

# *SYSTEME D'EXPLOITATION POUR L'EMBARQUE ET LE TEMPS RÉEL*

Ecole Centrale de Lille,  
CS 20048, F59651 Villeneuve d'Ascq Cedex  
Tél : (33)(0)3 20 33 54 56, (33)(0)6 80 08 48 64  
E-mail : [samir.elkhattabi@centralelille.fr](mailto:samir.elkhattabi@centralelille.fr)

# *Qu'est ce qu'un système embarqué*

---

Un système embarqué, composé d'une partie **électronique** (matériel) et d'une partie **informatique**, est dédié à une tâche spécialisée et ce de manière autonome.

Il est généralement **enfoui** :

- L'encombrement est réduit et donc les ressources matérielles le sont aussi,
- la consommation énergétique doit être réduite

Il est souvent **temps-réel**.

Et il doivent être **sûrs de fonctionnement**.

# *Propos du support*

---

Ici, on s'intéresse uniquement aux architecture à base de **micro-processeur** (les architectures à base de microcontrôleur ne sont pas étudiées).

Ainsi, le système électronique peut être qualifié de calculateur et donc doté d'un **système d'exploitation multitâches** et si nécessaire temps réel.

Ici, on s'intéresse uniquement à la **programmation système** en vue de la conception d'applications embarquées (la génération d'une solution système à embarquer fait l'objet d'un autre support).

Vues les contraintes d'un système embarqué, on utilisera une plateforme **UNIX** et une implémentation **POSIX**.

# *Calculateur / Système d'exploitation*

---

Tout calculateur est composé d'un **processeur**, d'une **mémoire** centrale et de **périphériques** tels que les disques, les interfaces d'interaction, des interfaces réseau et d'autres périphériques d'E/S.

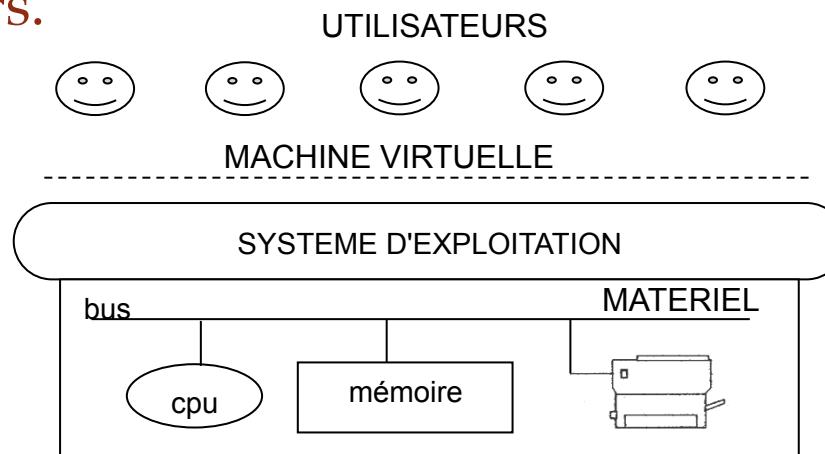
Un système d'exploitation (OS pour Operating System) est un ensemble de programmes qui **prend en charge** tous ces composants matériel, qui les **utilise correctement** et ce de manière optimale en fournissant à l'utilisateur des interfaces simplifiées :

- Interface d'interaction utilisateur / administrateur : environnement texte et / ou graphique,
- Interface de programmation : Librairies d'appels système,
- Interface de communication : échange de données à travers un réseau,

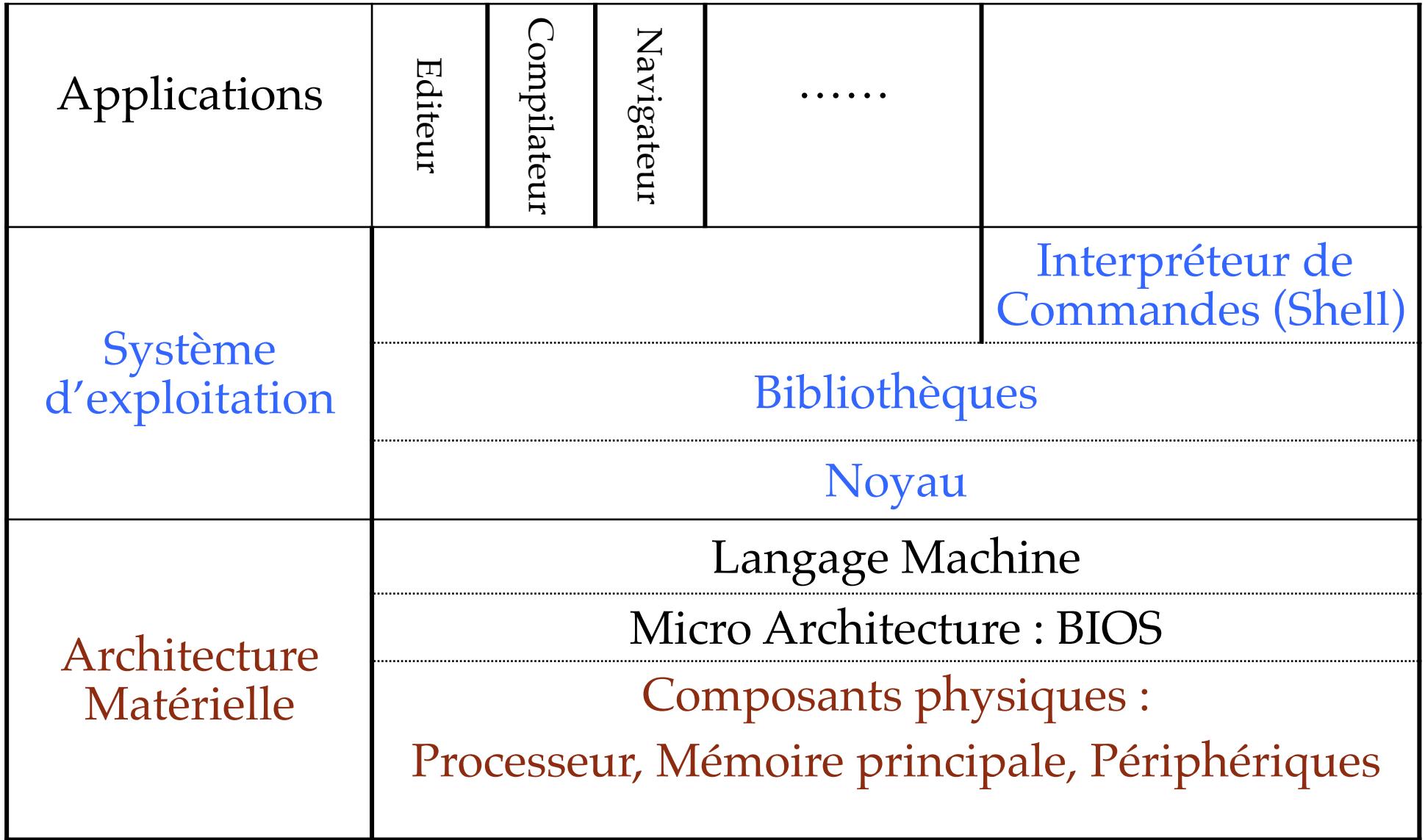
# Fonctions d'un OS

Le système d'exploitation est un ensemble de programmes qui réalise l'interface entre le matériel de l'ordinateur et les utilisateurs. Il a deux objectifs principaux :

- Construction au dessus du matériel d'une machine virtuelle avec un langage étendu plus facile d'emploi et conviviale,
- Prise en charge de la gestion de plus en plus complexe des ressources et partage de celles-ci entre des tâches et/ou des utilisateurs.



# *Système Informatique (1)*



# *Système Informatique (2) : Matériel*

---

Le plus bas niveau est du ressort de l'électronique, il est composé de circuits intégrés, de connectique, d'alimentation en tension, ...

La **micro-architecture** regroupe les composants en unités fonctionnelles comme la CPU qui exécute un programme chargé à une adresse mémoire donnée.

Le traitement effectué par le CPU est contrôlé par un **micropogramme** ou par des circuits matériels.

Les instructions sont exécutées en un ou plusieurs **cycles d'horloge** et sont écrites en assembleur : **Instruction Set Architecture** (ISA) appelé également langage machine.

Le langage machine est un jeu d'instructions assez réduit : de 50 à 300 instructions. Il s'agit principalement de réaliser des opérations arithmétiques et logiques ainsi que des opérations de déplacement de données (lecture/écriture).

# *Système informatique (3) : Noyau*

---

Le noyau est la composante principale d'un OS. Il est répondu à plusieurs besoins :

- Contrôler le « multitâche » par un **ordonnanceur**.
- Gérer les ressources matérielles par des **pilotes** dits « matériel ».
- Mettre en place des pilotes dits « logiciel » : TCP/IP, ...
- Partager avec protection les ressources aussi bien matérielles que logicielles par des mécanismes de **verrous**.
- Contrôler les **droits d'accès**.

Pour un noyau temps-réel, on ajoutera la propriété de **préemption**.

## *Système Informatique (4) : Bibliothèques*

---

Les périphériques sont contrôlés par des **registres de périphérique**. On peut demander par exemple :

- une lecture à un disque,
- en chargeant son adresse avec une adresse mémoire, un nombre d'octets, et le type de l'opération (lire / écrire) dans son registre.
- d'autres paramètres sont généralement nécessaires.
- un mot d'état est renvoyé à la fin d'opération.

Les bibliothèques d'un OS permettent de **masquer** cette complexité en fournissant au développeur un jeu d'instructions plus évolué et plus pratique (**langage de programmation**) . Pour notre exemple, il s'agira de lire des blocs au lieu de se préoccuper du déplacement des têtes de lecture, de la gestion de la latence, ...

# *Système Informatique (5)*

---

Au dessus de l'OS, on trouve les interfaces d'interaction (**Shell** et/ou système de fenêtrage) et les interfaces de programmation (**compilateurs**) qui permettent de créer des applications utilisateurs qui masquent à leur tour les langages de programmation.

L'OS est un programme qui fonctionne en mode **noyau** : l'OS est **protégé** contre les modifications de l'utilisateur par des mécanismes de **protection matérielles**.

Un utilisateur peut écrire son propre compilateur s'il le souhaite par contre il ne peut écrire son propre gestionnaire d'interruptions qui est protégé contre toute modification.

NB : vous trouverez en annexe un historique qui peut donner certaines indications.

# ***UNICS(X)***

---

En 1969, les laboratoires Bell se retire du projet MULTICS et Ken Thompson eut l'idée de réécrire une version simplifiée de MULTICS sur un mini-ordinateur de l'époque (un PDP-7) :

- Brian Kernighan l'appela par dérision **UNICS**.

Denis Ritchie rejoint le projet avec son département :

- Portage d'UNIX sur un PDP-11/20, **PDP-11/45, PDP-11/70** (machines largement utilisées).
- Dispositifs matériels de protection mémoire : plusieurs utilisateurs simultanés.

# *UNIX & C*

---

Thompson décide de développer un langage de haut niveau pour faciliter la réécriture du système, qu'il a appelé B (version simplifiée de CPL) :

- Manque de structures.

Ritchie développe alors le langage C.

Thompson et Ritchie ont réécrit UNIX en C : Publication décrivant UNIX en 1974 (ils obtiennent en 1984 le prix Turing de l'ACM).

# *Expansion de UNIX*

---

Bell distribue des licences aux universitaires moyennant une modique contrepartie :

- Les universitaires étaient équipés de PDP mais avec de mauvais systèmes d'exploitation.
- UNIX offrait un environnement ouvert avec un source modifiable.

Les séminaires de recherche et de développement sur UNIX se multiplient :

- Amélioration du système.
- Diffusion rapide de nouvelles idées.
- Version 6 (6° édition du manuel) puis (quelques années plus tard) version 7.

# *Les branches UNIX : AT&T et BSD*

---

En 1984, AT&T est libéralisée et commercialise sa première version d'UNIX appelée System III :

- Un an plus tard, **System V** était disponible (System IV est un mystère).
- Release 2, Release 3, Release 4 : complexité croissante.

L'université de Berkeley aidée par les subventions de la DARPA a apporté de nombreuses modifications :

- Une première version fut appelée **1BSD**, suivie de **2BSD** sur PDP-11.
- **4BSD** développé pour le **VAX** a fait le succès de cette version à l'inverse de AT&T qui offrait une version **7 grossie**.

Améliorations :

- Mémoire virtuelle & pagination.
- Optimisation du système de fichiers.
- Fiabilisation des signaux.
- Communication entre machines : TCP / IP s'impose comme standard (#OSI).
- Utilitaires & compilateurs : vi, csh, Lisp, Pascal.

# *Portabilité d'UNIX*



Le premier portage vers une architecture différente (Interdata 8/32) a permis de mettre en évidence l'insuffisance des hypothèses initiales (un entier sur 16 bits, un pointeur est codé sur 16 bits  $\Rightarrow$  taille de programme limitée à 64 K).

Depuis UNIX a été portée sur une grande variété d'ordinateurs.

# *Portabilité d'UNIX*

---

Le portage nécessite de :

- Disposer d'un compilateur C sur la machine cible.
- Disposer des pilotes pour les périphériques d'E/S (C et assembleur).
- Ecrire un gestionnaire d'interruption en langage machine.
- Ecrire les routines de gestion de la mémoire.

Le portage a été facilité par l'écriture d'une version C portable par Steve Johnson.

Le portage et l'installation d'UNIX sur une nouvelle machine est devenu chose facile.

# *Noyau UNIX & portabilité*

---

Le noyau UNIX est composé de deux parties essentielles.

Un noyau dépendant de la machine : écrit en C mais doit être réécrit en assembleur en cas de portage sur une nouvelle machine.

- gestionnaire d'interruptions,
- pilotes d'E/S,
- une partie du logiciel de gestion mémoire.

Noyau indépendant de la machine (il est le même sur toutes les machines) : entièrement écrit en C.

- gestion des appels systèmes,
- gestion des processus, ordonnanceur,
- pagination, swapping,
- système de fichiers, tubes,

# *Standard UNIX*

---

Début 80, chaque constructeur propose sa version d'UNIX généralement basée sur 4.3BSD et rarement sur System V R3 (malgré la publication de System V Interface Definition) avec une incompatibilité dans le binaire (pas de norme initiale).

Regroupement constructeurs :

- Open Software Foundation (intégrer la norme POSIX, X11) :
  - ▶ Constitué de : IBM, Digital, HP
  - ▶ Travaux de spécifications en vue de la standardisation
  - ▶ OSF/1 V2.0 de Digital : Digital Unix.
- UI (Unix International) : Sun, AT&T, ...
- X/OPEN : association européenne (Guide XPG III).

# POSIX

---

Norme IEEE (ensuite ISO) pour la normalisation des appels systèmes : Portable Operating System IX (pour UNIX)

- Un bon compromis : on a procédé par intersection et les différences ont été ignorées. Résultat : version 7 + gestion des signaux de BSD + gestion des terminaux

Normes POSIX :

- POSIX 1003.1(a) : création de processus et exécution, gestions des I/O et fichiers
- POSIX 1003.1b : communication interprocessus, allocation du CPU, gestion de la mémoire et Asynchronisme
  - ▶ (P1003.1b : correspond à P1003.4/D14)
- POSIX 1003.1c : multithreads
  - ▶ POSIX 1003.4a/D4 : (i.e. Draft) non publié
- ....

# *Pourquoi POSIX ?*

---

**Normalisé par Institute of Electrical and Electronic Engineers (IEEE) et adopté par International Organization for Standardization (ISO) and the International Electrotechnical Commision (IEC)**

Solution portable non seulement sur des machines UNIX mais également sous Windows NT.

**POSIX SPECIFIE L'INTERFACE AVEC L'OS  
MAIS N'EST PAS L'OS  
POSIX N'EST PAS UNE IMPLEMENTATION**

# *Objectifs du cours*

---

Faire une programmation avancée mettant en jeu :

- des ressources à partager
  - ▶ mémoire, CPU, I/O
- des processus concourants & multithreading
- du synchronisme, de l'asynchronisme
- des algorithmes d'allocation du CPU
  - ▶ FIFO, Tourniquet
- Horloges et Temporiseurs (seconde et nanoseconde)
- Communication et Synchronisation interprocessus
  - ▶ mémoire partagée - signaux - sémaphores
  - ▶ Messages

Et du TEMPS REEL ...

# *Mise en œuvre*

## Comment : Mise en œuvre en C

- **Insertion de la librairie standard POSIX dans votre code**
  - ▶ `#include <unistd.h>`
- **Compilation :**
  - ▶ `-L` pour indiquer le chemin d'accès à une librairie
  - ▶ `-l` pour inclure une librairie
  - ▶ `-lpthread` : librairie des threads et sémaphores
  - ▶ `-lrt` : librairie temps-réel
  - ▶ `-non_shared` car pas de partage en temps réel
- **Exemple :**
  - ▶ `$ gcc -non_shared myprg.c -o myprg -L/usr/css/lib -lrt -lpthread`
  - ▶ `$ gcc myprg.c -o myprg -lpthread`

## Documents :

- manuels on line
  - ▶ `$ man 2 appel-système`
  - ▶ Exemple : `$ man 2 fork`

# Gestion des erreurs

---

L'appel d'une fonction système POSIX retourne une valeur positive en cas de succès et -1 en cas d'échec.

Dans ce dernier cas, la variable **errno** est positionnée avec le numéro d'erreur.

## Utilisation de **perror()**:

- **perror()** affiche un message relatif à **errno** qui est utilisé comme index dans une table de messages d'erreurs.
- **perror()** affiche (aussi) le message passé en argument pour personnaliser le message d'erreur.

On utilisera donc **systématiquement** une macro fonction pour vérifier tout appel système :

```
#define CHECK(sts,msg) if ((sts) == -1) {perror(msg); exit(-1); }
```

# Aide sous UNIX

---

Le manuel est divisé en huit sections (au moins) :

- 1 : Les commandes utilisateurs
- 2 : **Les appels système**
- 3 : **La librairie des sous-routines**
- 4 : Les formats de fichiers
- 5 : Les fichiers spéciaux
- 7 : **Les composantes système**
- 8 ou 1m : Les commandes d'administration système
- 9 : glossaire

Affichage paginé (more) des informations concernant une commande (ou toutes les commandes concernant un mot clé) :

**man [n] <commande>** n représente un numéro de section

**man -k <mot clé>** affiche une liste de commandes pour lesquelles le mot clé recherché apparaît dans le résumé

# *Commandes UNIX à connaître*

---

Manipulation de fichiers :

- **ls, cd, pwd, mkdir, cd, cp, rm, mv, ln, more, cat, find, tee, ...**

Manipulation des permissions d'accès :

- **chmod, chown, chgrp, umask, newgrp, groups, ...**

Gestion de processus :

- **ps, pstree, kill, top, ...**

Filtres :

- **grep, sort, tail, cut, paste, tee, od, tr, pr, ...**

Expressions régulières :

- **grep, tr, sed, awk**

Compilateurs & outils de développement :

- **make, nm, time, prof, lint, ld, ar, svn, sccs, dbx ou gdb, ...**

Développement de langage (pas pour tous) :

- **lex/flex, yacc/bison**

# *PARTIE 1 :*

## *PROCESSUS & ORDONNANCEMENT*

# *Multiprogrammation*

---

Un ordinateur peut exécuter plusieurs tâches à la fois :

- Exécution d'un programme utilisateur
- Lecture de données sur un disque
- Afficher les résultats sur un terminal
- ....

Un OS multiprogrammé **commute** d'une tâche à une autre en exécutant chaque tâche pendant quelques dizaines ou centaines de ms (notion de **quota CPU**).

A un instant donné, une seule tâche est exécutée : **faux ou pseudo-parallélisme**.

Problématique : contrôler plusieurs activités en parallèle

**Modèle de parallélisme avec partage de ressources communes (besoin de mécanismes de protection)**

# *Notion de processus*

---

Un processus est une instance dynamique d'un programme (en cours d'exécution).

Il est caractérisé par un état et un espace mémoire qui lui sont associés.

Deux types de processus :

- **normal** : un programme quelconque dédié à une tâche donnée
- **démon** : un programme de **service** (lancé au démarrage du système) qui s'exécute en permanence  
`cron, lpd, sendmail, ...`

Un processus bascule entre deux modes d'exécution :

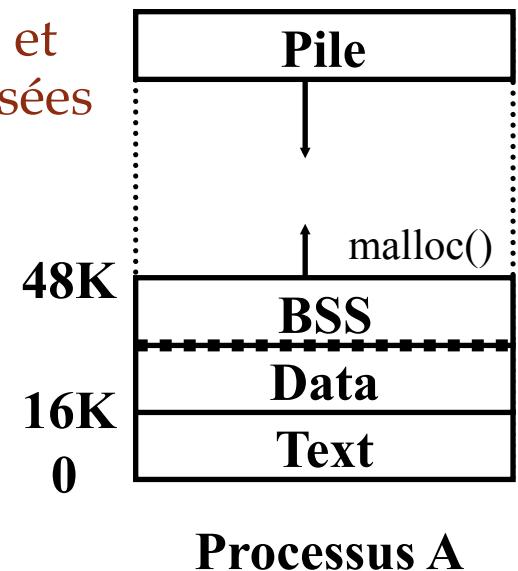
- **Noyau ou superviseur** : un programme nécessitant l'accès à une ressource système partagée
- **Utilisateur** : un programme ne nécessitant aucune ressource système

# Modèle mémoire pour un processus

Contraintes : le modèle doit être portable sur des machines hétérogènes qui possèdent des unités de gestion de mémoire (MMU) différentes.

UNIX propose un modèle avec un espace d'adressage à 3 segments :

- Un segment **texte** de taille fixe qui sera chargé avec les instructions du programme exécuté en langage machine,
- Un segment de données (**data**) composé d'un premier espace pour les données initialisées par programme et un deuxième espace pour les données non-initialisées (BSS),
- Un segment pile qui sera chargé avec les variables d'environnement exportées par le shell ainsi que la ligne de commande avec ses arguments. Cette pile contient aussi un espace pour les registres (machine d'exécution assembleur).



# *Mémoire et UNIX*

---

L'appel système **malloc()** permet d'allouer un espace de données pour des variables dynamiques en augmentant la taille du segment data. S'il n'y a plus d'espace contigu alors le segment est relogé ailleurs (suite à une interruption matérielle).

La **pile** est une interface entre le système et le processus : ce segment ne peut être modifié par programme. Elle contient toutes les informations nécessaires à la gestion du processus : registres, adresses des segments, ...

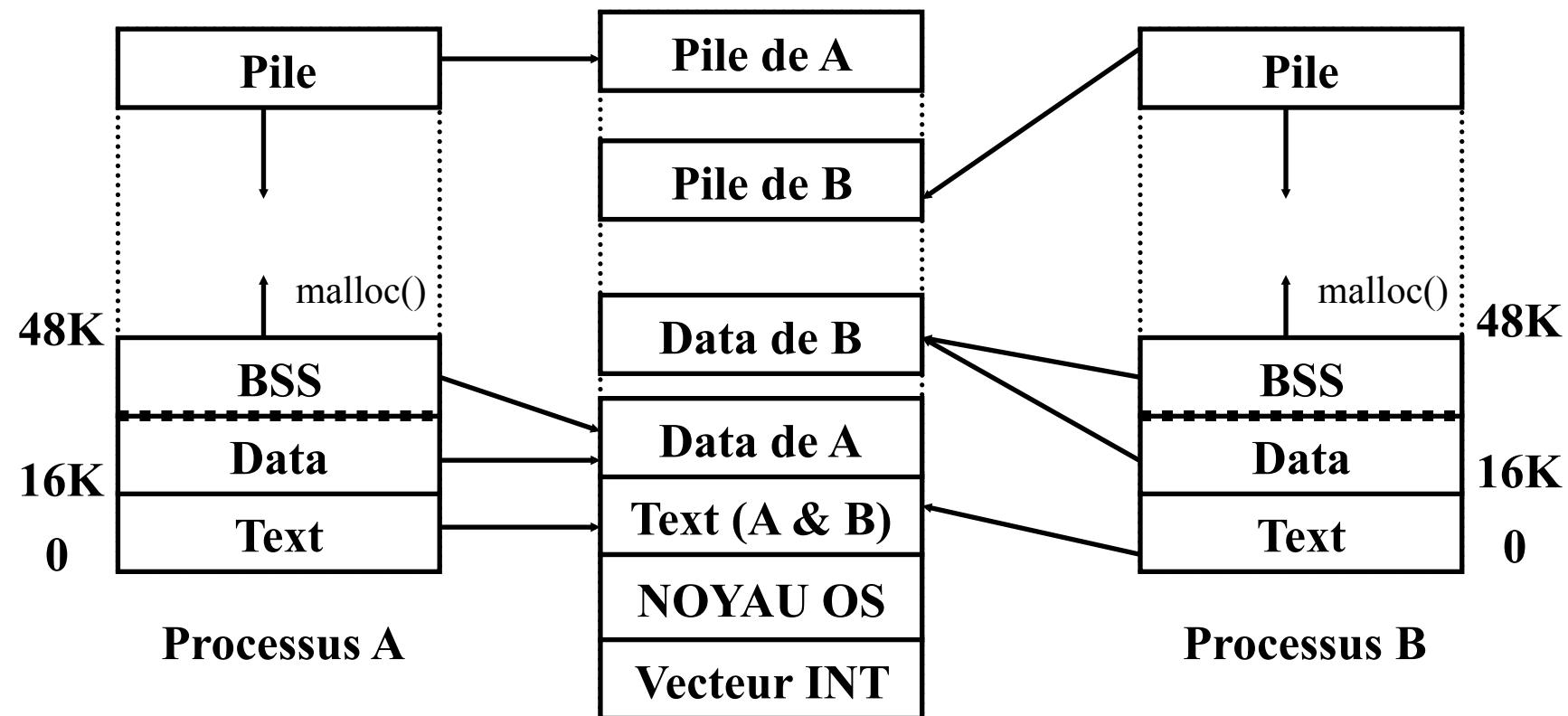
Ce segment sert aussi à l'exécution des fonctions du processus.

La pile est séparée en deux parties : une pour l'exécution en mode user et l'autre en mode noyau pour les appels système.

# UNIX & réentrance

Le modèle mémoire UNIX supporte la réentrance permettant ainsi à plusieurs processus de partager l'espace texte.

- Plusieurs demandes d'exécution d'un même programme ne génèrent pas plusieurs chargements : une **seule copie du programme** est effectuée.
- Chaque utilisateur a sa propre **zone de données**.



# *Création de processus*

---

Un système monotâche lance un processus par recouvrement (substitution) du processus courant :

- **execv()** permet ce mode de lancement sous UNIX.
- Exemple : le programme de connexion **login** est substitué par le shell de connexion. A la déconnexion, un autre **execv()** réactive le login.

Un système multitâche lance un processus par clonage du processus courant : **fork()** permet ce mode de lancement.

**fork()** crée un processus fils à l'image du père (chacun évoluant indépendamment de l'autre par la suite).

La seule différence entre le père et le fils se situe au niveau de la valeur de retour de cette fonction :

- 0 dans le fils
- Numéro du processus créé dans le père

# *fork()*

```
#include <unistd.h>
#include <stdlib.h>
#define CHECK(sts,msg) if ((sts) == -1) {perror(msg); exit(-1);}

int main ()
{
    pid_t pid;

    pid = fork();           // clonage
    CHECK(pid, "fork - Problème de création de processus !");
    // La valeur du pid dans le père vaut le PID du processus créé (dit fils)
    // Par contre pid vaut 0 dans le processus fils

    if (pid==0) {
        // traitement du fils
        exit(0);
        // Il faut prendre l'habitude d'associer systématiquement un exit avec chaque clone
        // de telle sorte que si on crée des processus en boucle, les fils ne se mettent pas à
        // leur tour à créer des clones.
    }
    // On continue le traitement spécifique du père ici
    // Un bon père doit attendre la terminaison du fils, qu'il a créé par le biais de wait() ou wait_pid()
    wait();
    exit(0);
}
```

# *Terminaison d'un processus*

---

Un processus doit toujours finir avec un `exit()`, ce qui provoque l'envoi d'un signal (cf. la partie à suivre) au processus père pour lui signifier sa terminaison.

Le processus père doit acquitter ce signal par un appel système de la famille `wait()` afin de libérer les ressources utilisées par le processus fils.

Sans cet acquittement, le processus devient Zombie et continue à consommer des ressources.

# *Processus & Identification*

---

Chaque processus est identifié par un numéro de processus (**PID**), et possède un processus père (**PPID**). Un processus ne connaît le PID d'un fils qu'à sa création.

Les appels **getpid()** et **getppid()** permettent de connaître l'identification attribuée par le système à un processus.

Un **groupe de processus** désigné par un PID ancêtre, est composé de l'ancêtre et tous les processus descendants. Un tel groupe est représenté par **-PID** comme valeur de PID.

Le processus est lancé avec l'UID de l'utilisateur. L'**EUID** (Effective) prend la valeur de UID ou SUID si ce bit est actif. Il en est de même pour le **EGID**. Les appels **getuid()** et **getgid()** permettent de connaître ces valeurs.

L'UID 0 (root) donne accès à un nombre limité et contrôlé d'appels système protégés :

- Accès au bloc 0 d'un disque pour connaître sa taille (interdiction de lire les octets bruts).

# *Implémentation d'un processus*

---

Tout processus est constitué d'une partie utilisateur et d'une partie noyau (pile).

La partie noyau n'est activée que lors d'un appel système et dispose de sa propre pile : protection contre des accès illicites.

Le noyau UNIX gère deux structures descriptives, relatives aux processus :

- Une entrée dans la **table des processus** présente en permanence en mémoire :
  - ▶ Paramètres d'ordonnancement : Priorité, Temps UC, Temps de sommeil, ...
  - ▶ Image mémoire : pointeurs sur les différents segments, ou pointeurs sur pages de swap,
  - ▶ Signaux : masque des signaux à ignorer ou à délivrer
  - ▶ Autres : PID, PPID, UID, GID, Temps avant alarme, événements attendus
- Une **structure** dite **utilisateur** est transférée sur disque en même temps que le processus et contient des informations utilisées uniquement en exécution.

# *Implémentation d'un processus*

---

La structure utilisateur contient :

- Les registres de la machine d'exécution : sauvegarde avant appel système
- Appel système : paramètres de l'appel, état, résultat produit,
- Table des descripteurs de fichiers : chaque index (fd) pointe sur l'i-node du fichier associé (cf open())
- Comptabilité : Temps de calcul consommé, Temps limite, Taille max de la pile, Nombre de pages mémoire max

La structure utilisateur est toujours contigu à la pile.

Création d'un processus par fork() :

- Allocation d'une entrée dans la table des processus,
- Duplication de l'entrée père,
- Allocation des segments data et pile (pas de segment text grâce à la réentrance),
- Duplication des segments père dans les segments fils
- Le fils est prêt pour l'exécution,

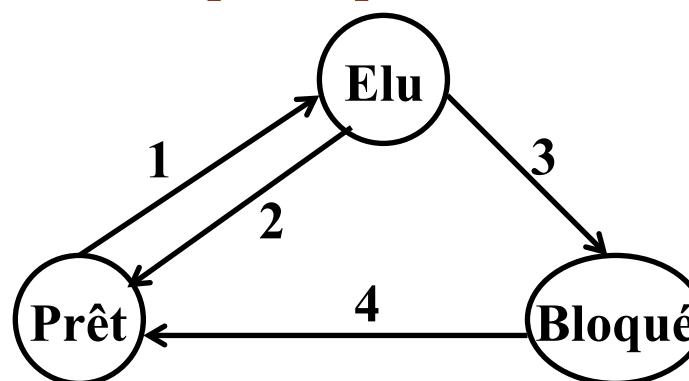
# Diagramme d'états d'un processus

Un processus possède trois états principaux :

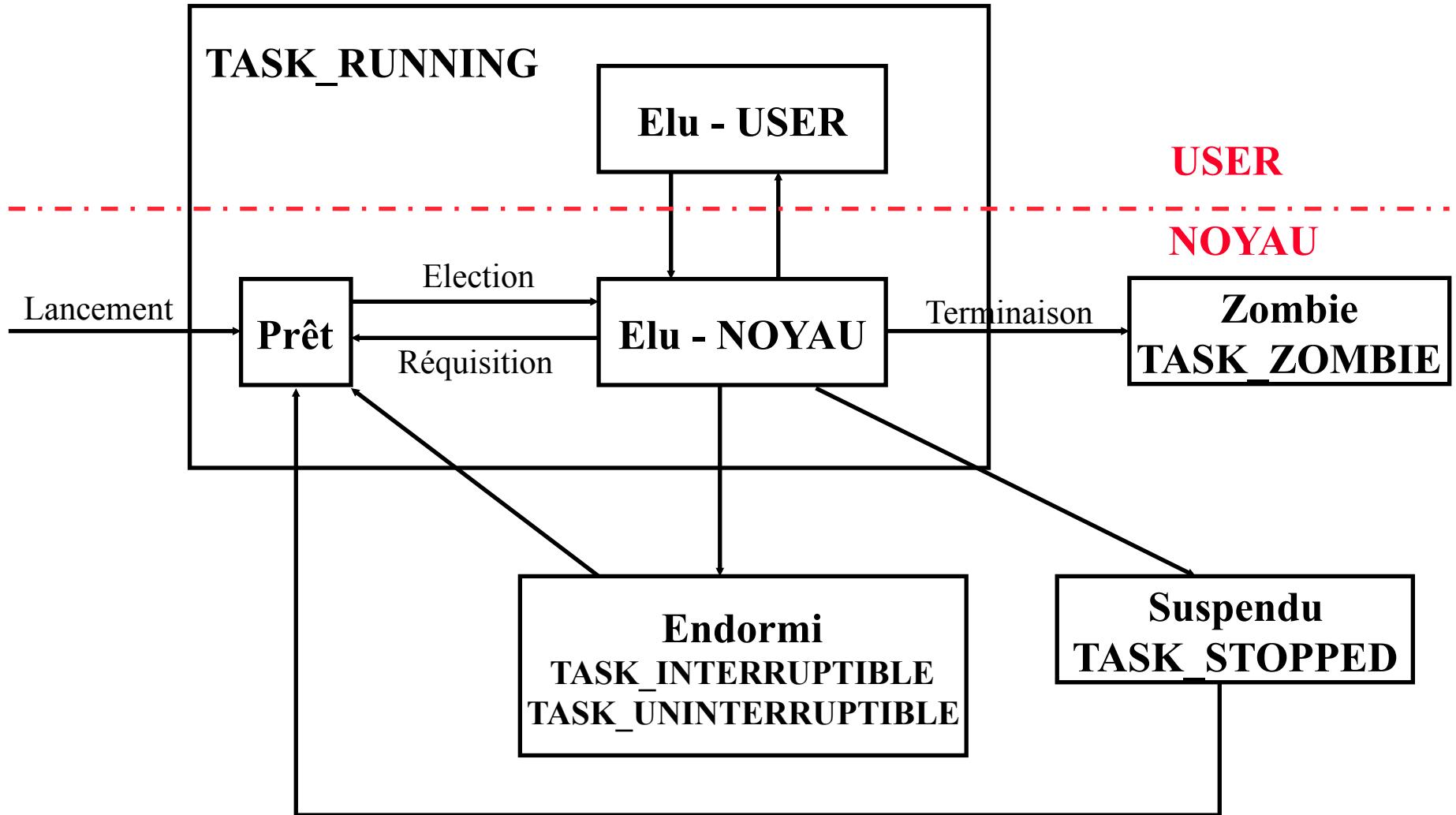
- ELU : le processus occupe le CPU
- PRÊT : le processus est en attente de l'allocation CPU
- BLOQUE : le processus est en attente de données indisponibles

Un processus change d'état suite à la réception d'un signal :

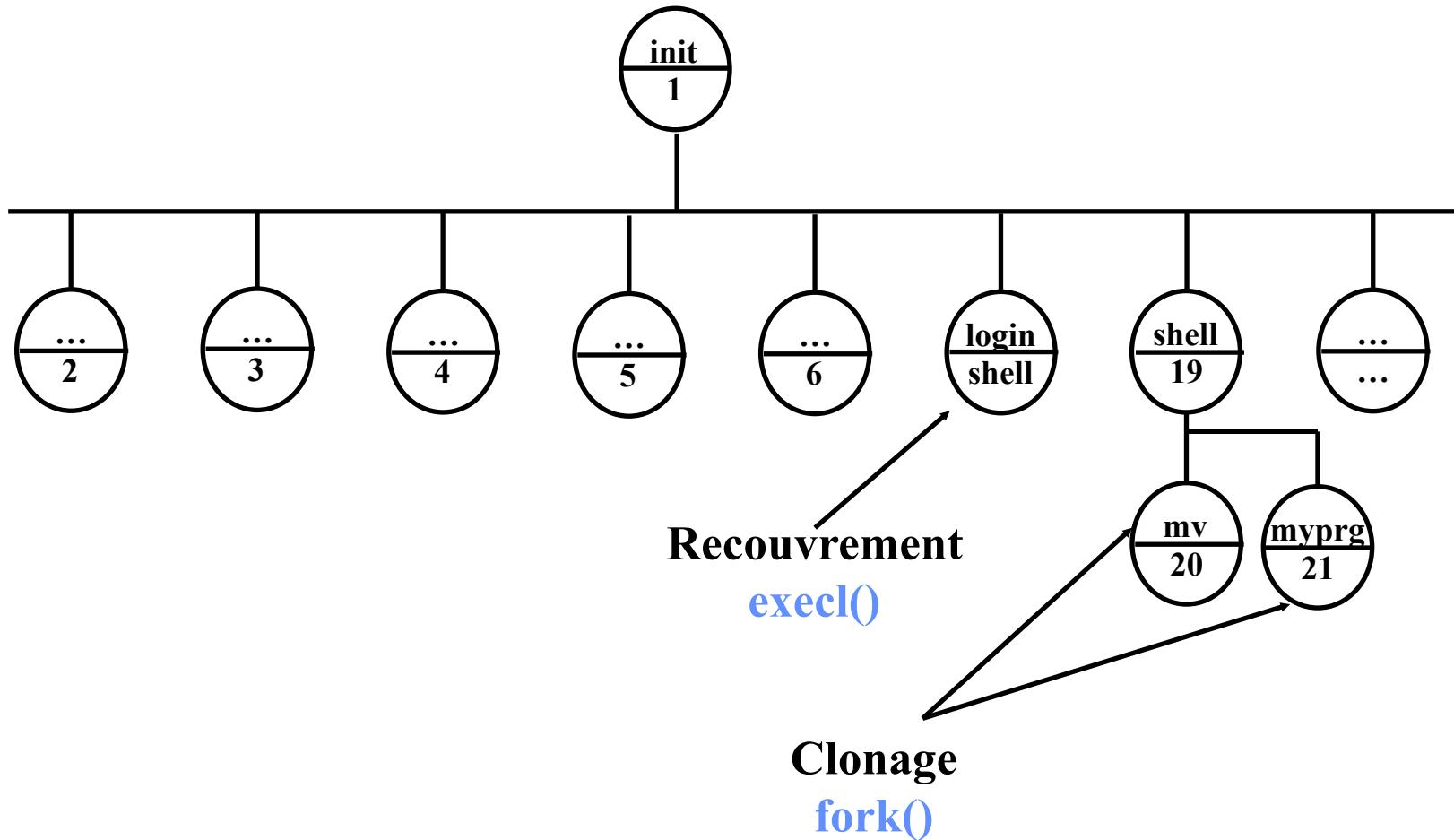
- **Election/Allocation (1)** : le processus est sélectionné par l'ordonnanceur (répartition équitable entre les processus)
- **Réquisition (2)** : le temps CPU alloué est écoulé
- **Blocage (3)** : signal généré quand le processus a besoin de données
- **Réveil (4)** : signal reçu par le processus délivrant les données



# Diagramme d'états en Linux



# Hiérarchie de processus



# *Commande ps ...*

La commande ps : afficher l'état des processus.

- a : processus d'autres utilisateurs
- r : processus "running"
- l : format long
- S : temps CPU accumulé
- u : nom d'utilisateur

Résultats de la commande ps :

- **PID** : processus identificateur
- **TT** : Terminal
- **1° STATUS** : état d'exécution.

- R : Running
- T : Terminated
- D : processus non interruptible
- P : Page wait
- S : Sleep (moins de 20 secondes)
- I : Idle (sleep depuis plus de 20 secondes)
- Z : Zombie : terminated and wait for parent

# *Commande ps*

Résultats de la commande ps (suite) :

- **2° STATUS : état de la mémoire.**
  - rien : chargé en mémoire
  - W : processus swapped out
  - > : dépassement mémoire
- **3° STATUS : priorité.**
  - N :priorité réduite
  - < : priorité augmentée
- **TIME : temps CPU consommé**
- **COMMAND : commande exécutée**
- **USER : propriétaire du processus**
- **UID : user identificateur**
- **PPID : parent processus identificateur**
- **PRI : priorité du processus**
- **NI : priorité affecté par nice**
- **SZ : taille mémoire occupée en Ko**

# Threads

Limites du fork() :

- Pas de partage de variables,
- Création d'un nouveau contexte,
- Commutation de contexte

Un thread ressemble beaucoup à un processus fils :

- Pas de création de nouveaux processus
- Partage de l'espace d'adressage
- Partage des variables globales et statiques (déclarées localement)
- Partage des descripteurs de fichiers

Utilisation sous Linux :

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

```
#include <pthread.h>
```

- Compilation :

```
▶ $ gcc -o myThreads myThreads.c -lpthread
```

# *Mise en place d'un thread (1)*

---

On commence par déclarer le traitement du thread sous forme d'une **fonction** dont le **résultat** est de type **void \***, avec **un seul argument** de type **void \*** et qui se termine par l'appel **pthread\_exit(0)**

Exemple :

```
void * myThreadFunction ( void * arg)
{
    // arg sera converti par un cast
    // Traitement du Thread
    pthread_exit(0);
}
```

## *Mise en place d'un thread (2)*

---

Ensuite, dans le `main()` on déclare une **variable** de type opaque `pthread_t` et on associe cette variable à la fonction grâce à l'appel :

```
int pthread_create(pthread_t *thread,  
                  const pthread_attr_t *attr,  
                  void *(*start_routine)(void *),  
                  void *arg);
```

Exemple :

```
void main() {  
    pthread_t th;  
    void * ret;  
    pthread_create(&th, NULL, myThreadFunction,  
                  valeur_de_arg);  
    ....  
    pthread_join(th, &ret);  
}
```

## *Modes d'exécution*

---

Par défaut, un processus est exécuté en mode user afin de protéger le système contre les actions malencontreuses : jeu d'instructions limité sans manipulation des interruptions.

L'OS s'exécute en mode noyau (superviseur) sans aucune restriction d'accès.

Cette différenciation se fait au niveau du micro-processeur (registre d'état). Intel propose 4 niveaux et Linux n'en exploite que deux :

- Noyau : niveau 0
- User : niveau 3

## *Commutation de contexte ...*

---

Définition : Une commutation de contexte caractérise la **transition** réalisée lorsque'un programme invoque un autre programme (fonction). La fin du programme appelé rétablit le contexte initial.

A l'échelle de l'OS, cette commutation se produit lors d'un appel système avec un passage du mode user au mode noyau.

Cette commutation s'accompagne d'une sauvegarde du contexte actuel, la création d'un nouveau contexte chargé avec l'adresse de la routine système et le passage du registre d'état en mode noyau.

# *Commutation de contexte*

---

3 causes de commutations :

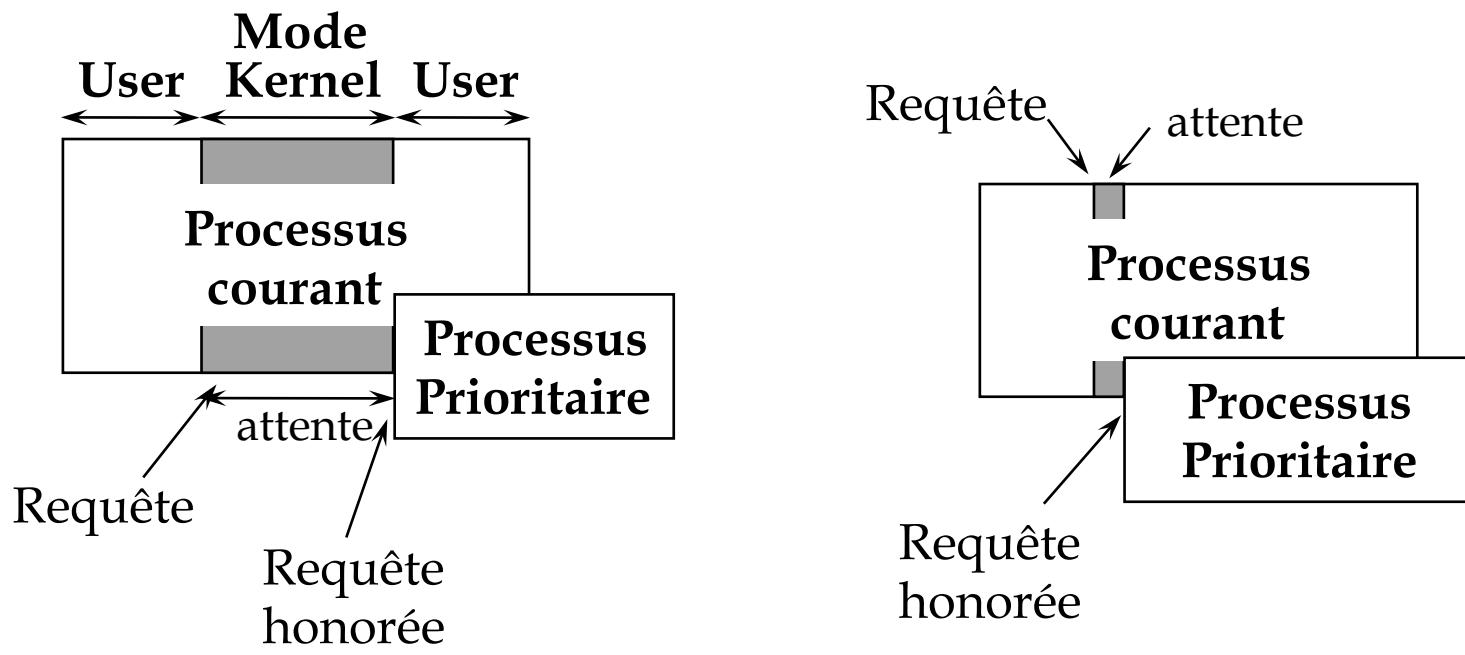
- Appel système : demande explicite
- Opération illicite : on parle de trappe ou exception. La conséquence directe d'une telle situation est l'arrêt du programme ayant provoqué l'exception.
  - ▶ Division par 0,
  - ▶ Violation d'accès mémoire,
  - ▶ Instruction interdite,
  - ▶ ...
- Prise en compte d'une interruption matérielle par l'OS :
  - ▶ Arrêt du programme utilisateur,
  - ▶ Exécution de la routine associée à l'interruption,
  - ▶ Reprise du programme utilisateur

# Préemption

Noyau Préemptif : un processus de priorité supérieure peut interrompre un processus de priorité inférieure quelque soit le mode (user ou kernel)

Unix standard (Linux) est non Préemptif !

Linux peut être configuré en mode préemptif



# *Ordonnancement des processus*

---

Trois politiques d'ordonnancement cohabitent (POSIX 1003.4) :

- Deux politiques pour les processus temps réel : SCHED\_FIFO et SCHED\_RR
- Une politique pour les processus classiques : SCHED\_OTHER

Ordre de priorité entre les politiques :

- FIFO, RR et enfin OTHER

Chaque processus se voit attribuer une politique et une priorité. Cette priorité est **fixe** pour FIFO et RR. Par contre, un processus de type OTHER a une priorité **dynamique** en fonction de sa consommation UC passée (équité).

L'ordonnancement réalisé par le noyau est découpé en périodes. Au début de chaque période, on calcule le temps CPU attribué à chaque processus : nombre de ticks d'horloge durant lequel un processus peut être exécuté.

## ***SCHEDEX FIFO***

---

Premier arrivé, premier servi

Priorité fixe

Parmi les processus ayant la priorité la plus élevée en choisi le plus âgé de tous.

Le processus garde le CPU jusqu'à la terminaison ou préemption par un autre processus plus prioritaire.  
Dans ce cas le processus reprend sa place dans la file.

## ***SCED\_RR***

---

Tourniquet à quantum de temps entre processus de même priorité.

Le processus ayant la plus haute priorité est élu.

Le processus est préempté à la fin du quantum alloué et réintègre la queue de la file correspondant à sa priorité.

Après préemption d'un processus RR, le processus temps réel ayant la priorité la plus élevée est élu.

# *SCED\_OTHER : Temps partagé*

Deux niveaux algorithmiques :

- Niveau bas : élection d'un processus à exécuter parmi les processus prêts.
- Niveau haut : va et vient entre le disque et la mémoire pour donner une chance à tous les processus de s'exécuter.

On utilise une file d'attente pour regrouper les processus candidats (en mémoire) et ayant la même priorité :

- Mode noyau : priorité négative,
- Mode utilisateur : priorité positive.

L'ordonnanceur de bas niveau parcourt la table des files d'attente en partant de la priorité la plus haute (la valeur la plus négative), jusqu'à trouver une file d'attente non vide.

Un processus garde le CPU pendant un temps max dit quantum. A chaque pulsation d'horloge, on incrémente le compteur UC qui sera ajouté à la priorité pour affaiblir sa priorité.

## ***SCED\_OTHER (2)***

---

Un processus ayant consommé son quantum est mis en queue et l'algorithme s'exécute de nouveau : gestion en tourniquet pour les processus de même priorité.

Toutes les secondes, on recalcule les priorités :

$$\text{Nouvelle priorité} = \text{base (0 ou valeur de nice)} + \text{utilisation UC}/2$$

Un processus bloqué par un appel système sera retiré de la file d'attente. Sur occurrence de l'événement attendu, le processus est remis dans une file d'attente ayant une priorité négative selon l'événement.

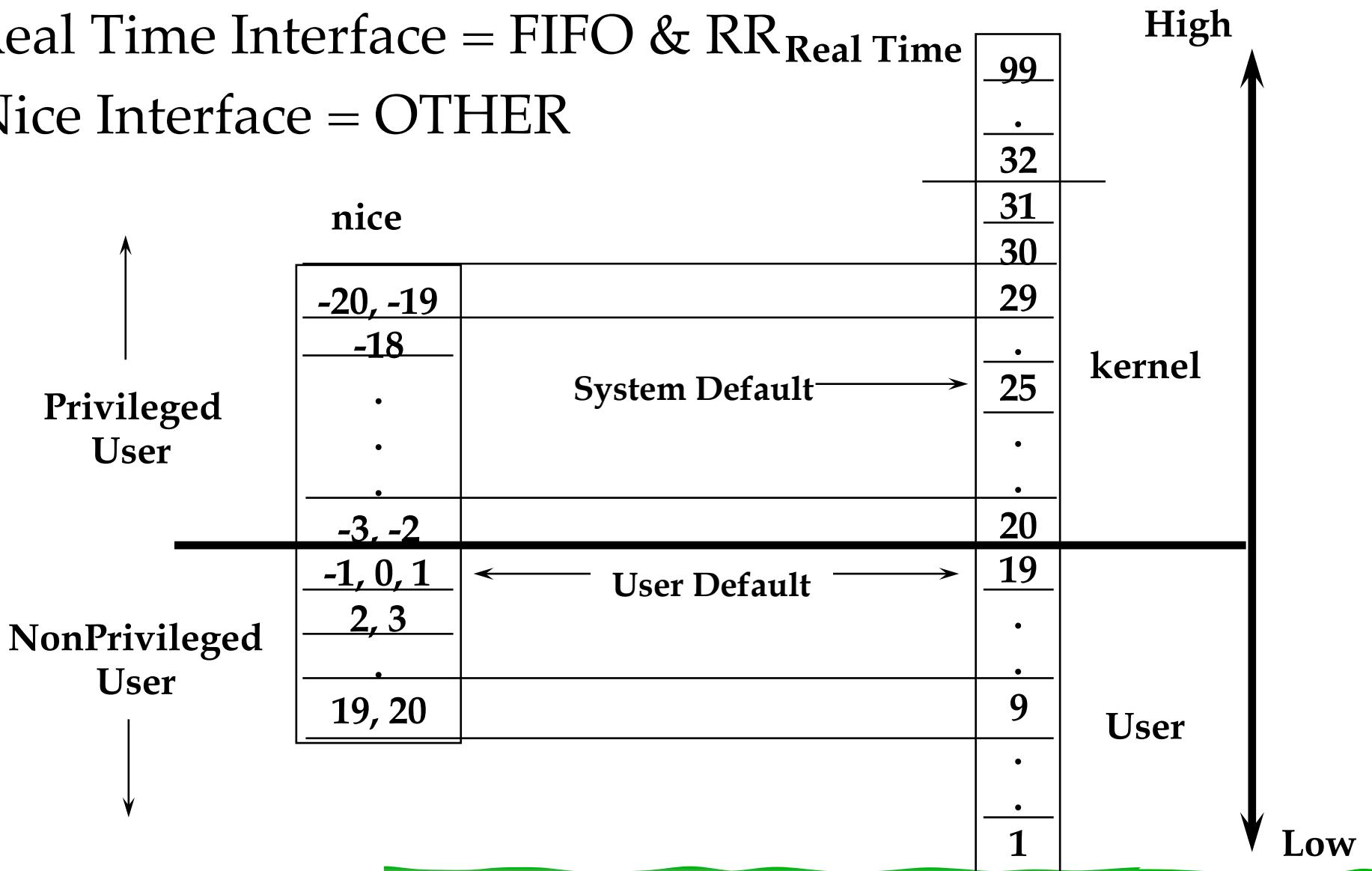
Priorité des processus en mode noyau :

- -1 : attente fin d'un fils
- -2 : attente sortie terminal
- -3 : attente entrée terminal
- -4 : attente tampon disque
- -5 : attente E/S sur disque
- ....

# Scheduling

Real Time Interface = FIFO & RR Real Time

Nice Interface = OTHER



# *Nice Interface*

au niveau du shell :

- **nice [-n priority] command**
- **renice [-n increment] pid**
  - ▶ increment seulement positif !
- pour visualiser ses priorités
  - ▶ **ps -O user,nice (A pour tous les processus)**

par programme :

- **setpriority() – getpriority() – nice()**

# *Realtime Interface*

au niveau du shell :

- pour visualiser ses priorités
  - ▶ \$ ps -ae0 psxpri
    - ⇒ -a et e <=> -A

par programme :

- Utilisation d'une structure dans sched.h

```
struct sched_param {  
    #ifdef POSIX_4D10  
        int priority;          /* Scheduling priority. */  
    #else  
        int sched_priority; /* Scheduling priority. */  
    #endif  
};
```

# Fonctions d'ordonnancement

`sched_getscheduler () :`

- retourne la politique de scheduling d'un process

`sched_getparam () :`

- priorité d'un process

`sched_get_priority_min () et sched_get_priority_max () :`

- priorité (min/max) pour une politique de scheduling

`sched_rr_get_interval () :`

- retourne quantum pour round-robin

`sched_setscheduler () :`

- fixe politique de scheduling et priorité

`sched_setparam () :`

- uniquement politique de scheduling

`sched_yield () :`

- permet de passer la main à un autre processus
- très utile pour FIFO scheduling, beaucoup moins pour Round-robin

# Exemple

```
#include <unistd.h>
#include <sched.h>
#include <stdio.h>
// Contrôle des appels système
#define CHECK(sts,msg) if ((sts) == -1) \
    perror(msg); exit(-1);

int main (void)
{
    int my_pid = 0; // le processus courant
    struct sched_param param;
    int old_policy, old_priority;
    int low_priority, high_priority;
    // Valeur max de la priorité avec la politique FIFO
    CHECK( high_priority =
        sched_get_priority_max(SCHED_FIFO),
        "sched_get_priority_max");
    printf("Priorité max pour SCHED_FIFO : %d\n",
        high_priority);
    // Valeur min de la priorité avec la politique FIFO
    CHECK( low_priority =
        sched_get_priority_min(SCHED_FIFO),
        "sched_get_priority_min");
    printf("Priorite min pour SCHED_FIFO : %d\n",
        low_priority);
    // Valeur courante de la politique
    CHECK( old_policy = sched_getscheduler(my_pid),
        "sched_getscheduler");
    printf("Scheduling politique du process : %d\n",
        old_policy);
    // Valeur courante de la priorité
    CHECK( sched_getparam(my_pid, &param) ,
        "sched_getparam");
    printf("Ancienne priorité du process : %d\n",
        param.sched_priority);
    // Prépare le nouveau paramètre avec une
    // priorité max pour une politique FIFO
    param.sched_priority = high_priority;
    // Appliquer une politique FIFO
    CHECK( sched_setscheduler(my_pid,
        SCHED_FIFO, &param),
        "sched_setscheduler");
    exit(0);
}
```

Dommage !

## PARTIE 2 : SIGNAUX

# *Vecteur d'interruptions*

---

Le système gère une table de type `idt_table` à 256 entrées appelé vecteur d'interruptions :

- Index : numéro de l'interruption,
- Chaque entrée de la table contient l'adresse mémoire du gestionnaire chargé de la prise en compte de l'interruption,
- 0-31 : interruptions non-masquables et trappes,
- 32-47 : interruptions matérielles levées par les périphériques,
- 128 : appel système,
- 48-255 (sauf 128) : interruptions programmables.

Ce vecteur est chargé en mémoire lors du démarrage du système.

# *Prise en compte d'une interruption*

---

L'UC avant d'exécuter une nouvelle instruction vérifie si une INT lui a été délivrée. Si présence alors :

- L'UC contrôle l'autorité de l'interruption (vérification de la source),
- L'UC accède à l'entrée dans le vecteur pour récupérer l'adresse mémoire du gestionnaire.
- Passage en mode noyau si nécessaire et commutation de contexte : sauvegarde du contexte actuel et chargement du contexte gestionnaire.
- Exécution du gestionnaire de l'interruption
- Restauration du contexte précédent

## *Interruption matérielle ...*

---

Une interruption matérielle est un mécanisme qui demande au système d'interrompre le traitement en cours et d'effectuer le traitement associé à l'interruption matérielle appelée routine de traitement de l'interruption.

La famille Intel 386 admet 15 niveaux d'interruptions (IRQ0 à IRQ15), qui sont gérés par un contrôleur d'interruptions (deux circuits 8259A cascadés par la broche IRQ2 du 1<sup>o</sup> circuit).

Un ou plusieurs périphériques sont branchés sur un niveau d'interruption. Si plusieurs, une scrutation est nécessaire pour identifier le périphérique concerné.

# *Interruption matérielle*

---

Protocole de traitement :

- Émission par le contrôleur d'un signal sur la broche INT,
- Le processeur acquitte sur la broche INTA,
- Le contrôleur place le numéro de l'interruption sur le bus de données,
- L'UC utilise ce numéro pour indexer le vecteur (offset de 32),
- Exécution du gestionnaire avec 3 situations possibles :
  - ▶ Actions critiques, immédiates, interruptions masquées,
  - ▶ Actions non critiques, exécution rapide, interruptions démasquées,
  - ▶ Actions qui peuvent être différées, exécution longue, non critiques pour le fonctionnement du noyau,
- Restitution du contexte d'appel :
  - ▶ Si le niveau d'imbrication est à 1 alors finalisation des actions différées et retour au mode user.

# *Exceptions ou trappes*

---

Quand l'UC rencontre une situation anormale, une exception est levée (une trappe est ouverte) (on parle aussi d'interruption logicielle) :

- Division par 0, Violation d'accès mémoire, Instruction interdite,  
...

Le traitement d'une trappe est assez similaire à celui des interruptions matérielles. La différence réside dans le fait que le traitement est **synchrone** avec l'exception.

Sur occurrence d'une trappe, un signal est envoyé au programme fautif qui exécute alors un traitement par défaut ou programmé.

# *Traitement d'une exception*

---

Un processus peut programmer une interruption, l'ignorer (bloquer le signal), ou réaliser le traitement par défaut (en général, tuer le processus) :

- La programmation d'un signal permet d'interrompre le traitement principal, et de passer la main à une procédure de traitement avant la reprise de celui-ci (traitement principal).
- Le traitement d'une interruption peut être à son tour interrompu par une autre interruption
- Les traitements par défaut possibles sont :
  - ▶ TERM (terminaison),
  - ▶ CORE (terminaison avec image mémoire),
  - ▶ STOP (suspension),
  - ▶ CONT (reprise),
  - ▶ IGN (ignorer),

# *Appel système*

---

Un appel système est invoqué soit par un programme soit par le biais d'une commande.

Protocole de traitement d'un appel système :

- Copie des paramètres (maximum 5) dans les registres EBX, ECX, EDX, ESI et EDI,
- Chargement du registre EAX avec le numéro de la routine, qui sert d'index pour le gestionnaire des appels dans une table d'appels système (256 entrées) chargée avec les adresses des routines,
- Interruption logicielle (numéro 128),
- Sauvegarde du contexte sur la pile noyau
- Passage en mode noyau,
- Exécution de la routine invoquée avec résultat dans EAX,
- Restauration du contexte

# *Signaux classiques ...*

---

Un **signal** est un **message** (non-valué) envoyé par le noyau à un processus ou un groupe de processus pour signifier qu'un **événement** est survenu au niveau du système.

Le **numéro** de signal (nom de signal) représente l'événement qui vient de se produire :

- 1-31 : signaux classiques
- 32-63 : signaux temps-réel
- Les signaux portent un nom préfixé par SIG : SIGINT, SIGCHLD,

Si le processus n'est pas élu alors le signal est **mis en attente**. Plusieurs occurrences d'un même signal sont réduites à une seule occurrence.

# *Signaux classiques*

---

Si le processus est élu alors le signal est **délivré** de suite.

La réaction à un signal délivré est immédiate  
(comportement **synchrone**).

La délivrance d'un signal à un processus provoque une interruption logicielle (déroulement de processus)

- Un processus, sur réception du signal, exécute une routine :  
« **signal handler** »

Un processus a la possibilité de **bloquer** un signal (ne pas être réceptif à un signal). Certains signaux ne peuvent être bloqués.

Attention à ne pas confondre interruptions et signaux.

# *signal.h ...*

*/\* \* valid signal values: all undefined values are reserved for future use  
•note: POSIX requires a value of 0 to be used as the null signal in kill() \*/*

#define SIGHUP	1	<i>/* hangup, generated when terminal disconnects */</i>
#define SIGINT	2	<i>/* interrupt, generated from terminal special char */</i>
#define SIGQUIT	3	<i>/* (*) quit, generated from terminal special char */</i>
#define SIGILL	4	<i>/* (*) illegal instruction (not reset when caught) */</i>
#define SIGTRAP	5	<i>/* (*) trace trap (not reset when caught) */</i>
#define SIGABRT	6	<i>/* (*) abort process */</i>
#define SIGEMT	7	<i>/* (*) EMT instruction */</i>
#define SIGFPE	8	<i>/* (*) floating point exception */</i>
#define SIGKILL	9	<i>/* kill (cannot be caught or ignored) */</i>
#define SIGBUS	10	<i>/* (*) bus error (specification exception) */</i>
#define SIGSEGV	11	<i>/* (*) segmentation violation */</i>
#define SIGSYS	12	<i>/* (*) bad argument to system call */</i>
#define SIGPIPE	13	<i>/* write on a pipe with no one to read it */</i>
#define SIGALRM	14	<i>/* alarm clock timeout */</i>
#define SIGTERM	15	<i>/* software termination signal */</i>
#define SIGURG	16	<i>/* (+) urgent contition on I/O channel */</i>

# ... *signal.h*

#define SIGSTOP	17	/* (@) stop (cannot be caught or ignored) */
#define SIGTSTP	18	/* (@) interactive stop */
#define SIGCONT	19	/* (!) continue if stopped */
#define SIGCHLD	20	/* (+) sent to parent on child stop or exit */
#define SIGTTIN	21	/* (@) background read attempted from control terminal */
#define SIGTTOU	22	/* (@) background write attempted to control terminal */
#define SIGIO	23	/* (+) I/O possible, or completed */
#define SIGXCPU	24	/* cpu time limit exceeded (see setrlimit()) */
#define SIGXFSZ	25	/* file size limit exceeded (see setrlimit()) */
#define SIGVTALRM	26	/* virtual time alarm (see setitimer) */
#define SIGPROF	27	/* profiling time alarm (see setitimer) */
#define SIGWINCH	28	/* (+) window size changed */
#define SIGINFO	29	/* (+) information request */
#define SIGUSR1	30	/* user defined signal 1 */
#define SIGUSR2	31	/* user defined signal 2 */

# Fonctions principales

## sigprocmask() :

- permet de définir la réceptivité du processus vis-à-vis de chaque signal :
  - ▶ Ainsi, sur occurrence d'un signal, le processus y sera soit **réceptif** ou **non**

## sigaction() :

- permet de définir le signal handler (le traitement à effectuer sur réception d'un signal) :
  - ▶ Nom de la routine utilisateur pour le traitement de l'interruption. Cette routine ne prend qu'un **seul** argument de type int et il prend la **valeur du signal** ayant provoqué l'interruption.
  - ▶ **SIG\_DFL** : action par défaut
  - ▶ **SIG\_IGN** : nop
  - ▶ La commande **man 7 signal** permet de connaître le traitement par défaut de chaque signal

## kill() :

- permet d'envoyer un signal à un process

# *Interactive kill*

\$ kill -l

- liste les signaux gérés par le système

\$ kill [-signal\_name | -signal\_number] pid ...

- \$ kill -9 6996 équivalent à \$kill -KILL 6996
- \$ kill -KILL 0
  - ▶ Cette commande envoie SIGKILL à tous les processus appartenant au « shell process group »
- \$ kill -USR1 6996

**Appellation trompeuse** : kill délivre un signal en général et pas seulement pour tuer un processus (un signal parmi d'autres).

# *kill()*

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

si  $pid > 0$ , kill envoie le signal au process pid

si  $pid < 0$ , kill envoie le signal aux process dont le groupe ID est égal  $|pid|$

Parricide :

```
if (kill(getppid(), SIGTERM) == -1)
    perror("Error in kill");
```

la fonction **raise()** permet de s'auto-envoyer un signal.

# *Masque : préparation*

---

Permet de bloquer ou d'accepter un signal

Utilise un masque de type opaque **sigset\_t**

```
#include <signal.h>
```

```
int sigemptyset(sigset_t *set);           // masque vide
int sigfillset(sigset_t *set);             // masque plein
int sigaddset(sigset_t *set, int signo);   // ajoute
int sigdelset(sigset_t *set, int signo);   // enlève
int sigismember(const sigset_t *set, int signo); // membre ?
```

# *Masque : application*

---

Pour rendre opérationnel ce nouveau masque, on utilise la fonction **sigprocmask()**.

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
```

how

- **SIG\_BLOCK** : ajoute les signaux du masque au masque courant
- **SIG\_UNBLOCK** : retire les signaux du masque courant
- **SIG\_SETMASK** : affecte et remplace le masque courant

set : valeur du masque (peut être NULL)

oset : ancienne valeur (peut être NULL)

# *Exemple : mise en œuvre d'un masque*

```
#include <unistd.h>
#include <signal.h>
#define CHECK(sts,msg) if ((sts) == -1) { perror(msg); exit(-1); }

int main () {
    sigset_t intmask;
    int sts;
    // Initialise et affecte le masque
    sts = sigemptyset(&intmask);
    CHECK(sts, "appel de sigemptyset");
    sts = sigaddset(&intmask, SIGINT);
    CHECK(sts, "appel de sigaddset");
    // On interdit le control-C
    sts = sigprocmask(SIG_BLOCK, &intmask, NULL);
    CHECK(sts, "appel de sigprocmask");
    // On attend pour tester
    alarm(60);
    pause();
    exit(0);
}
```

L'appel `alarm()` arme une alarme et continue le traitement. Sur fin de temporisation `SIGALRM` est reçu par le processus.

L'appel `pause()` se débloque sur réception d'un signal qcq

Le programme dure 60 secondes et quitte; pendant ce temps le control-C est inhibé.

# *sigaction()*

Installe un déroutement sur occurrence d'un signal.

Ce déroutement peut être celui par défaut (**SIG\_DFL**), ou un traitement particulier défini par le programme (**fonction**) ou un traitement nul (**SIG\_IGN**).

```
#include <signal.h>
int sigaction(int signo,
              const struct sigaction *new_act,
              struct sigaction *old_act);
struct sigaction {
    void (*sa_handler)();           // SIG_IGN, SIG_DFL ou fonction
    sigset_t sa_mask;               // masque durant le déroutement
    int sa_flags;} ;                // options particulières
```

# Déroulement de signal

```
#include <signal.h>
#define CHECK(sts,msg) if ((sts) == -1) {perror(msg); exit(-1); }

struct sigaction newact;
void monSIGUSR1(int numero_signal);

newact.sa_handler = monSIGUSR1;
sigemptyset(&newact.sa_mask);
newact.sa_flags = 0;
CHECK(sigaction(SIGUSR1, &newact, NULL),
      "problème avec sigaction");
```

La routine **monSIGUSR1** n'accepte qu'un seul paramètre qui est affecté avec le numéro de signal déclencheur.

# Exemple : déroutement d'un signal

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

#define CHECK(sts,msg) \
    if ((sts) == -1) {perror(msg); exit(-1);}

int main()
{
    struct sigaction newact;
    int sts;
    void deroute();

    newact.sa_handler = deroute;
    CHECK(sigemptyset(&newact.sa_mask),
          "problème sigemptyset");
    newact.sa_flags = 0;

    // on installe le handler deroute
    CHECK(sigaction(SIGUSR1, &newact, NULL),
          "problème sigaction");

    // on attend les signaux
    for(;;)
        pause();
    printf("On est passé par ici : "
           "pause se réveille sur un signal.\n");
    exit(0);
}

void deroute (int signal_number)
{
    switch (signal_number) {
    case SIGUSR1 : printf("Signal SIGUSR1 "
                           "reçu.\n"); break;
    case .....
    }
}
```

## *Appels annexes*

---

**alarm()** : déclenche **SIGALRM** après tempo

**pause()** : attend l'occurrence d'un signal

**wait()**: attend les enfants

**waitpid()** : attend un processus fils donné

**sleep()** et **nanosleep()** : suspend le traitement pendant la durée (paramètre) et se déclenche sur occurrence d'un signal de fin de sommeil.

De toute manière, il faut toujours vous référer au **man** de **votre** système.



# *Signaux temps-réel*

---

Linux propose 32 signaux dits temps-réel :

- 32-63,
- Pas de nommage : SIGRTMIN=32 et SIGRTMAX=64
- On désigne un signal TR par son rang effectif dans l'intervalle :
  - ▶ SIGRTMIN+1, SIGRTMIN+3, SIGRTMAX-5, ...

Caractérisation :

- Empilement des occurrences de chaque signal : file d'attente,
- Priorité entre signaux : SIGRTMIN est prioritaire sur SIGRTMAX,
- Fourniture d'informations supplémentaires en plus du numéro de signal.

# *Fonctions de gestion des signaux TR ...*

Envoi d'un signal TR :

- `kill()` : pas d'informations supplémentaires,
- `int sigqueue (pid_t pid, int num_sig, const union sigval info):`
  - ▶ `info.sival_int` de type `int`,
  - ▶ `info.sival_ptr` de type `void *`,

Définition d'un traitement d'un signal TR :

- On définit un gestionnaire de la forme  
`void deroute(int num_sig, struct siginfo *info, void *rien)`
- Contenu de la structure `siginfo` :
  - ▶ `int si_signo` : numéro du signal,
  - ▶ `int si_code` : source du signal,
  - ▶ `int si_value.sival_int` : correspond au champ `info.sival_int` de `sigqueue()`,
  - ▶ `int si_value.info.sival_ptr` : correspond au champ `info.sival_ptr` de `sigqueue()`,

# *Fonctions de gestion des signaux TR*

---

Association d'un traitement à un signal TR :

- On renseigne une structure de type `sigaction` comme pour les signaux classiques sauf qu'on utilise le champ `sa_sigaction` et non pas `sa_handler` avec le gestionnaire.
- La structure `sigaction` est modifiée :

```
struct sigaction {  
    union {  
        void (*sa_handler) ();  
        void (* sa_sigaction)(int, struct siginfo*, void*);  
    } u;  
    sigset_t sa_mask;  
    unsigned long sa_flags;};
```

- Attachement du traitement au signal par `sigaction()` comme pour les signaux classiques.

# *Exécution d'un gestionnaire TR*

Pour éviter les commutations de contexte (pénalisant pour le TR), Linux offre deux primitives qui permettent d'attendre l'occurrence de signaux (sans invocation du gestionnaire de signaux par le noyau) :

- `int sigwaitinfo(const sigset *signaux, struct siginfo *info);`
- `int sigtimedwait(const sigset *signaux, struct siginfo *info, const struct timespec *delai);`

`const sigset *signaux` désigne les signaux attendus.

`sigwaitinfo()` attend indéfiniment l'occurrence d'un des signaux alors que `sigtimedwait()` permet une attente sur un intervalle de temps fini.

`struct timespec { long tv_sec, long tv_nsec;};`

## PARTIE 3 : *SYNCHRONISATION & INTERBLOCAGE*

# Ressource

---

La notion de ressource recouvre toute entité nécessaire à l'exécution d'un processus. Elle peut être :

- Matérielle : un processeur, une imprimante, un écran, ...
- Logicielle : une variable, une zone mémoire,

Le noyau système garantit :

- la protection des données utilisateurs : tous les segments mémoire
- Les appels systèmes donnent un accès exclusif aux ressources système qu'elles soient matérielles ou logicielles

Dès l'instant où une ressource est partagée par plusieurs processus alors en se retrouve en situation de **concurrence** :

- Nécessité de mécanismes de protection pour permettre le partage entre processus utilisateurs (droits d'accès).

# *Ressource & synchronisation*

---

Une ressource se caractérise par son état et le nombre d'exemplaires (nombre d'utilisateurs maximal).

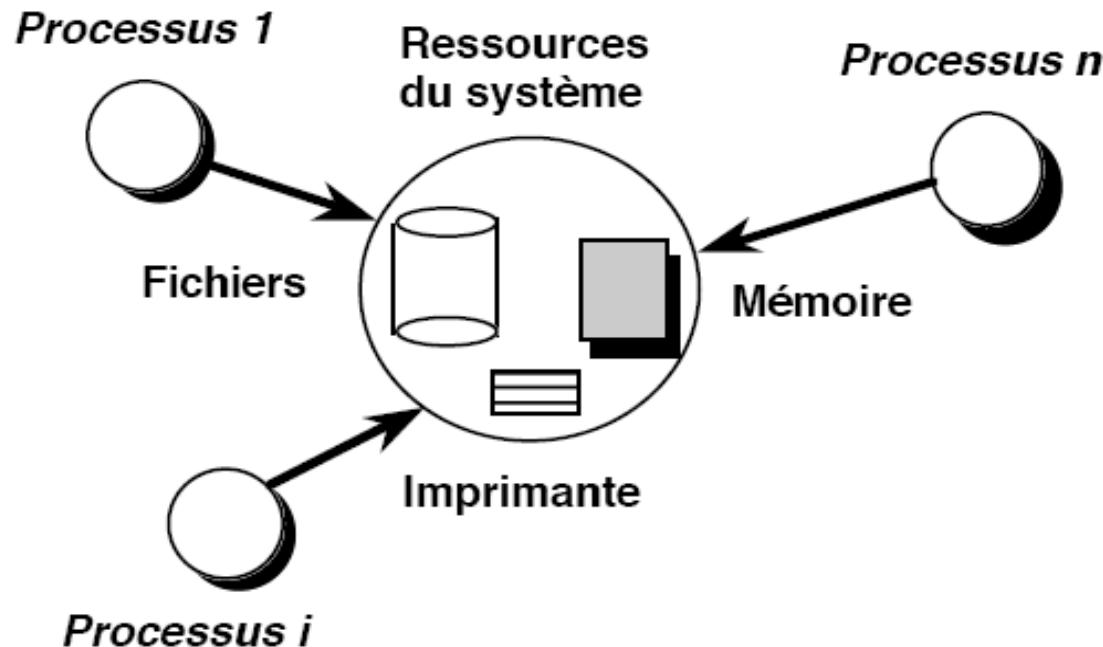
L'utilisation d'une ressource doit respecter un protocole en trois étapes :

- Allocation : étape bloquante
- Utilisation
- Libération / Restitution

Plusieurs schémas classiques de la communication interprocessus ont été définis :

- Exclusion mutuelle,
- Synchronisation,
- Producteurs / Consommateurs & Rédacteurs / Lecteurs

# Ressource critique



**Accès exclusif : une ressource critique est une ressource ne pouvant être utilisée que par un seul processus à la fois.**

# *Exclusion mutuelle*

---

L'accès à une ressource critique doit se faire en **exclusion mutuelle**.

Pour réaliser l'exclusion mutuelle, il y a plusieurs conditions ou propriétés à respecter :

1. à tout instant **un processus au plus** peut se trouver en section critique,
2. si plusieurs processus sont bloqués en attente de la ressource critique, **l'un d'eux doit pouvoir y entrer au bout d'un temps fini**,
3. si un processus est hors de sa section critique, **il ne doit pas empêcher** l'entrée d'un autre processus dans sa section critique,
4. **la solution doit être la même** pour tous les processus.

Hypothèse :

- Tout processus sort de sa section critique au bout d'un temps fini.

# *Désactivation des interruptions*

---

La solution la plus simple pour réaliser l'exclusion est de désactiver les interruptions durant le traitement critique : l'ordonnanceur ne peut donc interrompre le traitement.

Solution utilisée par le système en mode noyau :

- Le mode noyau permet la désactivation des interruptions
- Les programmes utilisateurs ne peuvent désactiver les interruptions :
  - ▶ Que se passerait-il si l'utilisateur ne réactive pas les interruptions après avoir quitter le traitement critique ?
  - ▶ Solution non appropriée
  - ▶ Il faut donc proposer un autre mécanisme.

# *Implémentation de la gestion des ressources*

---

Plusieurs études ont été menées pour gérer une ressource critique (Dekker, Peterson). Néanmoins, les solutions sont toutes basés sur l'attente **active**. La solution qui a été la plus utilisée est à base de TSL (Test and Set Lock) : cellule câblée et ensuite instruction assembleur.

Les **sémaphores** de Dijkstra (1965) proposent un mécanisme utilisant une attente **passive** : réveil sur disponibilité de la ressource.

NB : cf. annexe pour les solutions de Dekker, Peterson et TSL

# *Les Sémaphores*

---

Gestion d'une ressource partageable par n processus ( $n > 0$ ).

Pour la gestion du sémafore, on dispose de :

- une valeur initiale  $e0[s]$ ,
- une variable entière  $e[s]$ ,
- une file d'attente  $f[s]$ .

Le sémafore est manipulé uniquement par trois primitives indivisibles (interruptions masquées) :

- $\text{init}(s, \text{val}0)$ ,
- $\text{P}(s)$  : Prise de sémafore,
- $\text{V}(s)$  : Libération de sémafore.

# *Les sémaphores : Initialisation*

---

L'opération init :

- elle a pour but d'initialiser le sémaphore, c'est-à-dire qu'elle met à vide la file d'attente  $f(s)$  et initialise avec la valeur  $e0(s)$  le compteur  $e(s)$ . on définit ainsi le nombre de jetons initiaux dans le sémaphore.

```
void init(semaphore sem, int val0) {  
    masquer_it;  
    e0[sem] = val0;  
    e[sem] = e0[sem];  
    Mettre f[sem] à vide;  
    démasquer_it;  
}
```

# *Les sémaphores : Allocation*

L'opération P(sem) :

- L'opération P(sem) attribue un jeton au processus appelant s'il en reste au moins un ; sinon elle bloque le processus dans la file f(s). L'opération P est donc une opération bloquante pour le processus élu qui l'effectue. En cas de blocage, il y a réordonnancement et un nouveau processus prêt est élu. Concrètement, le compteur e(sem) du sémaphore est décrémenté d'une unité. Si la valeur du compteur devient négative, le processus est bloqué.

```
void P(semaphore sem) {
    masquer_it;
    e[sem] = e[sem] - 1;
    if (e[sem]<0) {
        Mettre r dans f[sem];           // Mise en FIFO des processus bloqués
        Bloquer le processus r;         // r est le processus appelant
    };
    démasquer_it;
}
```

# *Les sémaphores : Libération*

## L'opération V(sem) :

- L'opération V(sem) a pour but de rendre un jeton au sémaphore. De plus, si il y a au moins un processus bloqué dans la file d'attente f(s) du sémaphore, un processus est alors réveillé. La gestion des réveils s'effectue généralement en mode FIFO. L'opération V est une opération qui n'est jamais bloquante pour le processus qui l'effectue.

```
void V(semaphore sem) {  
    masquer_it;  
    e[sem] = e[sem] + 1;  
    if (e[sem] <= 0) {          // il y a un processus en attente  
        Enlever r de f(sem);    // processus en tête de FIFO  
        Mettre r dans l'état ELU;  
    };  
    démasquer_it;  
}
```

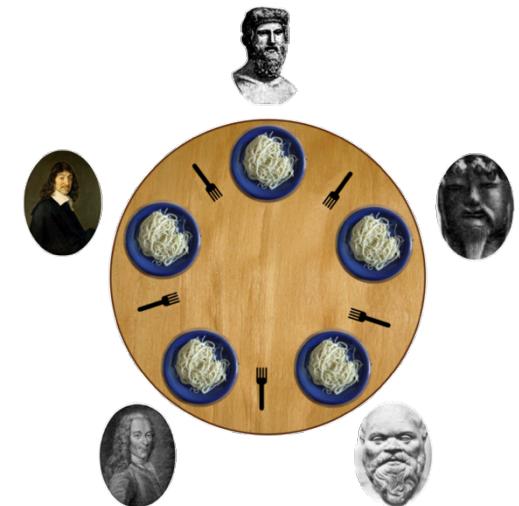
# *Problème des philosophes*

Un philosophe passe son temps à manger et à penser.

Lorsqu'un philosophe a faim, il tente de s'emparer des deux fourchettes qui sont à sa droite et à sa gauche, une après l'autre.

L'ordre importe peu. S'il obtient les deux fourchettes, il mange pendant un certain temps, puis repose les fourchettes et se remet à penser.

La question est la suivante : écrire un programme qui permette à chaque philosophe de se livrer à ses activités sans jamais être bloqué ou bloquer un autre philosophe ?



# *Les philosophes : 1<sup>o</sup> solution*

```
void Philosophe (int i) {  
    Penser;      //le philosophe pense  
    //Prendre fourchette gauche  
    Prendre_fourchette(i);  
    //Prendre fourchette droite  
    Prendre_fourchette(i+1);  
    Manger ;    // miam, miam, spaghetti  
    //Poser fourchette gauche  
    Poser_fourchette(i);  
    //poser fourchette droite  
    Poser_fourchette(i+1);  
}
```



# *Les philosophes : 2<sup>o</sup> Solution*

```
semaphore SP[n];      // tableau de sémaphores privés
etat e[n];            // tableau d'états des philosophes
void Philosophe (int i) {
    Penser;           //le philosophe pense
    //Action pour demander de manger
    P(mutex)
        if (e[i-1] !=M && e[i+1] !=M) {e[i]=M; V(SP[i]);}
        else e[i] = S;
    V(mutex)
    P(SP[i])
    Manger ; // miam, miam, spaghetti
    // Action pour arrêter de manger
    P(mutex)
        e(i) = P;
        if (e[i+1]==S && e[i+2] !=M) {e[i+1]=M; V(SP[i+1]);}
        if (e[i-1]==S && e[i-2] !=M) {e[i-1]=M; V(SP[i-1]);}
    V(mutex)
}
```

# *Les philosophes : Solution générale ...*

```
semaphore SP[n];//tableau de sémaphores privés v.i.=0
semaphore mutex;//v.i.=1
état e[n];           //tableau d'états v.i.=P

// il faut définir deux macros :
// vd(i) pour voisin droit
// et vg(i) pour voisin gauche

void tester (int num) {
    if (e[num]==S
        && e[vd(num)]!=M
        && e[vg(num)]!=M)  {
        e[num] = M;
        V(SP[num]);
    }
}
```

## *... Les philosophes : Solution générale*

```
void Philosophe (int i) {
    Penser();          //le philosophe pense
    //Action pour demander de manger
    P(mutex);
    e[i] = S;
    tester(i);
    V(mutex)
    P(SP[i])
    Manger();         // miam, miam, spaghetti
    // Action pour arrêter de manger
    P(mutex);
    e(i) = P;
    tester(vd(i));
    tester(vg(i));
    V(mutex);
}
```

# Producteur / Consommateur

Producteur()	Consommateur()
<b>Produire le message</b>	
<code>nvide = nvide - 1; si nvide = -1 alors attendre</code>	<code>nplein := nplein - 1 si nplein = -1 alors attendre</code>
<b>Déposer un message</b>	<b>Prélever un message</b>
<code>nplein := nplein + 1 si consommateur en attente alors réveiller consommateur</code>	<code>nvide := nvide + 1 si producteur en attente alors réveiller producteur</code>
	<b>Consommer le message</b>
<b>Recommencer</b>	<b>Recommencer</b>

# Inter-blocage

Inter-blocage dû à l'imbrication de ressources :

- Considérons à présent la situation suivante pour laquelle deux processus P1 et P2 utilisent tous les deux, deux ressources critiques R1 et R2 selon le code suivant.

Processus 0	Processus 1
$P(R1);$ $P(R2);$ <b>// Utilisation de R1 &amp; R2</b> $V(R2);$ $V(R1);$	$P(R2);$ $P(R1);$ <b>// Utilisation de R1 &amp; R2</b> $V(R1);$ $V(R2);$

# *Inter-blocage : conditions de réalisation*

---

4 conditions pour être en inter-blocage :

- Une ressource au moins est non partageable
- Occupation d'une ressource et attente d'une autre ressource
- Pas de réquisition : la libération d'une ressource ne peut se faire que par le détenteur
- Attente circulaire : il existe un **cycle d'attente** entre au moins deux processus.

Exemple d'attente circulaire :

- 2 processus et 3 ressources
- P1 : R1, R2, R3
- P2 : R3, R2, R1
- Scénario de blocage : P1/R1, P2/R3, P1/R2,
- P1 attend R3 et P2 attend R1

# *Inter-blocage : traitement*

---

## 4 méthodes de traitement :

- Détection / Guérison : algorithme basé sur un graphe représentant l'allocation des ressources et les processus en attente
- Prévention :
  - ▶ Éviter la circularité en imposant un ordre total
  - ▶ Protéger la prise de ressources par une mutex
- Evitement : à chaque demande d'allocation, un algorithme de sûreté est déroulé pour prévenir un inter-blocage à venir. Si tel est le cas, l'allocation est différée. Cette technique est basée sur la notion d'état sain. Ainsi, si l'état est malsain, on préfère ne pas attribuer la ressource : algorithme du banquier
- Autruche

Les 3 premières politiques sont très coûteuses alors on applique souvent la politique de l'autruche

# *Synchronisation*

---

Hypothèse :

- Les processus sont complètement asynchrones.

Objectifs : franchissement simultanée d'une barrière

- Bloquer un processus ou soi même,
- Activer un processus avec ou sans mémorisation.

Réalisation : Deux techniques

- Synchronisation directe : on nomme explicitement le processus,
- Synchronisation indirecte : on actionne un mécanisme qui agit sur d'autres processus.

# Sémaphores POSIX

---

Variable de type opaque `sem_t`

Opérations d'initialisation, d'incrémentation et de décrémentation

Deux possibilités :

- Sémaphores non-Nommés
  - ▶ Partageable par un processus créateur du sémaphore et ses fils
  - ▶ Implémentés sous Linux à partir du noyau 2.6
- Sémaphores Nommés
  - ▶ Partageable entre tous les processus : les mécanismes d'accès sur les fichiers sont applicables

```
#include <semaphore.h>
sem_t sem;
```

# *Unnamed semaphore*

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

value : valeur initiale du sémaphore

pshared <sup>1</sup> 0 : permet de partager le sémaphore avec ses fils

Exemple de création :

```
sem_t mon_semaphore
```

```
CHECK (sem_init(&mon_semaphore, 1, 1) == -1),
      "Problème durant sem_init");
```

# Utilisation du Sémaphore

`int sem_post(sem_t *sem);`

- incrémenté le sémaphore

`int sem_wait(sem_t *sem);`

- si value = 0 alors attente
  - ▶ soit value devient > 0
  - ▶ soit signal !!!

`int sem_trywait(sem_t *sem);`

- identique à sem\_wait mais non bloquant si value <= 0
  - ▶ retourne alors -1 et errno = EAGAIN

`int sem_getvalue(sem_t *sem, int *sval);`

- retourne valeur du sémaphore
- processus en attente
  - ▶ soit 0, soit valeur négative fonction du nbre

**AMBIGUË**  
la valeur indiquée n'est  
peut être plus la bonne

`int sem_destroy(sem_t *sem);`

- détruit un sémaphore existant

# *Named Semaphores ...*

```
#include <semaphore.h>
#include <sys/fcntl.h>
sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag, mode_t mode,
unsigned int value);
```

**name** : nom de fichier du sémaphore. Il sera rangé dans le répertoire système /dev/shm

**oflag = 0**

- On utilise la première forme d'appel
- Utilisation d'un sémaphore préalablement créé
- Si sémaphore n'existe pas alors retourne -1 et errno à ENOENT

## *... Named Semaphores ...*

---

**oflag = O\_CREAT (1) ou O\_CREAT | O\_EXCL (2)**

- (1) : crée si non existant sinon utilise l'existant et ignore mode et value
- (2) : crée un nouveau; si existe au préalable alors erreur (retourne -1 et errno à EEXIST)

Paramètres pour la seconde forme :

- mode : spécifie les protections (comme dans «open»)
- value : valeur initiale du sémaphore

Le sémaphore correspond à un fichier qui contient le pointeur sur la structure en mémoire : chaque sem\_open incrémentera le nombre de liens système sur le fichier

## *... Named Semaphores*

---

Les fichiers sémaphores sont créés dans le répertoire  
/dev/shm

Attention, une ouverture avec l'ancien nom crée un nouveau semaphore tant que le précédent existe encore.

**sem\_close()** : coupe le lien

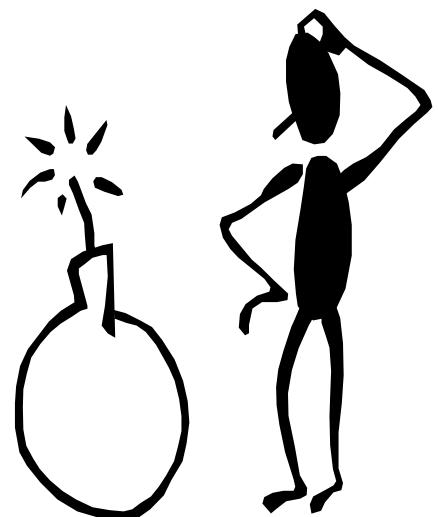
**sem\_unlink()** : décrémente le nombre de liens système sur le fichier et détruit le semaphore si le nombre de liens passe à 1

# Sémaphore et signal

Les primitives de gestion de sémaphore s'arrêtent sur réception d'un signal !!!

```
#include <semaphore.h>
#include <errno.h>

int sem_wait_restart(sem_t *sem) {
    int retval;
    while(((retval = sem_wait(sem)) == -1)
          && (errno == EINTR))
    ;
    return retval;
}
```



# *Partage de données & synchronisation : mutex*



Déclaration d'un thread de type mutex :

```
static pthread_mutex_t      mutex;
```

Utilisation de la mutex :

```
pthread_mutex_init(&mutex, NULL);  
pthread_mutex_lock(&mutex);  
pthread_mutex_unlock(&mutex);
```

## *Coiffeur endormi*

---

Cadre : Salon avec un fauteuil et N chaises.

Quand il n'y a pas de clients, notre coiffeur s'endort dans le fauteuil (chose qu'il commence par faire dès son arrivée au matin).

Le client doit réveiller notre coiffeur à son arrivée. S'il y a des clients et qu'il y a une chaise vide alors il attend son tour (éveillé).

Si le salon est plein, le client repart sans se faire coiffer.

Proposez une solution sans produire aucune situation de concurrence.

## PARTIE 4 : MÉMOIRE PARTAGÉE

# *Shared Memory*

---

Partage de l'espace d'adressage en vue d'une IPC (Communication InterProcess) optimale.

Cette solution est utile pour des processus ayant des droits d'accès différents :

- Processus appartenant à des utilisateurs différents
- Pour une application appartenant à un même utilisateur, il faut privilégier les threads.

Approche de mise en œuvre :

- On utilise un fichier «mappé et partagé»,
- Les données sont écrites dans le fichier par l'intermédiaire de l'espace d'adressage réservé, sont accessibles à l'ensemble des processus mappés sur ce fichier.
- La protection repose sur les droits d'accès applicables aux fichiers.
- Ce mécanisme repose sur les sémaphores.

## *Mise en oeuvre*

---

Le protocole d'accès à une zone mémoire partagée se fait en plusieurs étapes :

- Ouverture (avec création éventuelle) d'un objet mémoire partagé : pseudo-fichier. Cette opération crée un lien sur le fichier.
- Allocation et mappage d'une zone mémoire sur l'objet mémoire
- Exploitation de la zone mémoire en partage
- Démappage
- Clôture
- Suppression du lien : quand le nombre de liens est à 1, l'objet mémoire est détruit.

# *Fonctions de base*

---

## Ouverture : **shm\_open()**

- Ouvre un objet mémoire partagé et retourne un descripteur de fichier

## Etalonnage : **ftruncate()**

- Taille le fichier (objet mémoire) en un multiple de pages mémoire

## Mappage : **mmap()**

- Mappe l'objet en mémoire

## Déconnexion : **munmap()**

- Démappa l'objet en mémoire

## Fermeture : **shm\_close()**

## Suppression : **shm\_unlink()**

- Détruit l'accès au fichier

# Ouverture

```
int shm_open(const char *name, int oflag, mode_t  
mode);
```

- name : nom du fichier mémoire (dans `/dev/shm`)
- oflag : (comme pour sémaphore)
  - ▶ ou logique entre `O_RDONLY`, `O_RDWR`, `O_CREAT`, `O_EXCL` (exclue), ...
- mode : seulement si `O_CREAT`
  - ▶ syntaxe dans `/usr/include/sys/mode.h`

```
#include <sys/mman.h>  
#include <sys/mode.h>  
  
int fd;           // file descriptor  
fd = shm_open("maZone",  
              O_CREAT | O_RDWR,  
              S_IRWXU)
```

# Mappage

```
caddr_t mmap(caddr_t addr, size_t len, int prot, int flags, int  
filedes, off_t off)
```

- addr : adresse de début de la région (0 : attribution par le système)
- len : taille de la région mappé en multiple de pages
- prot : PROT\_NONE, PROT\_READ, PROT\_WRITE, PROT\_EXEC
- flags:
  - ▶ MAP\_SHARED partage des modifications
  - ▶ MAP\_PRIVATE modifications privées
  - ▶ MAP\_FIXED interprète rigoureusement les adresses !
- filedes : file descriptor (créé par `shm_open()`)
- off : offset dans le fichier

```
pageSize = sysconf(_SC_PAGE_SIZE);
```

```
CHECK(ftruncate(fd, pageSize), "Problème ftruncate");
```

```
virtAddr = mmap(0, pageSize, PROT_WRITE, MAP_SHARED,  
fd, 0);
```

# Utilisation

---

On utilise l'adresse de la région mappée comme un tableau d'octets :

virtAddr[index\_dans\_1\_objet\_mémoire\_partagé]

Exemple :

| virtAddr[i]='a';

**int msync(caddr\_t addr, size\_t len, int flags);**

addr :      virtAddr

len :      pageSize

flags :

MS\_SYNC cohérence synchrone

MS\_ASYNC les maj sont planifiées

MS\_INVALIDATE oblige nouvelle consultation (cache)

# *Clôture*

---

Il est nécessaire d'effectuer les opérations nécessaires à la libération des espaces mémoires et la clôture des objets mémoires :

```
int munmap(caddr_t addr, size_t len);  
int shm_close(int filedes);  
int shm_unlink(const char *name);  
    ▶ comme pour sémaphore
```

```
munmap(virtAddr, pageSize);  
close(fd);  
shm_unlink("maZone");
```

## *Exercices didactiques*

---

Tester la mise en œuvre d'une zone commune avec un processus écrivain et un processus lecteur.

Utiliser le partage en mémoire pour implémenter le problème des philosophes et spaghetti : on partage l'état des philosophes.

Mettre en œuvre l'algorithme du producteur / consommateur en utilisant la zone mémoire comme une FIFO de messages.



## PARTIE 5 : *SYSTÈME DE FICHIERS*

également la banalisation des E/S en fichiers

# *Système de fichiers*

---

Un système de fichier est une structuration hiérarchique (répertoire / fichier) d'un espace disque :

- Découpage de l'espace en deux parties :
  - ▶ Espace de gestion des index (moyen d'accès aux données)
  - ▶ Espace de stockage des données
- Appels système pour créer / accéder, lire / écrire, ...

Comment gérer plusieurs disques appartenant à une même machine ?

- Une première solution consiste à installer un système de fichiers sur chaque disque et de les gérer séparément (cas du DOS, VMS & NT).
- Une deuxième solution consiste à **monter** l'arborescence d'un disque sur une autre arborescence de disque (cas d'UNIX). Différents systèmes de fichiers cohabitent dans la même arborescence.

## *Accès concurrents à un fichier*

---

Utiliser un sémaphore sur le fichier : trop contraignant

Verrouiller une zone d'un fichier : deux processus appartenant à des utilisateurs différents peuvent alors lire en même temps deux zones différentes d'un même fichier.

Le verrou peut être soit **exclusif** ou **partagé** (seuls les appels partagés seront acceptés).

Un appelant peut rester bloqué (s'il le souhaite) tant que le verrou n'est pas levé.

# *Fichiers & Arborescence (1)*

---

Un fichier UNIX est une suite d'octets au niveau le plus bas (taille quelconque) : banalisation de la notion de fichier.

Nom de fichier :

- Aucune restriction sur les noms de fichiers (N'importe quel caractère).
- Les fichiers commençant par un point servent à la configuration de l'environnement.

La notion de répertoire (un fichier structuré en une suite de couples fichier / adresse) permet d'introduire une hiérarchie entre les fichiers :

- Regroupement de fichiers et de répertoires, concernant une même entité (personne, projet, application, ...) dans un même répertoire.

Chemin d'accès à un fichier :

- Absolu : chemin complètement spécifié depuis la racine (/).

Exemple : `/home/eleves/truc/.cshrc`

- Relatif : chemin spécifié depuis le répertoire courant.

Exemple : `bin/exemple`

# *Fichiers & Arborescence (2)*

---

Répertoires particuliers :

- . & ..
- **Working directory** : répertoire de travail ou répertoire courant.
- **Home directory** : à chaque utilisateur correspond un répertoire privé qui est son répertoire de travail par défaut : il est automatiquement positionné sur ce répertoire lors de la connexion.

Classification des fichiers selon leur structure :

- **régulier (-)** : fichier texte ou exécutable
- **répertoire (d)** : un fichier contenant d'autres fichiers,
- **device (c/b)** : imprimante, terminal, disque dur, floppy, lecteur de bande, disque numérique, ...
- **lien (l)**
- **socket (s)**
- **fifo (p)**
- **rien** : le trou noir (/dev/null)

# Répertoires d'un système UNIX

/bin	Commandes utilisateur de base
/sbin	Commandes d'administration
/dev	Fichiers spéciaux pour les E/S sur périphériques
/etc	Configuration et administration système
/lib	Bibliothèques système
/include	Fichiers d'entête système
/var	Journaux, Comptabilité & Queues
/tmp	Fichiers temporaires
/usr ou Extension du système	/usr/bin : Exécutables /usr/lib : Librairies /usr/include : Fichiers entête système /usr/man : Manuels d'aide en ligne /usr/var : Comptabilité & Queues
/home	Répertoires utilisateurs
/opt ou /usr/local	Applications spécifiques

# *Implémentation du système de fichiers*

Tous les disques UNIX possèdent la même structure (représentation sous forme d'un tableau):

- Bloc 0 : contient le code d'initialisation de l'ordinateur (non utilisé);
- Bloc 1 (**superbloc**) :
  - ▶ Nombre d'i-nodes
  - ▶ Nombre de blocs de données,
  - ▶ Pointeur sur la liste des blocs libres.
- La table des **i-nodes** : un ensemble contigu de blocs numérotés :
  - ▶ I-node 1 : liste des blocs défectueux
  - ▶ I-node 2 : racine du disque
- Les **blocs de données** : espace restant utilisé pour le stockage du contenu des fichiers et des répertoires. Les blocs d'un fichier sont non contigus en général.

La structure **i-node** permet le contrôle et la localisation des données pour un fichier :

- Informations de contrôle : mode, nb-liens, UID, GID, taille, Dates
- Informations de localisation : liste des dix premiers blocs, adresse du bloc d'indirection simple (double et triple).

## *Implémentation du système de fichiers (2)*

---

Un répertoire est une suite non triée d'entrées de 16 octets (System V) où chaque entrée est une paire d'informations :

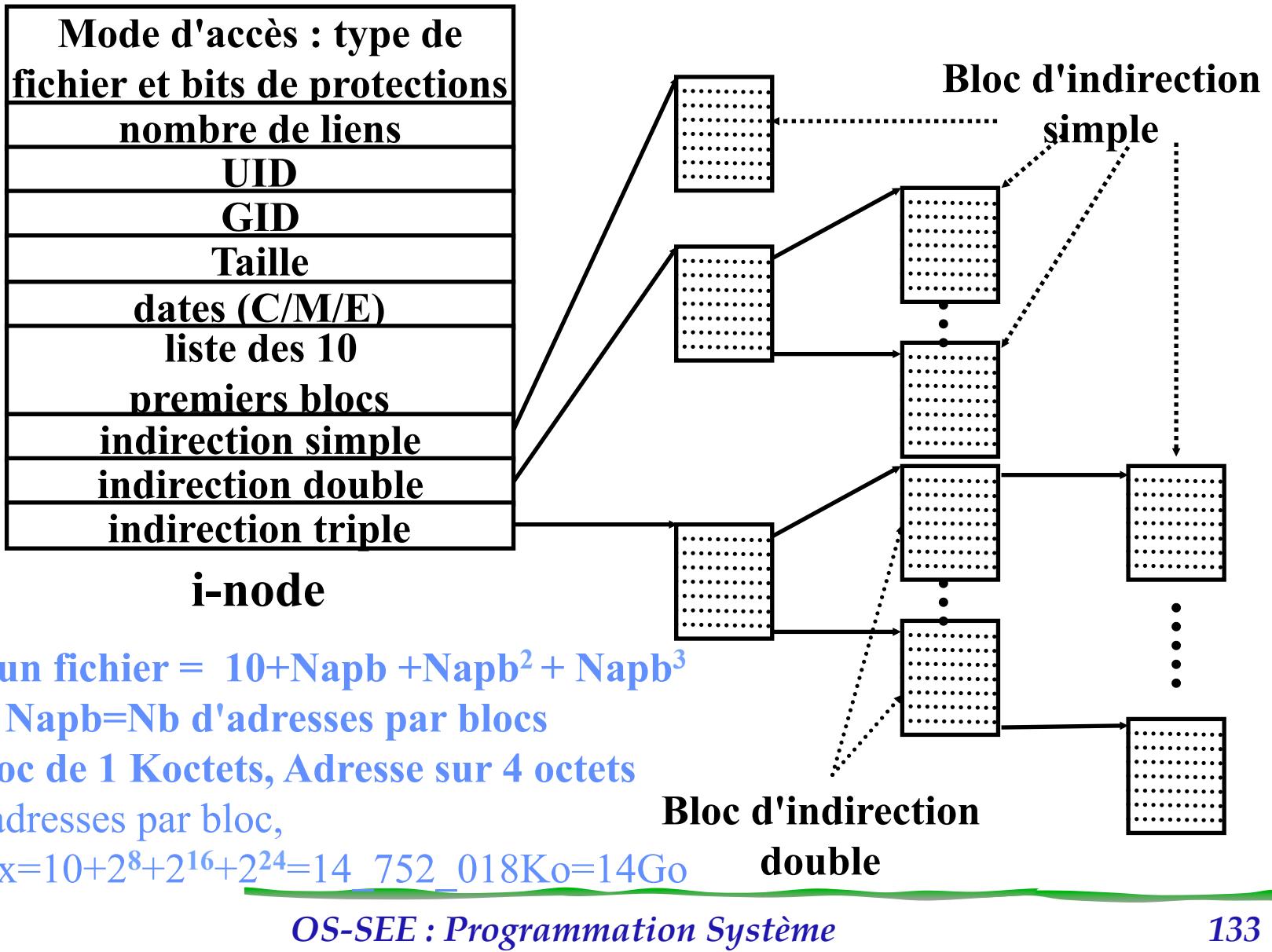
- Numéro d'i-node : 2 octets
- Nom de fichier : 14 octets

Par des comparaisons successives des répertoires composant un chemin d'accès à un fichier avec les entrées d'un répertoire, on finit par trouver le fichier (son i-node) ou non : le chemin relatif est plus rapide.

L'i-node est utilisé comme index dans la table des i-nodes sur disque.

La structure de l'i-node est chargée en mémoire dans la table des fichiers ouverts et pointée par la structure utilisateur du processus.

# Implémentation d'un i-node



# Processus & Fichiers

---

L'appel système `open(<nom de fichier>, <mode d'accès souhaité>)` permet de tester l'**existence** du fichier et de **contrôler la cohérence** du mode d'accès avec les permissions du fichier spécifié.

Cet appel renvoie un entier positif (descripteur de fichier, à partir de 3) puisque par défaut tout programme possède trois descripteurs (0 pour **STDIN**, 1 pour **STDOUT**, et 2 pour **STDERR**).

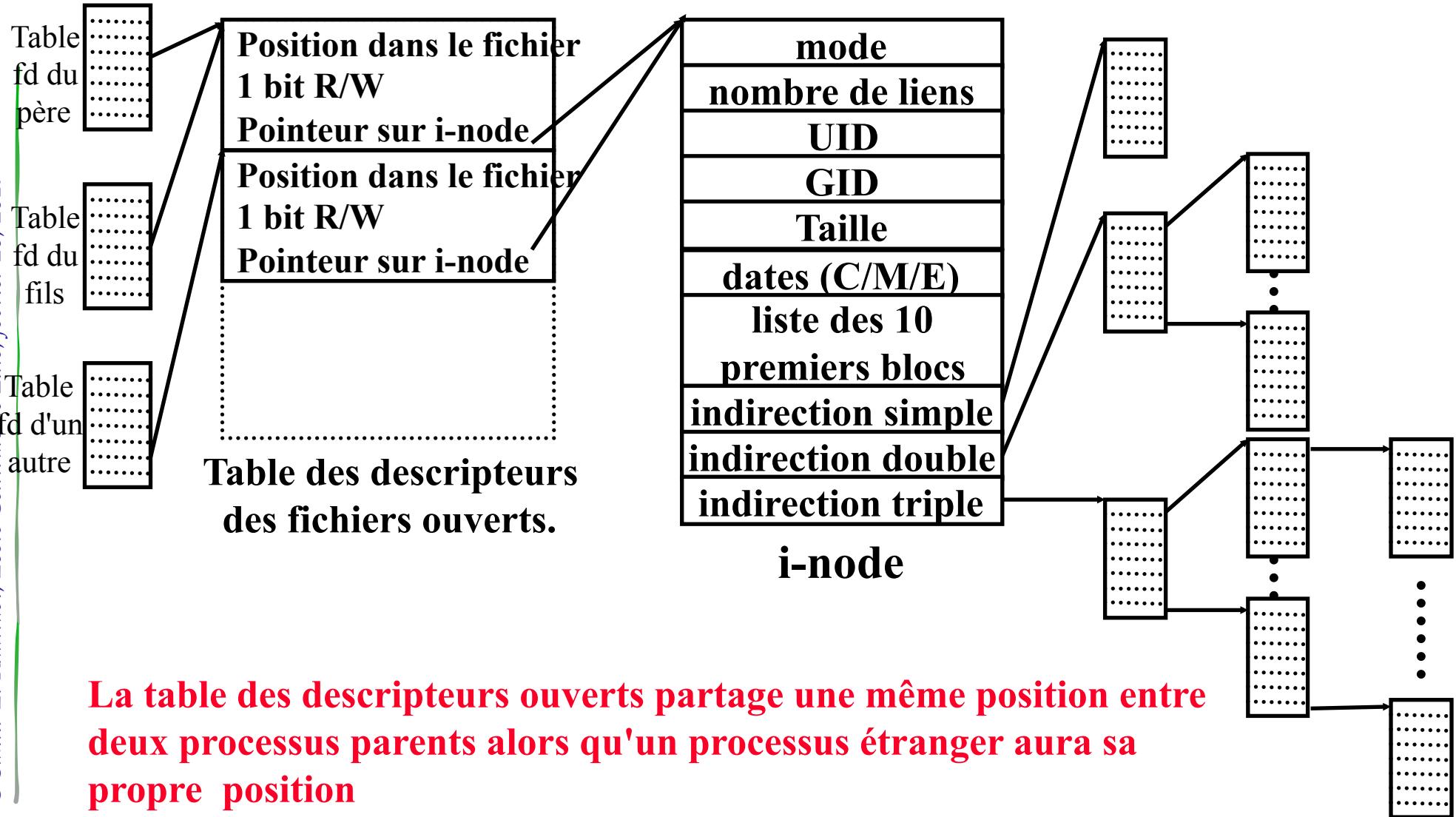
Deux modes d'accès sont possibles pour les fichiers :

- Chemin absolu,
- Chemin relatif : variable d'environnement **PWD** (ou **cwd**) dans segment pile.

L'appel système `link()` permet à des utilisateurs de partager un même fichier.

L'appel système `unlink()` coupe le lien et supprime l'i-node si le nombre de liens est à 1.

# *Exemple d'un appel de lecture sur fichier*



## *Améliorations du système de fichiers de Berkeley*

---

Les noms de fichiers sont stockés sur 255 caractères.

POSIX fournit les appels **OPENDIR**, **REaddir**, **CLOSEDIR** afin de rendre les programmes compatibles sur les deux systèmes.

La deuxième modification concerne la division d'un disque en un groupe de cylindres appelé partition ou disque logique. Un disque logique dispose de la même structure qu'un disque physique : superbloc, table d'i-nodes et blocs de données.

La troisième amélioration se situe au niveau de la taille bloc : en effet, elle peut être différente d'une partition à une autre. Il est ainsi possible d'avoir des partitions avec un bloc de 512 octets pour les petits fichiers et des partitions avec un bloc de 4 Ko pour les grands fichiers

# *Principe d'Entrées / Sorties*

---

La gestion des périphériques est intégrée dans le système de fichiers en considérant les unités comme des fichiers spéciaux.

A chaque unité est associé un fichier dans `/dev` :

- `/dev/ttyxx` : terminal virtuel n° xx
- `/dev/lp` : port parallèle
- `/dev/hdx`: disque dur n° x (a, b, c, ....)
- `/dev/sdx` : disque SCSI n° x (a, b, c, ....)

Accessibilité similaire aux fichiers : on utilise les appels **READ** et **WRITE** comme on le ferait avec n'importe quel fichier.

## *Principe d'Entrées / Sorties (2)*



Les mécanismes de protection habituels sont appliqués.

Deux types de fichiers spéciaux :

- Fichiers bloc destinés aux périphériques de stockage : adressage d'un bloc (lire / écrire un bloc)
- Fichiers caractère destinés au périphériques de flux : écran, réseau, clavier, ....

Les fichiers de type caractère ne peuvent pas utiliser l'adressage direct d'un caractère.

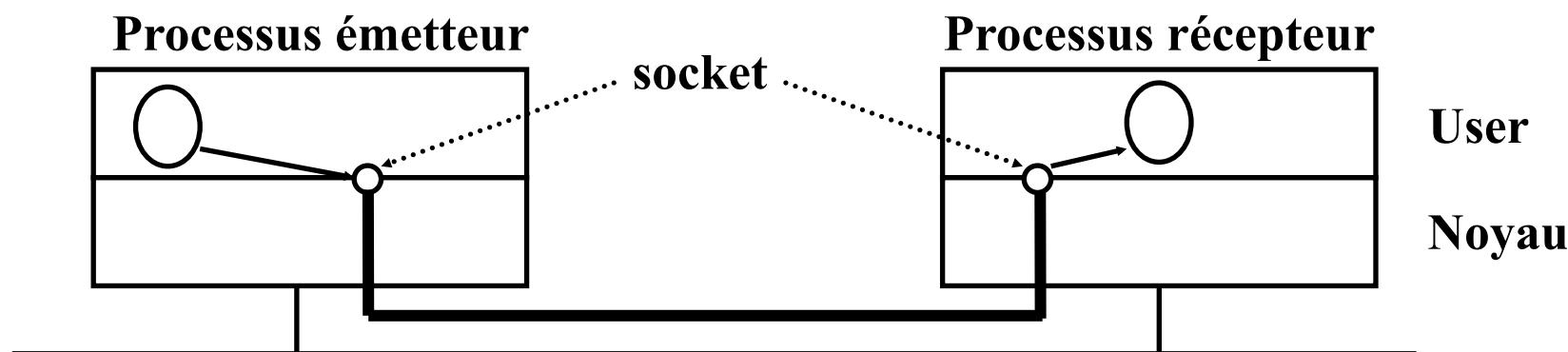
## Principe d'Entrées / Sorties (3)

Les périphériques de type caractère disposent de caractéristiques spécifiques :

- Clavier : Effacement = DEL / BaackSpace, ...
- Ecran : Nombre de lignes, nombre de colonnes, ...
- Réseau : Taille d'une trame, ...

Il existe pour chaque type de périphérique, un ensemble d'appels systèmes permettant de modifier ces valeurs.

Un autre modèle d'IPC que les tubes est introduit par Berkeley : notion de socket (cf cours programmation réseaux TCP/IP).



## *PARTIE 6 : ANNEXES*

## *PERMISSIONS*

# Permissions (1)

Bases :

- Un utilisateur est membre d'un seul groupe.
- Un fichier appartient à un seul utilisateur et à un seul groupe.

Droits d'accès sur les fichiers :

- 3 permissions **rwx** applicables à 3 classes d'utilisateurs
  - r** droit de lecture du fichier,
  - w** droit d'écriture du fichier,
  - x** droit d'exécution
- Les classes d'utilisateurs sont :
  - user** le propriétaire du fichier
  - group** les utilisateurs du même groupe que l'utilisateur,
  - other** tous les autres utilisateurs.

Droits d'accès sur les répertoires :

- Les clés **rwx** perdent leur signification habituelle :
  - r** droit de lecture et donc de listage (**ls**),
  - w** droit d'écriture sur le répertoire (**rm**),
  - x** droit de traversée du répertoire (**cd**).

# Permissions(2)

Expression des permissions :

- Les permissions d'un fichier (répertoire) sont exprimés sous la forme de 10 caractères :

d/l/-	r/-	w/-	x/-	r/-	w/-	x/-	r/-	w/-	x/-
user			group			other			

**d** : répertoire     **c** : device     **b** : disk     - : fichier régulier     **l** : lien  
**r/-** : read / not read     **w/-** : write / not write     **x/-** : execute / not execute

Protégez vos données :

- Utilisez **-rw-----** pour les fichiers,
- Utilisez **drwx--x--x** pour les répertoires.
- La commande **umask 066** permet de créer les fichiers et les répertoires avec les protections ci-dessus : sécurité et souplesse.
- La valeur de umask est exprimée en base **octale**.
- Pour obtenir les protections, on **XOR** **666** pour les fichiers et **777** pour les répertoires.

# Permissions(3)

Table XOR :

A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0

Expression des permissions en base octale :

- 3 permissions d'accès (**rwx** : 1 chiffre en base 8),  
    **0** : non autorisé    **1** : autorisé,
- 3 classes d'utilisateurs (3 chiffres en base 8),  
    Exemple :    **-rwxrw-r--** correspond à **764**

La commande **newgrp** :

- Elle permet à l'utilisateur de changer de groupe et d'accéder ainsi aux fichiers et répertoires appartenant à des utilisateurs de ce groupe.
- Syntaxe : **newgrp groupe**                                      (l'utilisateur doit appartenir au groupe).

# Permissions(4)

## La commande **chown** :

- Cette commande permet de changer le propriétaire et éventuellement le groupe du fichier.

`chown <user>[:<group>] <liste des fichiers à céder>`

## La commande **chgrp** :

- Cette commande permet de changer le groupe du fichier.

`chgrp <group> <liste des fichiers à céder>`

## La commande **chmod** :

- Cette commande permet de changer les protections des fichiers.

`chmod <perms en octal> <liste des fichiers à modifier>`

`chmod <liste des perms> <liste des fichiers à modifier>`

Où chaque permission s'exprime comme suit :

`[ugoa][+-][rwxst]` ou `[ugoa]=[une combinaison de rwxst]`

## *PARTIE 6 : ANNEXES MODES D'EXÉCUTION*

# *Modes d'exécution (1)*

Différents types d'exécution :

- Interactif ou foreground : **fg**.
- Asynchrone ou background : **bg**.
- Différée : **at**.
- Queue : **batch**.
- Cyclique : **crontab**.

Exécution interactive :

- C'est la plus courante. La commande est lancée au terminal et rend le contrôle au terminal une fois qu'elle s'est terminée.

Exécution asynchrone :

- On l'utilise pour des commandes dont le résultat n'est pas immédiat.  
*Exemple : compilation d'un source.*
- Cette exécution se fait par le lancement de la commande **et** du caractère **&**.  
*Exemple : make test &*

# *Modes d'exécution (2)*

---

## Exécution asynchrone (suite) :

- Une commande asynchrone continue à envoyer ses messages sur l'écran. Il faut donc les rediriger.
- A une exécution asynchrone est associé un **n° de job** et un **n° de process**.

## Exécution différée :

- Cette exécution se fait par la commande **at**.
- Elle permet d'exécuter une commande à une date ultérieure et une heure ultérieure.
- Les résultats sont renvoyés par mail.
- La commande **atq** liste les exécutions différées.
- La commande **atrm** supprime une exécution différée.
- L'utilisation est autorisée ou non par l'administrateur.

# *Modes d'exécution (3)*

## Exécution cyclique :

- Cette exécution se fait par la commande **crontab**.
- Elle permet d'exécuter une commande de manière **périodique**.
- La spécification de la commande cyclique se fait dans un **fichier cron**.
- Le fichier cron contient une **liste** des commandes cycliques (**une commande par ligne**).
- Chaque **ligne** est composée de **six champs**.

Format de la ligne : minute heure jour-du-mois mois jour-semaine cde  
Une étoile indique que le champ n'est pas spécifié.

- La commande **crontab file** soumet le fichier cron au **daemon** associé.
- La commande **crontab -l** affiche le fichier cron.
- La commande **crontab -r** supprime le fichier cron.
- Les messages doivent être **redirigés**. Par défaut, le résultat est retourné par messagerie électronique.
- L'utilisation est autorisée ou non par l'administrateur.

# *Modes d'exécution (4)*

## Exécution Batch :

- Cette exécution se fait par la commande **batch**.
- La commande est placée dans une **file d'attente**. L'exécution se fait quand la commande atteint la **tête de la file** (FIFO).
- Par défaut, une seule file d'attente commune à tous les utilisateurs.

## Les commandes bg et fg :

- **bg n° job** : exécute le job en background.
- **fg n° job** : exécute le job en foreround.

# *PARTIE 6 : ANNEXES*

## *HISTORIQUE DES GÉNÉRATIONS DE CALCULATEURS*

## *Historique : les débuts*

---

En 1850, Charles Babbage (1792-1781) crée la première machine analytique :

- Cette machine n'a jamais fonctionné,
- Lancement de plusieurs études.

En 1945, Von Newman propose la première machine considérée comme un ordinateur :

- 1ère génération (1945 – 1955) : **tube à vide et tableaux d'interrupteurs**,
- **Pas de système d'exploitation** : langage machine basé sur le **câblage des entrées et sorties**,
- Les programmeurs n'ont pas accès à la machine (**cartes perforées**),
- ENIAC (université de Pennsylvanie).

# *Historique : la 2<sup>e</sup> génération (1950 – 1965)*

Apparition du transistor en 1950.

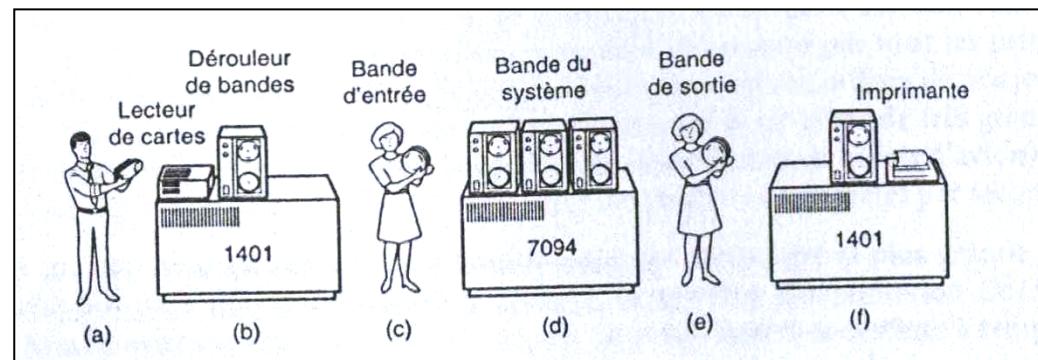
Premier ordinateur commercialisé.

Création d'un chargeur (loader) fin des années 1950.

Les programmeurs n'ont toujours pas accès à la machine :

- Programme utilisateur encodé sur cartes perforées en Assembleur ou FORTRAN,
- Regroupement des programmes par **lot** sur bande magnétique : **batch**,
- Résultats sur **imprimante**.

Mainframe : IBM 7094 doté du premier système IBM SYS.



# *Historique : 3<sup>e</sup> génération (1965 – 1980) (1)*

---

Apparition du circuit intégré

IBM présente son System /360 :

- Plusieurs séries seront produites (370, 4300, 3080, 3090),
- Performances variables et compatibilité.

Le temps inhérent aux E/S était très important :

- À peine 20% du temps de traitement était réellement affecté au calcul,
- Il fallait donc occuper le processeur pendant les opérations d'E/S,
- Apparition alors de la **multiprogrammation** basée sur le **partitionnement de la mémoire**. On distingue alors deux types de tâches programmables : celles liées aux E/S et celles liées au calcul.

Simultaneous Peripheral Operation On Line (SPOOL) :

- Plusieurs tâches sont chargées en mémoire,
- Le système **commute** entre les tâches sur blocage d'E/S.

DIGITAL propose le premier mini-ordinateur le DEC PDP-1.

## *Historique : 3<sup>e</sup> génération (1965 – 1980) (2)*

---

Darthmouth College et le MIT inventent le temps partagé :

- Introduction des **terminaux** : accès direct pour les utilisateurs,
- Le système de Dartmouth ne permettait de faire que du BASIC,
- CTTS (Compatible Time Sharing System) eut un franc succès auprès des scientifiques.

MIT, Bell et General Electric s'associent pour créer une deuxième génération de systèmes d'exploitation :

- **MULTplexed Information and Computing Service** : système multi-utilisateurs,
- MULTICS était écrit en **PL/I** et le compilateur est arrivé tardivement : échec mais de nombreuses innovations,
- MIT arrivera à produire une version opérationnelle en 1980, commercialisé par HP qui a racheté les activités informatiques de GE.
- En 1969, Ken Thompson (ancien programmeur de Bell) eut l'idée de réécrire une version simplifiée, mono-utilisateur de MULTICS sur un PDP-7 appelée **UNICS** rebaptisée **UNIX** (cf. plus loin).

## *Historique : 4<sup>e</sup> génération (Depuis 1980) (1)*

---

Apparition des circuits LSI (Large Scale Integration circuits).

Intel sort le 8080 (premier processeur à 8 bits), Gary Kildall conçoit un système d'exploitation orienté disque :

- **CP/M** (Control Program for Micro computers) créant ainsi le premier micro-ordinateur équipé d'un disque 8 pouces,
- Kildall fonde **Digital Research** pour commercialiser le CP / M.

En 1980, IBM crée l'IBM PC.

- Après refus de Kildall de concevoir un système pour son PC, IBM se tourne vers Bill Gates,
- Gates achète les droits du DOS (Disk Operating System) proposé par Seattle Computer Products (50000 \$ selon les rumeurs).

Gates fonde Microsoft et engage Tim Paterson (créateur de DOS)

- Après quelques modifications souhaitées par IBM, la première version de **MS-DOS** était née.

## *Historique : 4<sup>e</sup> génération (Depuis 1980) (2)*

---

En 1983, avec l'apparition du PC/AT (Intel 80286), MS-DOS était bien implanté alors que CP/M vivait ses derniers moments.

- La version initiale était très primitive et se voit enrichir avec des mécanismes empruntés à UNIX,
- Plusieurs versions vont voir le jour accompagnant plusieurs générations de processeurs Intel (80386, 80486).

S. Jobs d'Apple invente l'Apple :

- Premier ordinateur avec un SE à interface graphique,
- Lisa (première version) : trop chère, échec,
- Arrivée du Macintosh : un grand succès.

Microsoft contre attaque timidement en lançant plusieurs versions de Windows au dessus du DOS :

- Version 1 & 2 : interface simple, échec,
- Version 3.1 : enfin une version qui a marché,
- Windows 95/98 : 1<sup>e</sup> versions Windows indépendantes du DOS.

## *Historique : 4<sup>e</sup> génération (Depuis 1980) (3)*

---

Microsoft propose enfin un système totalement réécrit par David Cutler (ancien concepteur de VMS) et son équipe :

- Windows NT : New Technology, 32 bits,
- Une certaine compatibilité avec les différentes versions du DOS,
- Beaucoup de concepts empruntés à VMS,
- NT 4 est la première version considérée comme stable,
- En 1999, arrivée de NT 5 (baptisé 2000) pour unifier Win 98 et NT.
- En 2003, arrivée de Windows XP.
- En 2007, Windows Vista

UNIX se dote d'un système de fenêtrage appelé X Window autour du standard X11 produit par le MIT :

- Gestionnaire de fenêtres avec une IHM tel que Motif, OpenLook, DCE.

Milieu des années 80, la croissance des réseaux a amené le développement des systèmes d'exploitation distribués :

- Système classique doté en plus des primitives de communication réseau adaptées.

# *Historique : quelques chiffres (1)*

	1ère	2ème	3ème	4ème
Nom de la machine	L'ENIAC	PDP-1	PDP-8/1	LSI-11
Composants	Tube électronique	Transistor	Circuit intégré	Microprocesseur
Année	1950	1960	1965	1976
Encombrement	Un bâtiment	Une armoire	Un rack	Une carte
Dimensions	15*15*6 m	2.4*0.75*1.8 m	60*60*60 cm	22*25*1.2 cm
Consommation (watts)	150 000	2 500	250	50
Nombre d'accès mémoire par seconde	80 000	200 000	600 000	900 000
Prix (\$)	?	120 000	10 000	600

## *Historique : quelques chiffres (2)*

---

1964, OS/360 IBM :

- 1 Million d'instructions

1975, Multics et VMS de DEC :

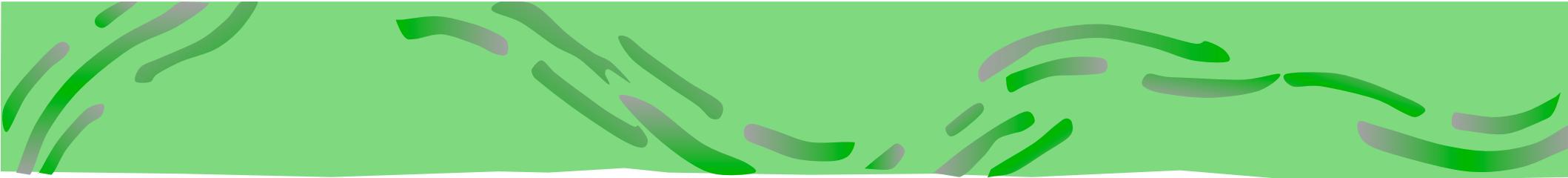
- 1 Million d'instructions

1981, DOS : 13 Ko

1987, DOS : 78 Ko

1988,

- Unix BSD 4.2 : 500 000 instructions
- OS/VS IBM : 5 Millions d'instructions
- Multics et MVS : 20 Millions d'instructions



## *PARTIE 6 : ANNEXES*

### *IMPLÉMENTATION D'UNE RESSOURCE*

### *CRITIQUE*

Dekker  
Peterson  
TSL

# *Problème de Dekker*

---

Les seules instructions indivisibles sont :

- l'affectation d'une valeur à une variable,
- le test de la valeur d'une variable.

On dispose de deux processus P0, P1.

Chacun des processus dispose d'une section critique en exclusion mutuelle

## Dekker (1): Variable de verrou & attente active

On protège la section critique avec une seule variable verrou :

- verrou==0 : libre
- verrou==1 : occupé

```
int verrou = 0;  
void processus() {  
    while (1) {  
        while (verrou==1);           // attente active  
        verrou=1-verrou;            // allocation  
        // Traitement de section critique  
        verrou=1-verrou;            // libération  
        // Suite du traitement (hors section)  
    }  
}
```



ne respecte pas le 1

## Dekker (2) : Alternance stricte

On protége la section critique avec une variable commune :

- tour = i ssi le processus Pi est autorisé à s'engager dans sa section critique (i vaut 0 ou 1)

```
int tour = 0;
```

```
void processus(int num) {
```

```
    while (1) {
```

```
        while (tour!=num); // attente active du tour
```

```
// allocation effectuée par l'autre processus
```

```
// Traitement de section critique
```

```
        tour=1-num; // allocation à l'autre
```

```
// Suite du traitement (hors section)
```

```
}
```

```
}
```



ne respecte pas le 3

## Dekker (3) : une variable par processus (1)

On introduit une variable booléenne par processus

- Si  $\text{etat}[i]==1$  alors Pi se trouve dans sa section critique
- Pi peut seulement lire  $\text{etat}[1-i]$

```
int etat[2] = {0, 0};
```

```
void processus(int num) {
    while (1) {
        while ( $\text{etat}[1-\text{num}]==1$ );      // attente active
        etat[num]=1;                      // allocation
        // Traitement de section critique
        etat[num]=0;                      // libération
        // Suite du traitement (hors section)
    }
}
```

ne respecte pas le 1

## Dekker (4) : une variable par processus (2)

On introduit une variable booléenne par processus

- Si  $\text{res}[i]==1$  alors Pi **demande à entrer** dans sa section critique
- Pi peut seulement lire  $\text{res}[1-i]$

```
int res[2] = {0, 0};  
void processus(int num) {  
    while (1) {  
        res[num]=1;          // réservation  
        while (res[1-num]==1); // attente active  
        // Traitement de section critique  
        res[num]=0;          // libération  
        // Suite du traitement (hors section)  
    }  
}
```

Exclusion mutuelle est garantie mais

- Ne respecte pas la **condition 2** :
  - ▶ Les deux processus peuvent s'engager dans une boucle infinie

## *... Dekker (4): Nouvelle solution*

```
int res[2] = {0, 0};  
void processus(int num) {  
    while (1) {  
        while (1) {  
            res[num]=1;  
            if (res[1-num]==1) res[num]=0;  
            else break;  
        }          // attente active avec réservation/annulation  
        // Insertion du traitement de section critique  
        res[num]=0;           // libération  
        // suite du traitement (hors section)  
    }  
}
```



## Dekker (5) : Solution

Utilisation de trois variables communes :

```
int res[2] = {0, 0}; int tour=0;
void processus(int num) {
    while (1) {
        while (1) {
            res[num]=1;
            if (res[1-num]==0) break;
            else {
                if (tour==num) continue;
                res[num]=0;
                while (tour==1-num);
            }
        }
        // Traitement de section critique
        tour=1-num; res[num]=0;           // libération
        // Suite du traitement (hors section)
    }
}
```

# *Solution de Peterson*

La même solution que Dekker en plus simple.

```
int res[2] = {0, 0};  
int tour=0;  
void processus(int num) {  
    while (1) {  
        res[num]=1;  
        tour=num;  
  
        while (tour==num && res[1-num]==1);  
        // Traitement de section critique  
        res[num]=0;      // libération  
        // suite du traitement (hors section)  
    }  
}
```

# *TSL (Test And Set Lock): TSL câblé*

Programmation très complexe

- utilisation en dernier ressort d'un mécanisme **câblé** qui réalise une forme élémentaire d'exclusion

Il s'agit d'une instruction élémentaire TAS (Test And Set) sur une cellule mémoire. Le test et l'affectation forment donc une instruction atomique (indivisible).

```
void TSL(cellule m) {  
    // Bloquer l'accès à la cellule m  
    // Lire la valeur de la cellule m  
    if (m==0) m=1;                      // allocation  
    // Libérer l'accès à la cellule m  
}
```

## *TSL soft*

```
int TSL(int * verrou) {  
    int temp;  
    masquer_it;  
    temp = *verrou;  
    *verrou = 1;  
    return temp;  
    démasquer_it;  
}
```

Utilisation : Mutex est initialisée à 0 (libre)

```
while ( TSL(&Mutex) );  
// Traitement section critique  
Mutex = 0;
```