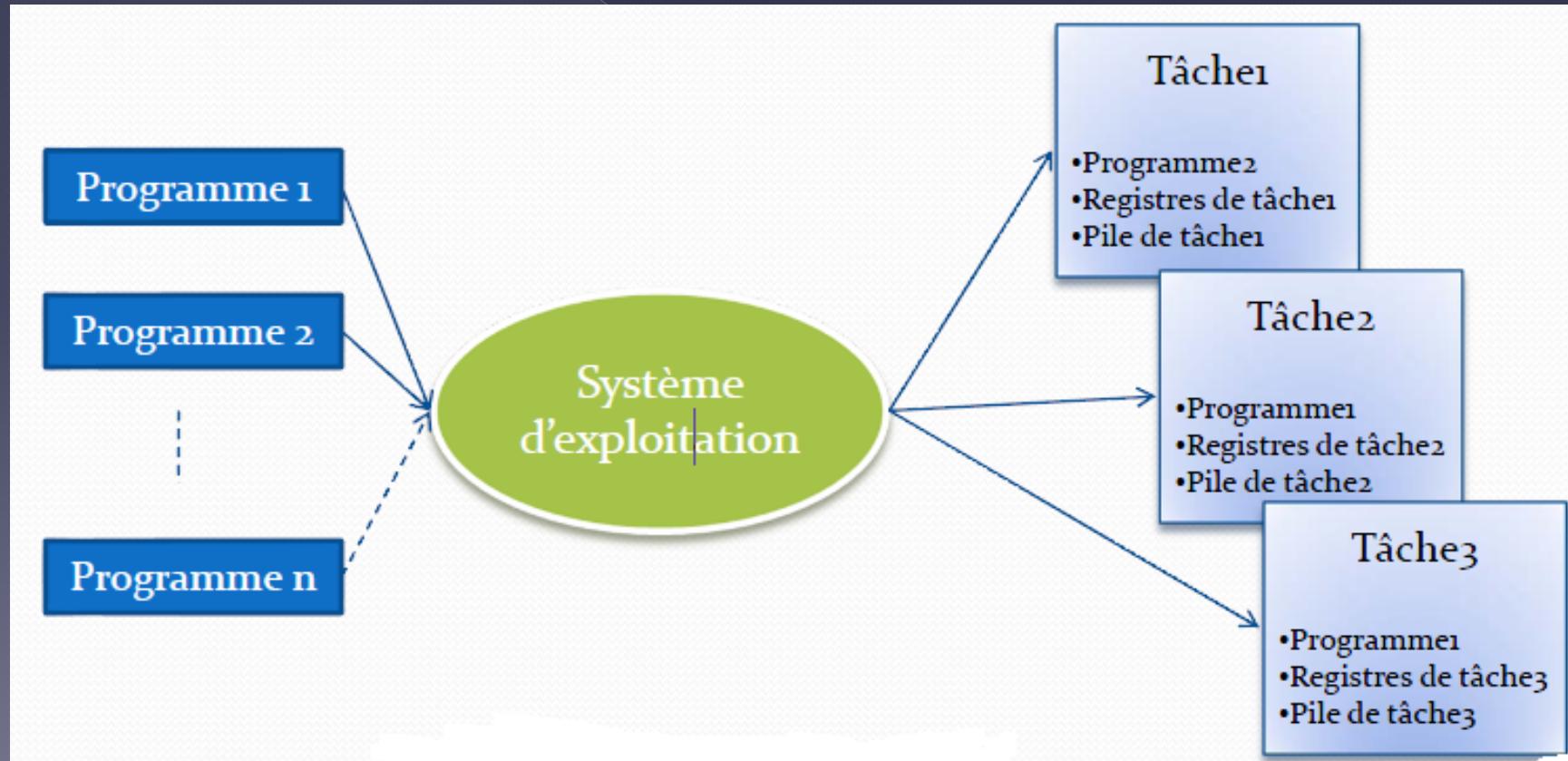
- Partagent: le même répertoire de travail, fichiers, dispositifs d'E/S, données globales, espace d'adressage, code du programme, etc.)
- Ne partagent pas: le compteur de programme, la pile, les information d'ordonnancement, etc.



# Multitâches et Commutation

## Mono tâche / Multitâche

- Mono tâche
- Multitâche: mono thread / multithread



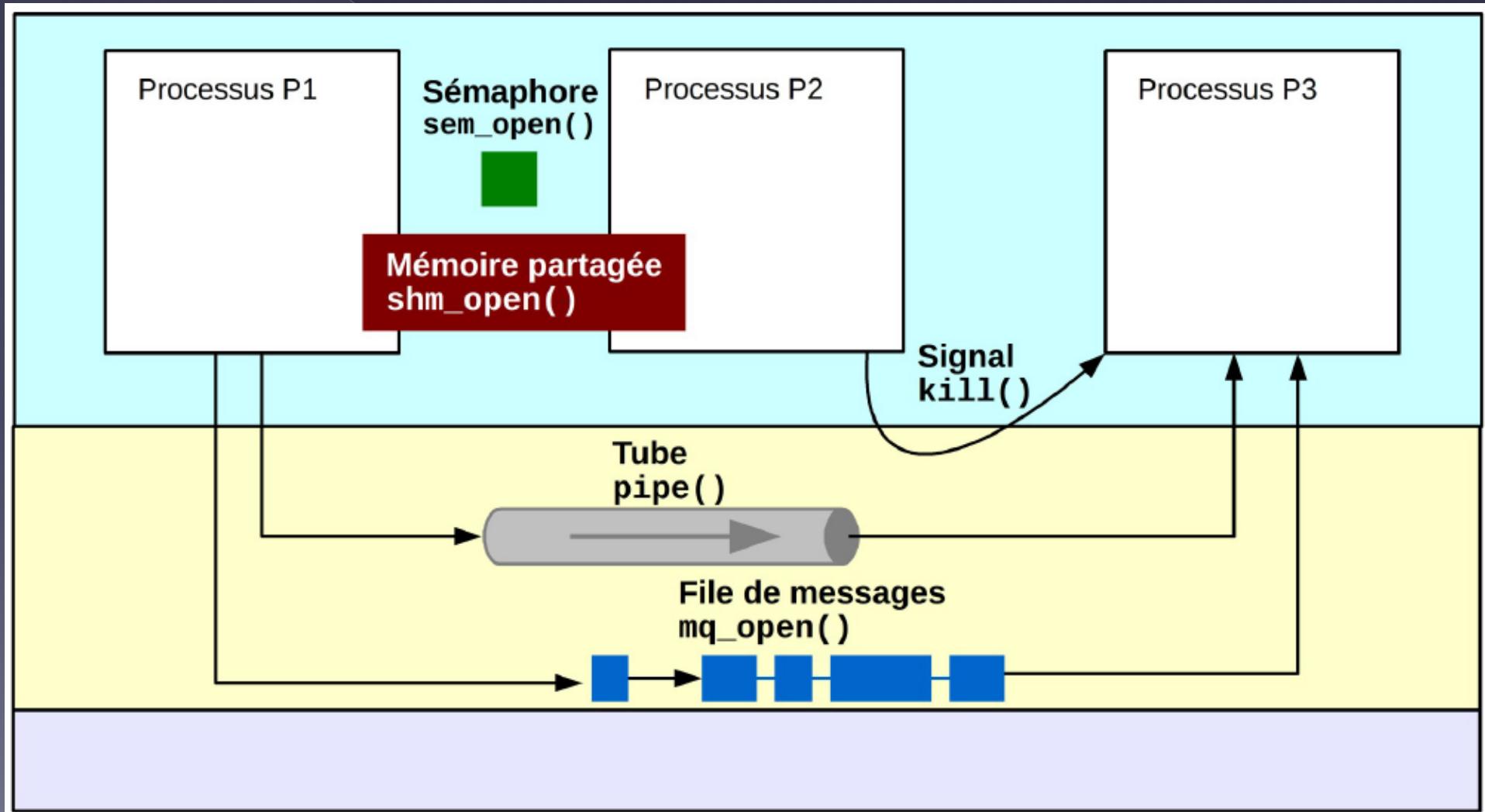
# Multitâches et Commutation

## Exemples

- DOS-C: OS embarqué mono tâche
- VxWorks (WindRiver): un unique type de thread/tâche  
→ chaque thread/tâche implémente un thread d'exécution
- Embedded Linux (TimeSys): 2 types de threads
  - Le forkLinux
  - Les tâches périodiques

# Multitâches et Commutation

## Communication inter processus (IPC)



# Multitâches et Commutation

## Création d'un nouveau processus

Evénements conduisant à la création d'un nouveau processus :

- Initialisation du système
- Exécution d'un appel système de création de processus par un processus en cours.
- Requête utilisateur sollicitant la création d'un nouveau processus
- Initiation d'un travail en traitement par lots

# Multitâches et Commutation

## Création de processus

La création d'un nouveau processus s'effectue via l'appel système **fork()**. Dans le processus père, fork() renvoie le PID (Process Identifier) du fils nouvellement créé, tandis que dans le fils, fork() renvoie toujours zéro.

- pid\_t fork (void)

**fork(), execve(), exit(), waitpid(),**

Un processus peut charger dans sa mémoire un nouveau code exécutable, abandonnant totalement son programme précédent pour le déroulement d'une nouvelle fonction main(). Ceci s'effectue avec l'une des fonctions de la famille **exec()**, dont seul **execve()** est réellement un appel système, les autres étant des fonctions de bibliothèques qui arrangeant leurs arguments avant de l'invoquer.

# Multitâches et Commutation

**Exemple: Interpréteur de commandes**

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

#define LG_LIGNE 256

int main (void)
{
    char ligne[LG_LIGNE];
    while(1) {
        // Afficher un symbole d'invite (prompt)
        fprintf(stderr, "--> ");
        // Lire une ligne de commandes
        if( fgets(ligne, LG_LIGNE, stdin) == NULL)
            break;
        //Supprimer le retour chariot final
        ligne[strlen(ligne)-1] = '\0';
        //Lancer un processus
        if (fork() == 0) {
            // --- PROCESSUS FILS ---
            // Exécuter la commande
            execlp(ligne, ligne, NULL);
            // Message d'erreur si on échoue
            perror(ligne);
            exit(EXIT_FAILURE);
        } else {
            // --- PROCESSUS PÈRE ---
            // Attendre la fin de son fils
            waitpid(-1, NULL, 0);
            // Et reprendre la boucle
        }
    }
    fprintf (stderr, "\n");
    return EXIT_SUCCESS;
}
```

# Multitâches et Commutation

## Parallélisme multithreads

Au cours des années 1980, plusieurs implémentations ont été proposées pour obtenir un mécanisme multitâche léger, fonctionnant à l'intérieur de l'espace mémoire d'un processus.

Dans les années 1990, une volonté d'uniformisation de l'API des systèmes Unix a donné naissance à la norme Posix, dont une section était consacrée aux **threads**. Cette série de fonction permet de gérer des **Posix Threads**, aussi appelés « **Pthreads** ».

# Multitâches et Commutation

## La Norme Posix

- Portable Operating System Interface [for Unix]
- POSIX est une famille de normes techniques définie depuis 1988 par IEEE.
- Uniformise les OS
- Première version publiée en 1988
- Souvent implémenté en partie

# Multitâches et Commutation

La création d'un nouveau thread s'obtient en appelant *pthread\_create()*.

**pthread\_t:** l'identifiant du thread

**pthread\_attr\_t:** les attributs du thread

Exemple :

```
Int pthread_create (pthread_t * thread,  
                    pthread_attr_t * attr,  
                    void * (*fonction) (void *),  
                    void * argument);
```

Dès que la fonction *pthread\_create()* se termine avec succès, nous savons qu'un nouveau fil d'exécution se déroule dans notre processus.

*Pthread\_exit ()* : déclare la fin du thread

Le pointeur renvoyé lors de la terminaison peut être récupéré par n'importe quel autre thread qui invoque *pthread\_join()*.

## Exemple

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

void * factorielle(void * arg)
{
    int i;
    int n = (int) arg;
    int resultat = 1;
    for (i= 2; i < n; i++) {
        resultat = resultat * i;
        fprintf(stderr, "%d! : en calcul ...\\n", n);
        sleep(1);
    }
    return (void *) resultat;
}
```

```
int main (int argc, char * argv[])
{
    pthread_t * threads = NULL;
    void * retour;
    int i;
    int n;
    // Vérification des arguments
    if (argc < 2) {
        fprintf(stderr, "usage: %s valeurs..\\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    // Allocation d'un tableau d'identifiants
    threads = calloc(argc-1, sizeof(pthread_t));
    if (threads == NULL) {
        perror("calloc");
        exit(EXIT_FAILURE);
    }
    fprintf(stderr, "main(): lancement des threads\\n");
    for (i = 1; i < argc; i++) {
        // Récupération de l'arguments numérique
        if (sscanf(argv[i], "%d", & n) != 1) {
            fprintf(stderr, "%s: invalide\\n", argv[i]);
            exit(EXIT_FAILURE);
        }
    }
}
```

```
// Lancement du thread
if (pthread_create(& (threads[i-1]), NULL, factorielle, (void *) n) != 0) {
    fprintf(stderr, "Impossible de demarrer le thread %d\n", i);
    exit(EXIT_FAILURE);
}
}
fprintf(stderr, "main(): tous threads lances \n");
for (i = 1; i < argc; i++) {
// Attente du thread
    pthread_join(threads[i-1], & retour);
    fprintf(stderr, "main(): %s! = %d\n",
            argv[i], (int) retour);
}
fprintf(stderr, "main(): tous threads termimes \n");
free(threads);
return EXIT_SUCCESS;
}
```

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

void * fonction_thread(void * arg);
typedef struct {
    int X;
    int Y;
} coordonnee_t;

int main (void)
{
pthread_t thr;
coordonnee_t * coord;
coord = malloc(sizeof(coordonnee_t));
if (coord == NULL) {
    perror("malloc");
    exit(EXIT_FAILURE);
}
coord->X=10;
coord->Y=20;
if (pthread_create(& thr, NULL,
                  fonction_thread, coord) != 0) {
fprintf(stderr, "Erreur dans
pthread_create\n");
exit(EXIT_FAILURE);
}
while (1) {
    fprintf(stderr, "Thread Main\n");
    sleep(1);
}
}
```

```
void * fonction_thread(void * arg)
{
    coordonnee_t * coord = (coordonnee_t *) arg;
    int X = coord->X;
    int Y = coord->Y;
    free(coord);
    while (1) {
        fprintf(stderr, "Thread X=%d, Y=%d\n", X, Y);
        sleep(1);
    }
}
```

# Multitâches et Commutation

## Systèmes multiprocesseurs

### Multiprocesseurs, multicœurs et hyperthreading

Le noyau Linux considère qu'il existe deux types de systèmes: les machines uniprocesseur sur lesquelles un seul fil d'exécution est présent à un moment donné, et les machines multiprocesseurs symétriques (SMP) qui permettent l'exécution parallèle de plusieurs tâches.

- **Multiprocesseur** contenant réellement plusieurs processeurs physiques distincts;
- **Multicoeur:** un seul processeur est présent, mais il dispose de plusieurs décodeurs d'instructions travaillant en parallèle.
- **Hyperthreading :** un seul processeur assure une commutation entre deux séquences d'instructions différentes.

# Multitâches et Commutation

## Exemple

La fonction **long sysconf (int nom)** avec l'argument **\_SC\_NPROCESSORS\_ONLN** permet de savoir le nombre de CPU de votre ordinateur sous Linux.

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

int main (void)
{
    printf("Nombre de CPU: %ld\n",
           sysconf(_SC_NPROCESSORS_ONLN));
    return EXIT_SUCCESS;
}
```

# Multitâches et Commutation

## Affinité d'une tâche

L'ordonnanceur de Linux est capable de placer (et de déplacer) des tâches sur les processeurs virtuels du système.

### Exemple :

Utilisation de l'utilitaire **GkrellM** (Gnome Krell Monitor)

Nous pouvons vérifier à tout moment le processeur sur lequel s'exécute une tâche avec la fonction (spécifique GNU):

```
int sched_getcpu (void) ;
```

Qui nous renvoie le numéro de processeur depuis lequel elle a été invoquée, ou -1 si elle ne peut le déterminer

## Exemple

```
#define _GNU_SOURCE
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
int main (void)
{
    Int n;
    Int precedent = -1;
    Time_t heure;
    Struct tm * tm_heure;
    While (1){
        n=sched_getcpu();
        if (n== -1){
            perror(''sched_getcpu'');
            Exit(EXIT_FAILURE);
        }
        if(precedent == -1)
            precedent = n;
        if(n !=precedent) {
            heure = time(NULL);
            tm_heure = localtime( & heure);
            print(`%02d:%02d:%02d migration %d -> %d\n` ,
                  tm_heure->tm_hour,    tm_heure->tm_min,
                  tm_heure ->tm_sec, precedent, n);
            Precedent = n;
        }
    }
    Return EXIT_SUCCESS;
}
```

# Multitâches et Commutation

**L'affinité d'une tâche** est la liste des CPU sur lesquels elle peut s'exécuter. On peut la consulter ou la fixer à l'aide des fonctions suivantes:

```
int sched_setaffinity (pid_t pid,  
                      size_t taille,  
                      const cpu_set_t * cpuset);
```

```
int sched_getaffinity (pid_t pid,  
                      size_t taille,  
                      cpu_set_t * cpuset);
```

```
void CPU_ZERO (cpu_set_t * ensemble);  
void CPU_SET (int cpu, cpu_set_t * ensemble);  
void CPU_CLR (int cpu, cpu_set_t * ensemble);  
void CPU_ISSET (int cpu, cpu_set_t * ensemble);
```

# Exemple

```
#define GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sched.h>

int main(int argc, char *argv[])
{
    cpu_set_t cpuset;
    int cpu;
    // Lire le numéro de CPU dans le premier argument
    if ((argc != 2) || (sscanf(argv[1], "%d", &cpu) != 1)) {
        fprintf(stderr, "usage: %s cpu\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    // Remplir l'ensemble avec le CPU indiqué
    CPU_ZERO(&cpuset);
    CPU_SET(cpu, &cpuset);
    // Fixer l'affinité
    if (sched_setaffinity(0, sizeof(cpuset), &cpuset) != 0)
    {
        perror(argv[1]);
        exit(EXIT_FAILURE);
    }
    while (1) {
        printf(" Je suis sur le CPU %d\n",
               sched_getcpu());
        sleep(1);
    }
    return EXIT_SUCCESS;
}
```

Ce programme permet l'accrochage d'un processus sur un CPU donné en argument. On notera que la valeur 0 en premier argument de `sched_setaffinity()` indique que l'on fixe le masque d'affinité pour le processus appelant

# Multitâches et Commutation

On peut fixer – et lire- l'affinité d'un thread dynamiquement avec:

```
int pthread_setaffinity_np (pthread_t thread,  
                           size_t      taille,  
                           const cpu_set_t * cpuset);  
  
int pthread_getaffinity_np (pthread_t thread,  
                           size_t      taille,  
                           cpu_set_t * cpuset);
```

On peut aussi fixer l'affinité d'un futur thread avant sa création.

```
int pthread_attr_setaffinity_np (pthread_attr_t thread,  
                                 size_t      taille,  
                                 const cpu_set_t * cpuset);  
  
int pthread_attr_getaffinity_np (pthread_attr_t thread,  
                                 size_t      taille,  
                                 cpu_set_t * cpuset);
```

# Multitâches et Commutation

## Exemple

Créer un programme qui lance autant de threads en parallèle qu'il y a de processeurs disponibles.

Nous commençons par le CPU 0, puis incrémentons le numéro jusqu'à ce que l'appel *pthread\_create()* échoue.

## Exemple

```
#define _GNU_SOURCE
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sched.h>

void * fonction(void * arg)
{
    int num = (int) arg;
    while(1) {
        printf(``[Thread %d] Je suis sur le CPU %d\n`` , num, sched_getcpu());
        Sleep(5);
    }
}
```

```
int main (void){  
cpu_set_t cpuset;  
pthread_t thr;  
pthread_attr_t attr;  
int i;  
i =0;  
while(1)  
{// initialiser avec les attributs par défaut  
pthread_attr_init( & attr);  
//Préparer le cpuset  
CPU_ZERO(&(cpuset));  
CPU_SET(i, &(cpuset));  
//Fixer l'affinité  
pthread_attr_setaffinity_np(& attr, sizeof(cpu_set_t), & cpuset);  
//Lancer le thread  
if (pthread_create(& thr, & attr, fonction, (void *) i) !=0)  
    Break;  
i++;  
}  
//Terminer le thread main en continuant les autres  
pthread_exit(NULL);  
}
```

# Multitâches et Commutation

Partage d'espace mémoire

# Multitâches et Commutation

## Partage d'espace mémoire

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
void * fonction_thread(void * arg);
#define NB_THREADS 5
int compteur = 0;
int main (void)
{
pthread_t thr[NB_THREADS];
long n;
for (n = 0; n < NB_THREADS; n++) {
if (pthread_create(&thr[n], NULL, fonction_thread, (void *) n) != 0) {
fprintf(stderr, "Erreur dans pthread_create\n");
exit(EXIT_FAILURE);
}
}
while (1) {
fprintf(stderr, "Thread Main, compteur = %d\n", compteur); sleep(1);
}
}
```

```
void * fonction_thread(void * arg)
{
long num = (long) arg;
while (1) {
fprintf(stderr, "Thread numero %ld,
compteur = %d \n", num+1, compteur);
compteur++;
sleep(1);
}
}
```

# Multitâches et Commutation

## Synchronisation entre threads - **MUTEX-**

- Un mutex ne peut être tenu que par un seul thread à la fois.
- Il existe deux fonctions de manipulation des mutex: une fonction de verrouillage et une fonction de libération.

```
int pthread_mutex_lock (pthread_mutex_t * mutex);  
int pthread_mutex_unlock (pthread_mutex_t * mutex);  
int pthread_mutex_trylock (pthread_mutex_t * mutex);
```

- On peut initialiser un mutex de manière statique ou dynamique, en précisant certains attributs à l'aide d'un objet de type `pthread_mutexattr_t`

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;  
int pthread_mutex_init (      pthread_mutex_t * mutex,  
                          const pthread_mutexattr_t * attributs);
```

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
    static void * routine_threads (void * argument);
    static int aleatoire (int maximum);
pthread_mutex_t mutex_stdout = PTHREAD_MUTEX_INITIALIZER;
int main (void)
{
    int i;
    pthread_t thread;
    for (i = 0; i < 5; i++)
        pthread_create(& thread, NULL, routine_threads, (void *) i);
    pthread_exit(NULL);
}
static void * routine_threads (void * argument)
{
    int numero = (int) argument;
    int nombre_iterations;
    int i;
    nombre_iterations = 1 + aleatoire(3);
```

```
for (i = 0; i < nombre_iterations; i++) {
    sleep(aleatoire(3));
    pthread_mutex_lock(& mutex_stdout);
    fprintf(stdout, "Le thread %d a obtenu le mutex\n", numero);
    sleep(aleatoire(3));
    fprintf(stdout, "Le thread %d relache le mutex\n", numero);
    pthread_mutex_unlock(& mutex_stdout);
}
return NULL;
}
static int aleatoire (int maximum)
{
    double d;
    d = (double) maximum * rand();
    d = d / (RAND_MAX + 1.0);
    return ((int) d);
}
```

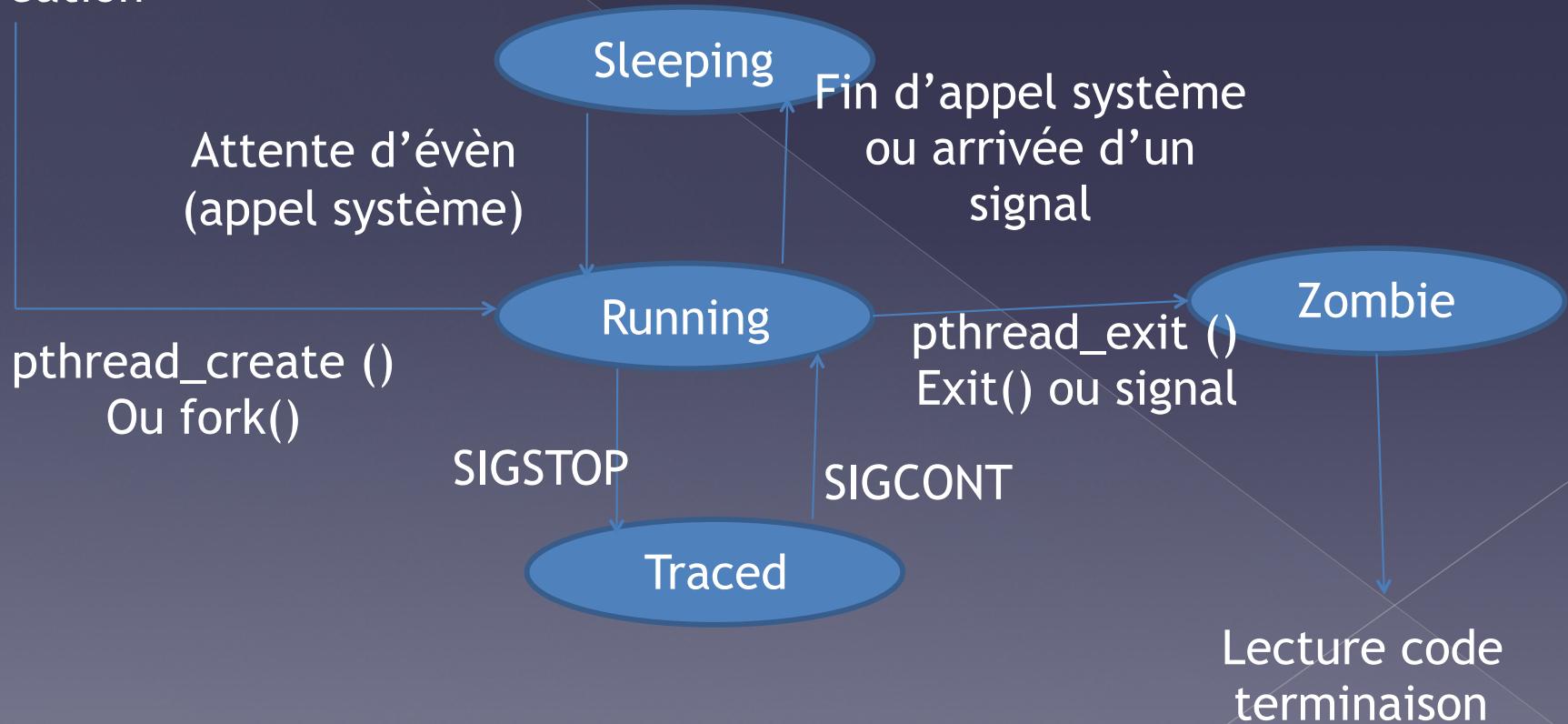
# Multitâches et Commutation

## Etats des tâches

**ps aux,**

**ps maux:** décrit l'activité des différents threads au sein de chaque processus

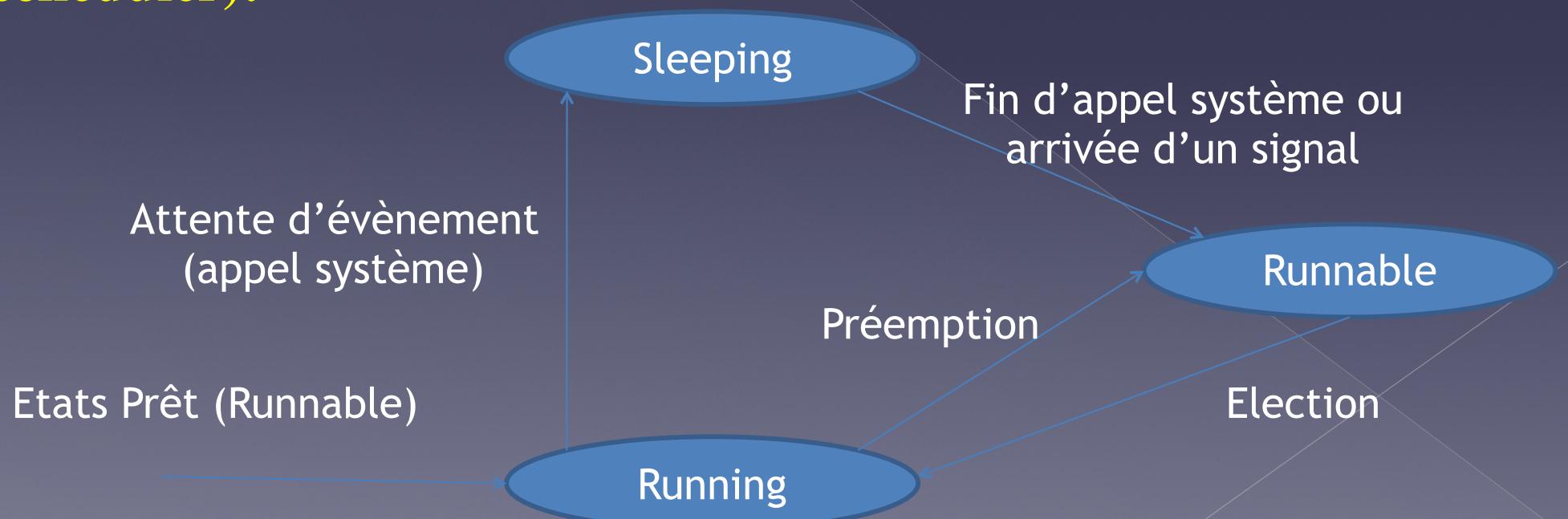
Création



# Multitâches et Commutation

## Ordonnancement

Lorsque nous disposons de 4 CPU, par exemple, il peut y avoir jusqu'à 4 tâches actives (**Running**) simultanément. Mais, sur la plupart des systèmes, il arrive que le nombre des tâches demandé excède le nombre de processeurs disponibles. Il faut donc organiser des commutations entre ces tâches, rôle de **l'ordonnanceur (scheduler)**.



# Multitâches et Commutation

## Préemption

Lorsqu'une tâche s'est exécutée pendant un temps prolongé au détriment des autres, le système peut décider de lui retirer temporairement le processeur et de la réinjecter dans la liste des tâches Runnable. On dit que la tâche a été **préemptée**, et plus généralement qu'il s'agit d'un système multitâche **préemptif**.

À l'inverse, un ordonnancement peut être collaboratif, c'est-à-dire que chaque tâche va volontairement céder régulièrement le CPU - quitte à le retrouver immédiatement si elle est la seule en activité