# Systèmes d'exploitation
# L3, S6

*Janvier 2009*

Vincent Rodin
Université de Bretagne Occidentale
vincent.rodin@univ-brest.fr
http://www.lisyc.univ-brest.fr/pages_perso/rodin/FTP/Enseignements/L3/Systeme





## Plan

▷ Introduction et historique
▷ Notion de processus
▷ Exclusion mutuelle
▷ Communication inter–processus : IPC
▷ Communication inter–processus : tubes
▷ Communication inter–processus : signaux
▷ Activités et modes d'ordonnancement
▷ Programmation multi-threads
▷ Systèmes de fichiers : structure, enregistrement, accès





## Systèmes d'exploitation (SE) : introduction

### Introduction

*Rappels généraux sur les systèmes d'exploitation*
*Historique*
*Structures des systèmes d'exploitation*





## Rôle et nature d'un système d'exploitation

Son rôle :

▷ Interfacer le matériel et les services applicatifs
▷ Piloter le matériel, fournir des abstractions
▷ Gérer la mémoire (réelle/virtuelle)
▷ Gérer les fichiers
  ◇ Disques/systèmes de fichiers
▷ Gérer la multi-programmation
  ◇ Partage du temps, des ressources, communication
  ◇ **Processus** $\simeq$ programme en cours d'exécution

## Rôle et nature d'un système d'exploitation

Un système d'exploitation est :

▷ mono–tâche ou multi–tâches,

▷ mono–utilisateur ou multi–utilisateurs.

- ◇ UNIX est un système multi–tâches, multi–utilisateurs.
- ◇ Windows NT4 est un système multi-tâches, mono-utilisateur.





## Rôle et nature d'un système d'exploitation

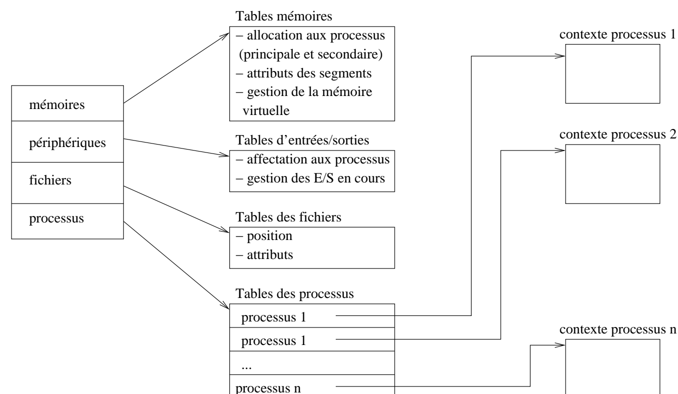
▷ Ce que c'est principalement
- ◇ Un noyau (ou plusieurs) contrôlant la machine
- ◇ Un ensemble de points d'entrée pour le solliciter
  - ◦ Les *appels-systèmes*

▷ Ce n'est pas *exclusivement*
- ◇ Un environnement de développement
- ◇ Un interpréteur de commandes
- ◇ Une interface graphique





## Structure de donnée interne








## Un peu d'Histoire, UNIX

▷ 1969 : K. Thompson & D. Ritchie (AT&T, Bell Labs) ancêtre d'UNIX

▷ 1973 : Première version d'UNIX en C (portable, sources disponibles)

▷ 1978 : UNIX version 7 devient commercial !

▷ 1980 : Variante BSD (Université de Californie, Berkeley)

▷ 1983 : AT&T UNIX System V

▷ 1984 : Naissance du projet GNU, Free Software Fundation

▷ 1987 : AT&T et Sun unifient BSD et System V (SunOS); HP-UX, AIX

▷ 1990 : System VR4 apporte de nouveaux standards d'unification

▷ 1991 : OSF/1, Open Software Fundation

▷ 1992 : Solaris de Sun basé sur System VR4

▷ 1993 : Linux

`http://www.levenez.com`

## Un peu d'Histoire, MSDOS, Windows

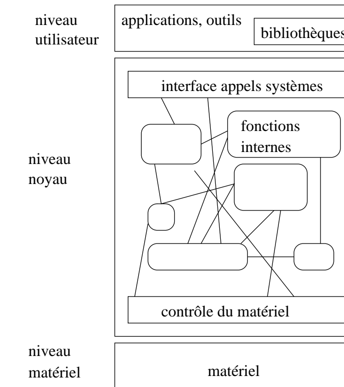
- ▷ 1974 : G. Kildall (Digital Research), CP/M pour 8080 et Z80
- ▷ 1981 : IBM PC, portage de CP/M (PCDOS)
- ▷ 1981 : Microsoft achète QDOS (inspiré de CP/M, T. Paterson, Seattle Computer Products) commercialisé sous le nom MSDOS 1.0
- ▷ 1985 : Windows 1.0, Microsoft
- ▷ 1988 : DR-DOS, Digital Research
- ▷ 1990 : Windows 3.0, Microsoft (énorme succès commercial)
- ▷ 1991 : MSDOS 5.0, Microsoft (gestion subtile de la mémoire)
- ▷ 1993 : WindowsNT, Microsoft
- ▷ 1995 : Windows95, Microsoft (MSDOS habilement dissimulé)
- ▷ 1998... : Windows98, Windows2000, WindowsXP, ... Microsoft





## Structures internes

Systèmes monolithiques (0S/360, Unix 4.3BSD, Linux2)



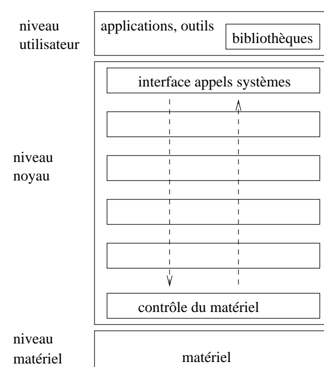

Problèmes : maintenance, évolution





## Structures internes

Systèmes en couches (Multics, OpenVMS)
⟶ hiérarchiser les fonctions :
la couche $i$ communique avec les couches $i-1$ et $i+1$.



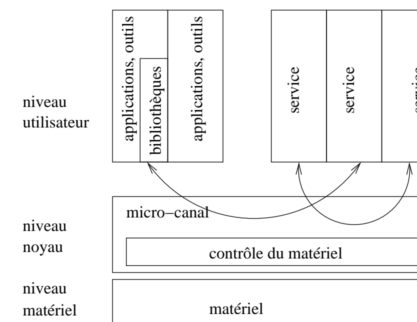

Problèmes : performances





## Structures internes

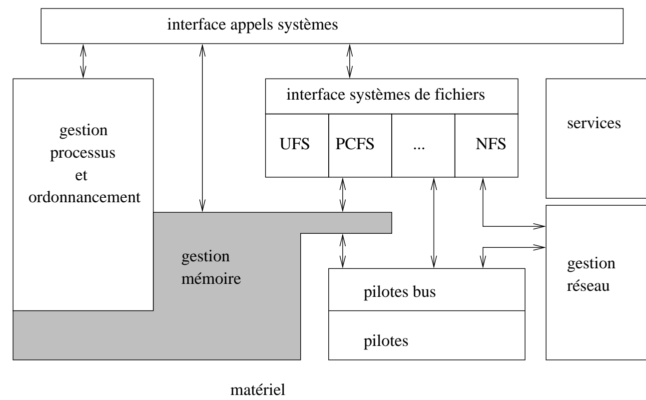
Systèmes avec micro–noyaux (Windows NT4)
⟶ ne conserver dans le noyau que les fonctions essentielles
⟶ interactions par passages de messages (client-serveur)




Services :
- Mémoire Virtuelle,
- Système de fichiers,
- Réseaux, ...

Avantages : extensibles, flexibles, portables
Inconvénients : performances (construction–décodage message, synchronisation)

## Exemple de structure interne : UNIX SVR4

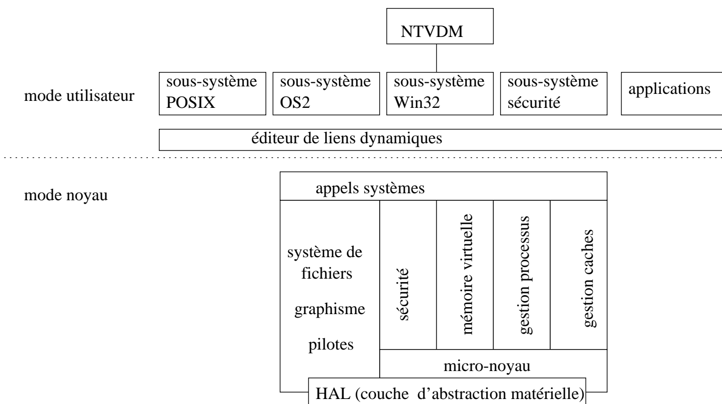



Monolithique





## Exemple de structure interne : Windows NT4




Micro–noyaux





## Mode d'exécution

Au niveau matériel,
les micro-processeurs fournissent plusieurs modes d'exécution.

Selon le mode choisi,
certains accès ou instructions sont autorisés ou non.

Exemples d'opérations liées à un "privilège":

- ▷ accès aux registres du MMU,
- ▷ gestion des interruptions,
- ▷ changement de mode d'exécution.

Les modes d'exécution sont un support indispensable
à la mise en œuvre d'un système d'exploitation fiable.





## Exécution du système d'exploitation

Dans la plupart des systèmes d'exploitation,
les processus peuvent s'exécuter dans deux modes différents :

- ▷ le **mode utilisateur** (*user mode*) :
  le processus n'accède qu'à son espace d'adressage (ses propres instructions et données) et exécute des instructions du programme correspondant ;
- ▷ le **mode noyau** (*kernel mode*) :
  le processus exécute des instructions en mode privilégié.

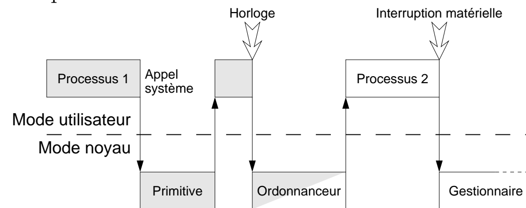
Le passage du mode utilisateur au mode noyau peut être provoqué :

- ▷ par un appel système,
- ▷ par une interruption extérieure.

## Exécution du système d'exploitation

▷ **Mode utilisateur**
- ◇ Mode de fonctionnement "*normal*" d'un processus
- ◇ Traitements dans son propre espace d'adressage
- ◇ Pas de risque majeur (sauf pour lui)

▷ **Mode noyau**
- ◇ Accès au matériel, à la mémoire physique
- ◇ Gestion des processus








## ⟹ Appel système ≠ fonction

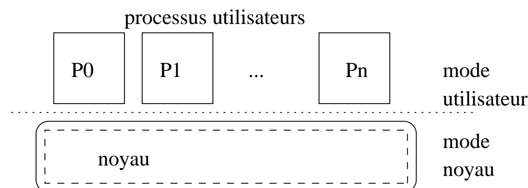
▷ **Appel système**
- ◇ Permet d'accéder aux services du système
  - ○ Passage en mode noyau
  - ○ Vérification si processus autorisé
- ◇ Mécanisme d'interruption très couteux en temps
  - ○ Sauvegarde des données du processus ...

▷ **Fonction**
- ◇ Calcul dans l'espace utilisateur
- ◇ Encapsulation des appels systèmes
  - ○ Services de plus haut niveau
  - ○ Regrouper plusieurs invocations en une seule
  - ○ Rôle de la bibliothèque standard du C





## Exécution d'un système à noyau séparé



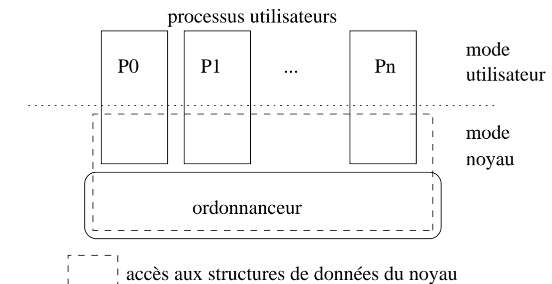

Le noyau est une entité particulière, unique, opérant en mode noyau.

Son comportement n'est pas régi par les mêmes règles que les autres processus du système.





## Exécution d'un système à noyau partagé




Chaque processus utilisateur peut passer en mode noyau et exécuter le code du noyau, dans son contexte.

Le noyau est une partie de chaque processus utilisateur.

En mode noyau, les processus ont accès à leur contexte, et aux données du noyau.

## La notion de processus

### La notion de processus

**L'exemple d'UNIX**

*Définition*
*Cycle de vie*
*Identification (PID)*
*Terminaison*
*Attente de la fin d'un processus fils*





## Processus

▷ Définition : Instance d'un programme en cours d'exécution

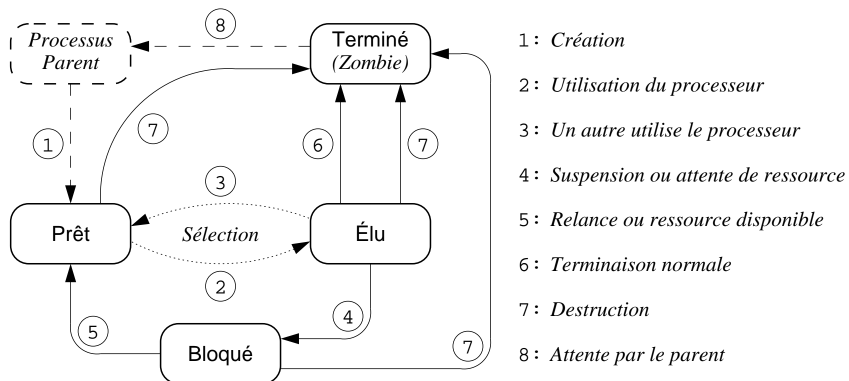
▷ Propriétés :

- ◇ identifiant unique (**PID**) `getpid()`
- ◇ priorité `[get/set]priority()`
- ◇ identité de l'utilisateur (**UID**) `getuid()`
  - $\Longrightarrow$ droits d'accès aux fichiers, aux périphériques,...
- ◇ variables d'environnements `getenv()`
  - $\Longrightarrow$ programme : exécution = fonction(données externes)
- ◇ comportement vis à vis des signaux





## Cycle de vie d'un processus




1 : *Création*
2 : *Utilisation du processeur*
3 : *Un autre utilise le processeur*
4 : *Suspension ou attente de ressource*
5 : *Relance ou ressource disponible*
6 : *Terminaison normale*
7 : *Destruction*
8 : *Attente par le parent*





## Identification d'un processus

Le `PID` (*Process IDentifier*)

▷ Le type `pid_t` ($\simeq$ entier)

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

▷ Récupérer l'identifiant (`man 2 getpid`)

```
#include <unistd.h>
pid_t getpid(void);     // processus courant
pid_t getppid(void);    // processus parent
```

▷ Toujours un résultat valide

## Création d'un processus

Appel système `fork()` (`man 2 fork`)

▷ `#include <unistd.h>`
`pid_t fork(void);`

Retourne :

`-1` si erreur, ou
le `PID` du processus créé (si l'on est dans le processus père),
`0` (si l'on est dans le fils)

▷ **Réplique quasi-identique du processus d'origine**

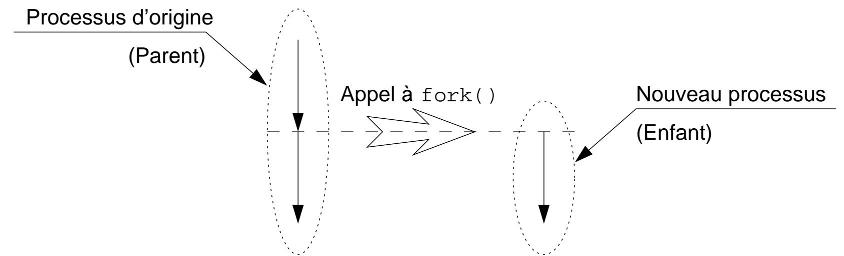
◇ Espace d'adressage (identique mais **indépendant**)
◇ Environnement, descripteurs de fichiers, signaux ...





## Création d'un processus

Dédoublement du flot d'exécution








## Création d'un processus

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

int main(void)          /* creationFils.c */
{
 pid_t result;

 fprintf(stderr,"Begin %d\n",getpid());
 result=fork();
 switch(result)
  {
    case -1: fprintf(stderr,"Pb with fork !\n"); /* Echec */
             break;
    case  0: /*  Processus enfant : result = 0 */
             fprintf(stderr,"Child : %d (parent=%d)\n",getpid(),getppid());
             break;
    default: /* Processus parent : result = PID du processus fils */
             fprintf(stderr,"Parent : %d (child=%d)\n",getpid(),result);
  }

 fprintf(stderr,"End %d\n",getpid());
 return EXIT_SUCCESS;
}
```

```
$ ./prog
Begin 1325
Parent : 1325 (child=1326)
Child : 1326 (parent=1325)
End 1325
End 1326
$
```





## Création d'un processus

Deux causes d'échec possibles pour `fork()` :

▷ Mémoire insuffisante pour la table des processus
`errno` vaut `ENOMEM` $\Longrightarrow$ Erreur grave, système saturé !

▷ Mémoire insuffisante pour dupliquer le processus
`errno` vaut `EAGAIN` $\Longrightarrow$ La mémoire peut se libérer
$\Longrightarrow$ recommencer

```
pid_t result=fork();   ===>   #include <errno.h>
                              ...
                              pid_t result;
                              do {
                                  result=fork();
                              } while ((result==-1) && (errno==EAGAIN));
```

## Terminaison d'un processus

Fonction `exit()` (`man 3 exit`)

▷ `#include <stdlib.h>`
`void exit(int status);`

▷ Vidage des tampons et fermeture des flux

▷ Possibilité de transmettre un code de terminaison
- Une valeur entière ; généralement `0` signifie OK
- Constantes `EXIT_SUCCESS` et `EXIT_FAILURE` de `stdlib.h`





## Attente de la fin d'un processus fils

Appels systèmes `wait()` et `waitpid()` (`man 2 wait`)

▷ Attente du signal `SIGCHLD` d'un enfant vers le parent

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

▷ Mise à jour de l'entier pointé par `status`

▷ Options de `waitpid()` (`0` ou combinaison (OR) bit à bit)

▷ Retourne le `PID` de l'enfant terminé ou
`0` si `WNOHANG` dans les options et pas d'enfant terminé ou
`-1` si enfant inconnu, mauvaise options, interruption (`EINTR`→ relancer)





## Attente de la fin d'un processus fils

```
#include <sys/types.h>             /* attenteFils.c */
#include <sys/wait.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <stdio.h>

int main(void)
{
 pid_t result, endPid;
 int   code, etat;

 do {
      result=fork();
 } while ((result==-1) && (errno==EAGAIN));
```





## Attente de la fin d'un processus fils

```
/* attenteFils.c (suite) */

 switch(result)
  {
   case -1: fprintf(stderr,"Pb with fork !\n");
            break;
   case  0: sleep(1);
            fprintf(stderr,"code d'exit : ");
            scanf("%d",&code);
            exit(code);
            break;
   default: do {
                 endPid=waitpid(result,&etat,0);
            } while ((endPid==-1) && (errno==EINTR));
            fprintf(stderr,"code d'exit du fils : %d\n",WEXITSTATUS(etat));
            break;
  }
 return EXIT_SUCCESS;
}
```

⇑ Macro

```
$ ./prog2
code d'exit : 3
code d'exit du fils : 3
$
```

## L'exclusion mutuelle

### L'exclusion mutuelle

*Objectif de l'exclusion mutuelle*
*Accès concurrents*
*Section critique*
*Exclusion mutuelle par attente active*
*Exclusion mutuelle sans attente active*





## Objectif de l'exclusion mutuelle

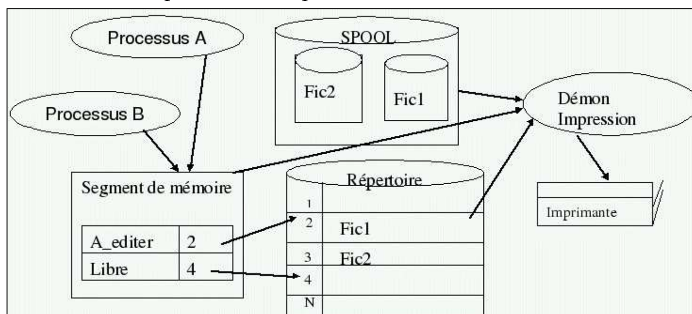
- L'exclusion mutuelle a pour but de **limiter l'accès** à une ressource à un ou à un nombre donné de processus.
- Ceci concerne, par exemple, un fichier de données que plusieurs processus désirent mettre  jour.
  - L'accès  ce fichier doit être réservé à un seul utilisateur pendant le moment où il le modifie, autrement son contenu risque de ne plus être cohérent.
- On appelle ce domaine d'exclusivité, une **section critique**.





## Accès concurrents – exemple (1)

- Fonctionnement d'un spool d'impression:
  - Quand un processus veut imprimer un fichier, il doit placer le nom de ce fichier dans le répertoire de spool.
  - Un autre processus, le démon d'impression, vérifie périodiquement s'il faut imprimer des fichiers. Si c'est le cas, il les imprime et retire leur nom du répertoire de spool.



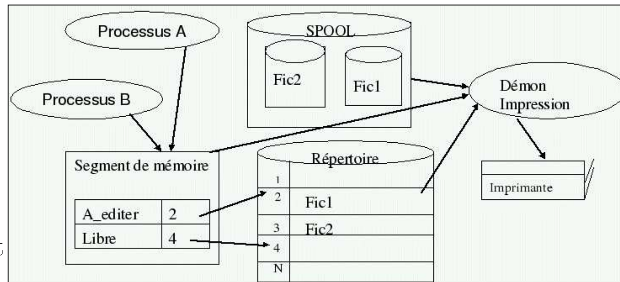





## Accès concurrents – exemple (2)

- On fait les hypothèses suivantes:
  - Le répertoire d'impression a un très grand nombre d'emplacements numérotés de `0` à `N`.
  - Chacun de ces emplacements peut contenir le nom d'un fichier à imprimer.
  - Il existe deux variables partagées qui sont sauvegardées dans un segment de mémoire partagée et accessible à tous les processus :
    - `A_éditer` qui pointe sur le prochain fichier à imprimer,
    - `Libre` qui pointe sur le prochain emplacement libre du répertoire.

## Accès concurrents – exemple (3)




▷ Situation courante
- ◇ La position 1 est libre,
- ◇ Les positions 2 et 3 sont occupées,
- ◇ Le prochain emplacement libre est le 4

▷ Imaginons maintenant que pratiquement au même moment, les processus A et B veulent placer chacun un fichier dans la file d'impression.





## Accès concurrents – exemple (4)

▷ Il risque de se produire la chose suivante:
- ◇ Le processus A lit la variable `Libre` et mémorise la valeur 4.
- ◇ Une interruption horloge se produit et l'usage du processeur est retiré au processus A pour être alloué au processus B.
- ◇ Le processus B lit également la variable `Libre` qui vaut toujours 4 et place le nom du fichier B à l'emplacement 4. Ensuite le processus B incrémente `Libre` qui passe à 5 puis poursuit ensuite son travail.
- ◇ Au bout d'un certain temps le processus A est relancé par l'ordonnanceur et reprend son exécution.
- ◇ Le processus A place son nom de fichier à imprimer à la position 4 du répertoire en effaçant le nom du fichier qui y avait été mis par B.
- ◇ Le processus A calcule ensuite la place libre suivante, obtient 5 et place cette valeur dans `Libre`.





## Accès concurrents – exemple (5)

▷ Le répertoire d'impression est alors dans un état **cohérent** et le démon d'impression ne détectera pas la moindre anomalie. Pourtant, **le fichier du processus B ne sera jamais imprimé**.

▷ Les situations de ce type, où deux processus ou plus
- ◇ lisent et/ou écrivent des données partagées,
- ◇ et où le résultat dépend de l'ordonnancement des processus,

sont qualifiées **d'accès concurrents**.





## Section critique (1)

▷ La **partie de programme** où peut se produire un conflit d'accès s'appelle une **section critique**.

▷ Pour éviter les conflits d'accès il faut un moyen d'interdire la lecture ou l'écriture des données partagées à plus d'un processus à la fois.
- ◇ En d'autres termes, il faut un **mécanisme d'exclusion mutuelle** qui empêche les autres processus d'accéder à un objet partagé si cet objet est en train d'être utilisé par un processus.

▷ Toutes les ressources dites partagées au sein d'un système d'exploitation ne peuvent être accédées qu'en **EXCLUSION MUTUELLE**.
- ◇ Le choix des primitives qui assurent l'exclusion mutuelle est un des points fondamentaux de la conception d'un système d'exploitation.

## Section critique (2)

▷ Le problème des conflits d'accès est résolu, si on peut s'assurer que **2 processus ne sont jamais en section critique en même temps**.

▷ Cette condition est suffisante pour éviter les conflits d'accès mais elle ne permet pas aux processus de coopérer.

▷ **Quatre conditions** sont dans ce cas nécessaires:

1. Deux processus ne peuvent être en même temps en section critique.
2. Aucune hypothèse ne doit être faite sur les vitesses relatives des processus et sur le nombre de processeurs.
3. Aucun processus suspendu en dehors d'une section critique ne doit bloquer les autres processus.
4. Aucun processus ne doit attendre trop longtemps avant d'entrer en section critique.





## Exclusion mutuelle par attente active

▷ Il existe plusieurs méthodes pour réaliser l'exclusion mutuelle par **attente active**

▷ Seules quelques unes de ces méthodes seront présentées dans ce cours:

1. Les variables de vérouillage ... ne marche pas !
2. l'instruction TSL





## Attente active: Les variables de vérouillage (1)

▷ Soit une variable (ou verrou) partagée, unique, qui a initialement la valeur 0 et dont le contenu permettra de savoir s'il y a déjà un processus en section critique:

- ◇ 0 : aucun processus n'est en section critique
- ◇ 1 : il y a déjà un processus en section critique

▷ Un processus doit tester ce verrou avant d'entrer en section critique.

- ◇ Si le verrou vaut 0, le processus le met à 1 et entre en section critique.
- ◇ Si le verrou est déjà à 1, le processus attend qu'il repasse à 0.





## Attente active: Les variables de vérouillage (2)

```
Processus 1               |      Processus 2
-----------               |      -----------
                          |
while (verrou==1);        |      while (verrou==1);
verrou=1;                 |      verrou=1;
section_critique();       |      section_critique();
verrou=0;                 |      verrou=0;
```

▷ Cette approche comporte un vrai défaut entre

- ◇ le moment où le processus lit le verrou et s'apperçoit qu'il est zéro: `while (verrou==1);`
- ◇ et le moment où il peut le mettre à 1: `verrou=1;`

▷ Il y a un risque que plusieurs processus entre en même temps en section critique !

## Attente active: L'instruction machine TSL (1)

- Si on peut faire une **lecture suivie d'une écriture de façon INDIVISIBLE**, alors l'algorithme de passage en section critique devient plus simple.
- En réalité, la plupart des ordinateurs dispose d'une instruction TEST AND SET LOCK (TSL)
  - Elle charge le contenu d'un mot mémoire dans un registre (**lecture**)
  - Puis met une valeur non nulle à cette adresse (**écriture**)
- Les opérations de lecture et d'écriture du mot sont garanties comme étant **indivisibles** (**atomiques**).





## Attente active: L'instruction machine TSL (2)

- Pour illustrer l'instruction TSL, nous nous servirons de la variable partagée `drapeau`:
  - Lorsque `drapeau` vaut `0`, tout processus peut lui donner la valeur `1` au moyen de l'instruction `TSL`.
  - Ce processus peut ensuite rentrer en section critique.
  - Lorsqu'il a terminé, il remet `drapeau` à `0` grâce à l'instruction `MOVE` classique.





## Attente active: L'instruction machine TSL (3)

- La solution comprend deux sous-programmes
  - `Entrer_SC` et `Sortir_SC`
  - Ecrits dans un langage assembleur fictif.

```
           Entrer_SC                       |   Sortir_SC
           ---------                       |   ---------
                                           |
boucler:                                   |
  tsl registre,drapeau // registre<-drapeau |
                       // ET drapeau=1      |   mov drapeau,#0
  cmp registre, #0     // si registre !=0   |   ret
  jnz boucler          // ==> vers boucler  |
  ret                                      |
```

**Rappel: TSL est atomique !**





## Attente active: L'instruction machine TSL (4)

```
Processus 1           |      Processus 2
-----------           |      -----------
                      |
Entrer_SC();          |      Entrer_SC();
section_critique();   |      section_critique();
Sortir_SC();          |      Sortir_SC();
                      |
```

## Exclusion mutuelle sans attente active

▷ L'exclusion mutuelle avec attente active consomme beaucoup de **temps processeur**.

- ◇ Le processus exécute une **boucle infinie** jusqu'à ce qu'il soit autorisé à entrer en section critique.
- ◇ Cette approche doit en général être évitée.

▷ Pour éviter cela, les systèmes d'exploitation disposent de primitives IPC (Inter Process Communiction) qui se bloquent au lieu de perdre du temps UC, lorsqu'elles ne sont pas autorisées à entrer en Section Critique.

- ◇ **Les sémaphores**





## Exclusion mutuelle sans attente active: Les sémaphores

▷ Généralités sur les sémaphores

▷ Mécanismes fournis par le noyau

▷ Utilisation classique des sémaphores

▷ Le problème des producteurs et des consommateurs





## Généralités les sémaphores (1)

▷ Un sémaphore est un mécanisme empêchant deux processus d'accéder simultanément à une ressource partagée.

- ◇ Sur les voies ferrées, un sémaphore empêche deux trains d'entrer en collision sur un tronçon de voie commun.
- ◇ Sur les voies ferrées comme en informatique, les sémaphores ne sont qu'indicatifs...

**il faut les prendre en compte !**





## Généralités les sémaphores (2)

▷ Un **sémaphore binaire** n'a que deux états:

- ◇ 0 verrouillé (ou occupé).
- ◇ 1 déverrouillé (ou libre).

▷ Un **sémaphore général** peut avoir un très grand nombre d'états car il s'agit d'un compteur qui:

- ◇ Décroît d'une unité quand il est acquis (verrouillé).
- ◇ Croît d'une unité quand il est libéré (déverrouillé).
- ◇ Quand il vaut zéro
  - - un processus tentant de l'acquérir doit attendre qu'un autre processus ait augmenté sa valeur
  - - car il ne peut jamais devenir négatif.

## Généralités les sémaphores (3)

▷ L'accés à un sémaphore se fait généralement par deux opérations:
- ⋄ **P** pour l'acquisition en néerlandais: Proberen, tester.
- ⋄ **V** pour la libération: Verhogen, incrémenter.

▷ Un moyen mnémotechnique:
- ⋄ **P**(uis-je) accéder à une ressource
- ⋄ **V**(as-y) la ressource est disponible





## Mécanismes fournis par le noyau (1)

▷ Les sémaphores doivent être fournis par le noyau qui, lui, peut:
- ⋄ Partager des données entre les processus.
- ⋄ Exécuter des opérations indivisibles (ou atomiques).
- ⋄ Allouer l'UC à un processus prêt quand un autre processus se bloque.

▷ La notion de sémaphore est implémentée dans la plupart des systèmes d'exploitation.
- ⋄ Il permet une solution efficace à la plupart des problèmes d'exclusion.
- ⋄ Ce concept nécessite la mise en œuvre d'une variable, le sémaphore, et de deux opérations atomiques associées `P` et `V`.





## Mécanismes fournis par le noyau (2)

Le mécanisme des sémaphores a été inventé par **Dijkstra**
pour résoudre le problème de *l'exclusion mutuelle des processus*

**Sémaphore S : Valeur $\geq$ 0 + File d'attente**

**Opérations :**

⋄ **P(S)** : Si Valeur = 0
Alors Placer le processus en attente
Sinon Valeur ← Valeur - 1

⋄ **V(S)** : Si File d'attente $\neq \emptyset$
Alors Débloquer **un** processus en attente
Sinon Valeur ← Valeur + 1

**Remarque : V(S)** ne débloque pas toujours le 1er dans la file !
**File ⟹ boîte**





## Utilisation classique des sémaphores

**Protection d'une section critique :**

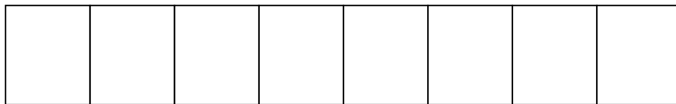
Soit le sémaphore **S** initialisé au départ avec Valeur = 1.

| Processus 1 : | | Processus 2 : |
|---|---|---|
| ..... | | ..... |
| **P(S)** | | **P(S)** |
| | ⇐ Section critique ⇒ | |
| **V(S)** | | **V(S)** |
| ..... | | ..... |

## Le problème des producteurs et des consommateurs (1)

▹ **Deux processus se partagent un tampon de données** de taille fixe (`N`) initialement vide:

- ◇ Un premier processus (le PRODUCTEUR) produit des données et les écrit dans le tampon.
- ◇ Un second processus (le CONSOMMATEUR) consomme les données du tampon en les lisant et en les détruisant au fur et à mesure de leur lecture.



N cases





## Le problème des producteurs et des consommateurs (2)

▹ Les problèmes surviennent lorsque:

- ◇ Le producteur veut mettre des informations alors que la **mémoire tampon est déjà pleine**.
  ⟹ Il devra attendre que le consommateur ait consommé
- ◇ De la même manière si le consommateur tente de retirer une information alors que la **mémoire tampon est vide**.
  ⟹ Il devra attendre que le producteur ait produit.





## Le problème des producteurs et des consommateurs (3)

▹ Synchronisation des processus producteur et consommateur grâce à des sémaphores:

- ◇ Le producteur et le consommateur doivent accéder de manière exclusive au tampon, le temps d'une lecture ou d'une écriture.
  ⟹ Un sémaphore d'exclusion mutuelle est donc nécessaire: `mutex`.
- ◇ Les ressources du producteur sont les emplacements vides du tampon qu'il est donc possible de matérialiser.
  ⟹ Un sémaphore `production` dont la valeur initiale correspond à la taille du tampon (`N`).
- ◇ Les ressources du consommateur sont les emplacements pleins du tampon qu'il est donc possible de matérialiser.
  ⟹ Un sémaphore `consommation` dont la valeur initiale est `0`.





## Le problème des producteurs et des consommateurs (4)

```
#define N 100

Sema mutex = 1;
Sema production = N;
Sema consommation = 0;

void producteur(void)
{
 while(1)
 {
  P(production);     // Une place vide en moins
  P(mutex);          // Entree section critique
  vider_case();      // ---- UTILISATION MEMOIRE PARTAGEE ----
  V(mutex);          // Sortie section critique
  V(consommation);   // Une place pleine en plus
 }
}
```

## Le problème des producteurs et des consommateurs (5)

```
void consommateur(void)
{
 while(1)
 {
  P(consommation);   // Une place pleine en moins
  P(mutex);          // Entree section critique
  vider_case();      // ---- UTILISATION MEMOIRE PARTAGEE ----
  V(mutex);          // Sortie section critique
  V(production);     // Une place vide en plus
 }
}
```





## Inter-Process Communication (IPC)

### Inter-Process Communication (IPC System V)

*Introduction*
*Points communs : clé, identifiant, contrôle*
*Les commandes Shell* `ipcs` *et* `ipcrm`
*Sémaphores*
*Segments de mémoire partagée*
*Files de messages*





## Introduction

**System V, oui mais IPC sur beaucoup de systèmes UNIX malgrè tout !**

IPC (Inter Processus Communication)
⟹ 3 mécanismes pour la communication et la synchronisation entre processus :

| | |
|---|---|
| ◇ les sémaphores | `#include <sys/sem.h>` |
| ◇ les mémoires partagées | `#include <sys/shm.h>` |
| ◇ les files de messages | `#include <sys/msg.h>` |

Ces mécanismes sont accessibles via des clés de type `key_t`.

Utilisation d'IPC : `#include <sys/types.h>`
`#include <sys/ipc.h>`





## Introduction

Pour chacun des 3 types d'objet,
une structure spécifique est disponible :

- ◇ `struct semid_ds` (ensemble de sémaphores)
- ◇ `struct shmid_ds` (segment de mémoire partagée)
- ◇ `struct msqid_ds` (file de messages)

Pour un objet donné, une structure de ce type contient notamment :
- les dates de dernières consultation et modification,
- les pids des processus ayant réalisé les dernières opérations sur l'objet,
- les droits d'accès à l'objet (via une structure `struct ipc_perm`),
- ...

## Clé d'IPC

**Clé d'IPC :** *nom externe* qui permet à un processus de retrouver un objet IPC donné.

⟹ Désignation du même objet IPC

⟹ Partage d'un objet IPC et donc communication via cet objet

**Obtention :** `key_t ftok(char *pathname, int proj);`

- Nom de fichier existant (`pathname`) connu de tous les processus.
- Identificateur de projet (`proj`) commun à tous les processus.

⇒ Clé d'accès à un IPC ou `-1` si problème...





## Identifiant d'IPC

### Création – Ouverture d'un IPC

**A partir d'une clé d'IPC**,
la **création** d'un nouvel objet ou l'**ouverture** d'un objet déjà existant se fait à l'aide d'un appel système `xxxget` (`semget`, `shmget`, `msgget`).

Un appel système de ce type retourne :

◇ un entier appelé **identifiant interne** de l'objet,

◇ `-1` en cas d'erreur...

**Un processus hérite des identifiants d'IPC connus par son père...**





## Identifiant d'IPC

### Création – Ouverture d'un IPC

`xxxget` permet, via un drapeau, de demander la création et/ou l'ouverture d'un objet désigné par sa clé.

**Drapeaux possibles :**

| | |
|---|---|
| `IPC_CREAT` | Ouverture si l'objet existe et création sinon |
| `IPC_CREAT \| IPC_EXCL` | Erreur si l'objet existe et création sinon |
| `IPC_CREAT` non indiqué | Ouverture si l'objet existe et erreur sinon |

Lors d'une création, il faut également indiquer les droits d'accès à l'IPC.





## Contrôle d'un IPC

Connaissant l'identifiant associé à un objet, `xxxctl` permet de :

◇ consulter et partiellement modifier la structure associée à cet objet (`semid_ds`, `shmid_ds` et `msqid_ds`),

◇ supprimer cet objet.

⟹ l'appel système `xxxctl` prend en paramètre une commande `cmd` :

| | |
|---|---|
| `IPC_STAT` | Consultation |
| `IPC_SET` | Modification |
| `IPC_RMID` | Supression |

En fonction des types d'IPC, d'autres commandes sont possibles...

## Les commandes Shell `ipcs` et `ipcrm`

`ipcs` permet d'obtenir des informations sur les IPCs du système

⟹ ◇ clés,
◇ identifiants,
◇ ...

`ipcrm` permet de détruire "à la main" un IPC ... peut être utile!





## Les sémaphores

**Création d'un tableau de sémaphores, ou récupération d'un tableau existant...**

`int semget(key_t key, int nsems, int semflg);`

- ◇ Retourne un identifiant de **tableau** de sémaphores, ou -1 si erreur
- ◇ `key` est la clé associée au tableau de sémaphores
- ◇ `nsems` est le nombre d'éléments du tableau (indices : `0...nsems-1`)
- ◇ `semflg` : drapeau d'ouverture.

  ⟹ OU binaire combinant :
  - le type d'ouverture (`IPC_CREAT` , `IPC_EXCL`) et
  - les droits d'accès au tableau (à préciser lors de la création)





## Les sémaphores

**Initialisation et contrôle d'un tableau de sémaphores**

```
int semctl(int semid, int semnum, int cmd,
           union semun {                   /* Si cmd est :    */
                int val;                   /* SETVAL          */
                unsigned short int *array; /* SETALL, GETALL  */
                struct semid_ds *buf;      /* IPC_STAT, IPC_SET */
           } arg);
```

Réalise l'opération `cmd` sur le tableau de sémaphores
ou sur le sémaphore d'indice `semnum`... ...dépend bien sûr de `cmd`





## Les sémaphores

**Initialisation et contrôle d'un tableau de sémaphores**

`int semctl(int semid, int semnum, int cmd, ... arg);`

- ◇ Retourne 0 si ok, ou -1 si erreur
- ◇ `semid` est l'identifiant du tableau de sémaphores
- ◇ `semnum` est l'indice du sémaphore concerné par la commande `cmd`.

  Si la commande `cmd` concerne le tableau en entier :
  - ⇒ valeur de `semnum` ?
  - ⇒ `semnum = 0` ????
  - ⇒ `semnum = le_nombre_de_sémaphores` ????
- ◇ `cmd` est le type de l'opération à réaliser
- ◇ `arg` est un argument pour l'opération

## Les sémaphores

### Initialisation et contrôle d'un tableau de sémaphores

`int semctl(int semid, int semnum, int cmd, ... arg);`

- Si `cmd=SETVAL` : le sémaphore d'indice `semnum` est initialisé à la valeur `arg` (`int`)
- Si `cmd=SETALL` : l'ensemble du tableau de sémaphores est initialisée avec les valeurs contenues dans le tableau `arg` (`unsigned short int *`)
- Si `cmd=GETALL` : l'ensemble des valeurs des sémaphores du tableau `arg` est recopiée dans le tableau `arg` (`unsigned short int *`)
- Si `cmd= ....` : ....





## Les sémaphores

### Initialisation et contrôle d'un tableau de sémaphores

`int semctl(int semid, int semnum, int cmd, ... arg);`

Et bien sûr ...

- Si `cmd=IPC_RMID` : le tableau de sémaphores est détruit
- Si `cmd=IPC_STAT` : consultation de l'état du tableau de sémaphores (→ `arg` : `struct semid_ds *`)
- Si `cmd=IPC_SET` : modification de l'état du tableau de sémaphores (← `arg` : `struct semid_ds *`)





## Les sémaphores

### Opération sur un sémaphore

`int semop(int semid, struct sembuf *sops,unsigned nsops);`

- ◇ Retourne 0 si ok, ou -1 si erreur, interruption (`EINTR` → relancer)
- ◇ `semid` est l'identifiant du tableau de sémaphores
- ◇ `sops` est un tableau d'opérations à réaliser
  Une case de ce tableau (une opération) est une structure qui possède 3 membres :
  - `sem_num` : l'indice du sémaphore concerné par l'opération
  - `sem_op` : un entier relatif à ajouter à la valeur du sémaphore
  - `sem_flg` : drapeau de l'opération (`0`, `IPC_NOWAIT`, `IPC_UNDO`)
- ◇ `nsops` est le nombre d'opérations à réaliser atomiquement...
  ⟹ C'est-à-dire qu'elles le sont toutes ou qu'aucune ne l'est !





## Exemple de P, V, Z (1)

### Opérations sur un sémaphore

```
                          /* opSem.h */

#ifndef    OPSEM_H
#define    OPSEM_H
                                            /* Processus en attente sur
                                               le semaphore sem_num si : */

void P(int sem_id, int sem_num);       /*    Valeur interne = 0     */

void V(int sem_id, int sem_num);

void Z(int sem_id, int sem_num);       /*    Valeur interne != 0    */

#endif  /* OPSEM_H */
```

## Exemple de P, V, Z (2)

### Opérations sur un sémaphore

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdio.h>
#include <stdlib.h>

#include "opSem.h"                /* opSem.c */

#define Pval -1
#define Vval  1
#define Zval  0

void semaphore (int sem_id, int sem_num, int op)
{
  struct sembuf Ops[1];
  int ok;

  Ops[0].sem_num = sem_num;
  Ops[0].sem_op  = op;
  Ops[0].sem_flg = 0;

  ok = semop(sem_id, Ops, 1);
  if (ok==-1) { perror("semop"); exit(EXIT_FAILURE); }
}
```





## Exemple de P, V, Z (3)

### Opérations sur un sémaphore

```
                              /* opSem.c (suite) */

void P(int sem_id, int sem_num) {
  semaphore (sem_id, sem_num, Pval);
}

void V(int sem_id, int sem_num) {
  semaphore (sem_id, sem_num, Vval);
}

void Z(int sem_id, int sem_num) {
  semaphore (sem_id, sem_num, Zval);
}
```





## Exemple d'utilisation de sémaphores (1)

```
                       /* description.h */

#define SEMNBR 1   /* Nombre de case(s) du tableau de semaphore(s) */

#define MUTEX  0   /* Case 0 du tableau de semaphore(s)            */

#define INIVAL 1   /* Valeur initiale du(des) semaphore(s)         */
```





## Exemple d'utilisation de sémaphores (2)

```
#include <Tous_Les_.h_Necessaires>
#include "description.h"                 /* creSem.c */

int main(void)             /* Un programme de creation de semaphore(s) */
{
 key_t sem_cle;
 int   sem_id;

 sem_cle = ftok("SEMAPHORE",100);
 if (sem_cle==-1) { perror("ftok"); exit(EXIT_FAILURE); }

 sem_id  = semget(sem_cle, SEMNBR, IPC_CREAT | IPC_EXCL | 0666);
 if (sem_id==-1)  { perror("semget"); exit(EXIT_FAILURE); }

 semctl(sem_id, MUTEX, SETVAL, INIVAL); /* Init du semaphore MUTEX */

 return 0;
}
```

## Exemple d'utilisation de sémaphores (3)

```
#include <Tous_Les_.h_Necessaires>
#include "description.h"                  /* desSem.c */

int main(void)             /* Un programme de destruction de semaphore(s) */
{
 key_t sem_cle;
 int   sem_id;

 sem_cle = ftok("SEMAPHORE",100);
 if (sem_cle==-1) { perror("ftok"); exit(EXIT_FAILURE); }

 sem_id  = semget(sem_cle, SEMNBR, IPC_CREAT | 0666);
 if (sem_id==-1)  { perror("semget"); exit(EXIT_FAILURE); }

 semctl(sem_id, SEMNBR, IPC_RMID, NULL);

 return 0;
}
```





## Exemple d'utilisation de sémaphores (4)

```
#include <Tous_Les_.h_Necessaires>
#include "description.h"                  /* procSem.c */
#include "opSem.h"

int main(void)           /* Un programme utilisant un(des) semaphore(s) */
{
 key_t sem_cle;
 int   sem_id;

 sem_cle = ftok("SEMAPHORE",100);
 if (sem_cle==-1) { perror("ftok"); exit(EXIT_FAILURE); }

 sem_id  = semget(sem_cle, SEMNBR, 0);
 if (sem_id==-1)  { perror("semget"); exit(EXIT_FAILURE); }

 P(sem_id,MUTEX);

 system("less livre");  /* Un truc bloquant comme un autre */

 V(sem_id,MUTEX);

 return 0;
}
```

**Programme à lancer plusieurs fois dans des fenêtres différentes...**





## Les segments de mémoire partagée

**Création d'un segment de mémoire partagée, ou récupération d'un segment existant...**

`int shmget(key_t key, int size, int shmflg);`

- ◇ Retourne un identifiant de **segment** de mémoire partagée, ou -1 si erreur
- ◇ `key` est la clé associée au segment de mémoire partagée
- ◇ `size` est la taille du segment de mémoire
- ◇ `shmflg` : drapeau d'ouverture.
  ⟹ OU binaire combinant :
  - le type d'ouverture (`IPC_CREAT` , `IPC_EXCL`) et
  - les droits d'accès au segment (à préciser lors de la création)





## Les segments de mémoire partagée

**Attachement d'un segment de mémoire partagée**

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

- ◇ Retourne l'adresse qui permettra d'atteindre le début du segment, ou `-1` si erreur
- ◇ `shmid` est l'identifiant associée au segment de mémoire partagée
- ◇ `shmaddr` est l'adresse souhaitée pour l'attachement.
  Si `shmaddr=NULL`, le système se débrouille...
- ◇ `shmflg` ... option d'attachement : `0`, `SHM_RDONLY`

## Les segments de mémoire partagée

### Détachement d'un segment de mémoire partagée

`int shmdt(const void *shmaddr);`

◇ Retourne 0 si ok, -1 si erreur

◇ `shmaddr` est l'adresse du segment à détacher.





## Les segments de mémoire partagée

### Contrôle d'un segment de mémoire partagée

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

◇ Retourne 0 si ok, ou -1 si erreur

◇ `shmid` est l'identifiant du segment de mémoire partagée

◇ `cmd` est la commande à réaliser :

- `IPC_RMID` ⇒ Suppression du segment de mémoire
- `IPC_STAT` ⇒ Consultation de l'état du segment (→ `buf`)
- `IPC_SET` ⇒ Modification de l'état du segment (← `buf`)

◇ `buf` est un pointeur sur une structure de type `struct shmid_d` permettant de consulter ou de modifier l'état du segment de mémoire partagée...





## Exemple d'utilisation de segments de mem partagée (1)

```
                       /* opSem.h */
#ifndef    OPSEM_H
#define    OPSEM_H
                                       /* Processus en attente sur
                                          le semaphore sem_num si : */

void P(int sem_id, int sem_num);       /*    Valeur interne = 0     */

void V(int sem_id, int sem_num);

void Z(int sem_id, int sem_num);       /*    Valeur interne != 0    */

#endif  /* OPSEM_H */


                       /* opSem.c */

          .....................................
```





## Exemple d'utilisation de segments de mem partagée (2)

```
                      /* description.h */

#define SEMNBR 1   /* Nombre de case(s) du tableau de semaphore(s) */

#define MUTEX  0   /* Case 0 du tableau de semaphore(s)            */

#define INIVAL 1   /* Valeur initiale du(des) semaphore(s)         */

typedef struct {
                pid_t lastPid;
                int   nbAccess;
               }                  data;
```

## Exemple d'utilisation de segments de mem partagée (3)

```
#include <Tous_Les_.h_Necessaires>
#include "description.h"                 /* creIPC.c */

int main(void)            /* Un programme de creation de semaphore(s) */
{                         /* et de memoire partagee                   */
 key_t sem_cle, shm_cle;
 int   sem_id, shm_id;

 unsigned short semValue[SEMNBR]; int i;
 data *shmAdr;

 sem_cle = ftok("SEMAPHORE",100);                                if ...
 sem_id  = semget(sem_cle, SEMNBR, IPC_CREAT | IPC_EXCL | 0666); if ...

 for(i=0;i<SEMNBR;i++) { semValue[i]=INIVAL; }

 semctl(sem_id,SEMNBR,SETALL,semValue);  /* Init de "tous" les semaphores */
```





## Exemple d'utilisation de segments de mem partagée (4)

```
                                    /* creIPC.c (suite) */

 shm_cle = ftok("MEMOIRE",100);
 if (shm_cle==-1) { perror("ftok"); exit(EXIT_FAILURE); }

 shm_id  = shmget(shm_cle,sizeof(data),IPC_CREAT | IPC_EXCL | 0666);
 if (shm_id==-1)  { perror("shmget"); exit(EXIT_FAILURE); }

 shmAdr = (data*)shmat(shm_id,NULL,0);
 if (shmAdr==(data*)-1) { perror("shmat"); exit(EXIT_FAILURE); }

 shmAdr->lastPid  = 0; /* Mise a 0 du pid_t en memoire partagee */
 shmAdr->nbAccess = 0; /* Mise a 0 de l'int en memoire partagee */

 shmdt(shmAdr);

 return 0;
}
```





## Exemple d'utilisation de segments de mem partagée (5)

```
#include <Tous_Les_.h_Necessaires>
#include "description.h"                 /* desIPC.c */

int main(void)            /* Un programme de destruction de semaphore(s) */
{                         /* et de memoire partagee                      */
 key_t sem_cle, shm_cle;
 int   sem_id, shm_id;

 sem_cle = ftok("SEMAPHORE",100);                            if ...
 sem_id  = semget(sem_cle, SEMNBR, IPC_CREAT | 0666);        if ...
 semctl(sem_id, SEMNBR, IPC_RMID, NULL);

 shm_cle = ftok("MEMOIRE",100);                              if ...
 shm_id  = shmget(shm_cle, sizeof(data), IPC_CREAT | 0666);  if ...
 shmctl(shm_id, IPC_RMID, NULL);

 return 0;
}
```





## Exemple d'utilisation de segments de mem partagée (6)

```
#include <Tous_Les_.h_Necessaires>
#include "description.h"                 /* procIPC.c */
#include "opSem.h"
int main(void)          /* Un programme utilisant un(des) semaphore(s) */
{                       /* et de la memoire partagee                   */
 key_t sem_cle, shm_cle;
 int   sem_id, shm_id;   data *shmAdr;

 sem_cle = ftok("SEMAPHORE",100);            if ...
 sem_id  = semget(sem_cle, SEMNBR, 0);       if ...
 shm_cle = ftok("MEMOIRE",100);              if ...
 shm_id  = shmget(shm_cle, sizeof(data), 0); if ...
 shmAdr  = (data*)shmat(shm_id, NULL, 0);    if ...

 P(sem_id,MUTEX);
  system("less livre");
  shmAdr->lastPid=getpid(); shmAdr->nbAccess++; /* Acces memoire partagee */
 V(sem_id,MUTEX);

 shmdt(shmAdr);
 return 0;
}
```

**Programme à lancer plusieurs fois "simultanément"...**

# Les files de messages
# Principes généraux (hors IPC System V)

- ▷ Les communications de messages se font à travers deux opérations fondammentales
  - ◇ send(destinataire, message)
  - ◇ receive(émetteur, message)
- ▷ Les messages sont de tailles variables ou fixes
- ▷ Les opérations d'envoi et de réception peuvent être:
  - ◇ soit **directe** entre les processus
  - ◇ soit **indirecte** par l'intermédiare d'une "boîte aux lettres"
- ▷ Les communications se font de manière **synchrone** ou **asynchrone**.





# Les files de messages
# Principes généraux (hors IPC System V)

## Le producteur-consommateur avec échange de messages (1)

```
#include "prototypes.h"
#define N 100           /* Nb emplacements   */
#define TAILLE 4        /* taille du message */

typedef int message[TAILLE];

void producteur(void)
{
 int objet;      /* tampon de message */
 message m;
 while (1) {
  produire_objet(&objet);     /* produire l'objet suivant         */
  receive(consommateur,&m);   /* attendre un message vide         */
  faire_message(&m,objet);    /* construire le message a envoyer  */
  send(consommateur,&m);      /* envoi de message au consommateur */
 }
}
```





# Les files de messages
# Principes généraux (hors IPC System V)

## Le producteur-consommateur avec échange de messages (2)

```
void consommateur(void)
{
 int objet,i;
 message m;

 for(i=0;i<N;i++) send(producteur,&m); /* N messages vides envoyes */

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





# Les files de messages
# Principes généraux (hors IPC System V)

## Le producteur-consommateur avec échange de messages (3)

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

## Les files de messages : IPC System V

**Création d'une file de message, ou récupération d'une file existante...**

```
int msgget(key_t key, int msgflg);
```

◇ Retourne un identifiant de file, ou -1 si erreur

◇ `key` est la clé associée à la file

◇ `msgflg` : drapeau d'ouverture.
⟹ OU binaire combinant :
– le type d'ouverture (`IPC_CREAT` , `IPC_EXCL`) et
– les droits d'accès à la file (à préciser lors de la création)





## Les files de messages : IPC System V

**Structure générique des messages**

```
struct msgbuf  {
                 long mtype;    /* message type, must be > 0 */
                 char mtext[1]; /* message data */
               };
```

**Dans la pratique :**

```
struct msgutile {
                  long mtype;
                  .....            /* Des donnees utiles ! */
                };
```





## Les files de messages : IPC System V

**Envoi d'un message**

```
int msgsnd(int msqid,
             struct msgbuf *msgp, int msgsz, int msgflg);
```

◇ Retourne 0 si ok, ou -1 si erreur

◇ `msqid` est l'identifiant de la file

◇ `msgp` est un pointeur sur la structure du message à envoyer

◇ `msgsz` est la taille du message (sans compter le `long` du début! )

◇ `msgflg` :
0 ⇒ bloquant si la file est pleine,
`IPC_NOWAIT` ⇒ non bloquant si la file est pleine
Dans ce cas : file pleine ⟹ retour -1 et `errno=EAGAIN`





## Les files de messages : IPC System V

**Réception d'un message**

```
int msgrcv(int msqid, struct msgbuf *msgp, int msgsz,
             long msgtyp, int msgflg);
```

◇ Retourne la longueur du message lu (sans le `long`) si ok, ou -1

◇ `msqid` est l'identifiant de la file

◇ `msgp` est un pointeur sur une structure de message (buffer réception)

◇ `msgsz` est la taille (sans le long) du message (taille buffer réception)

◇ `msgtyp` est le type de message à extraire de la file

◇ `msgflg` :
0 ⇒ bloquant si pas de message du bon type,
`IPC_NOWAIT` ⇒ non bloquant si pas de message du bon type
Dans ce cas : pas de message ⟹ retour -1 et `errno=ENOMSG`

## Les files de messages : IPC System V

### Réception d'un message

```
int msgrcv(int msqid,
           struct msgbuf *msgp, int msgsz,
           long msgtyp, int msgflg);
```

`msgtyp` est donc le type de message à extraire de la file :

- ◇ `msgtyp` $\geq 1$ : le premier message ayant le type `msgtyp` est extrait,
- ◇ `msgtyp` $= 0$ : le premier message est extrait,
- ◇ `msgtyp` $\leq -1$ : le premier message de type le plus petit inférieur ou égal à | `msgtyp` | est extrait.





## Les files de messages : IPC System V

### Contrôle d'une file de messages

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

- ◇ Retourne 0 si ok, ou -1 si erreur
- ◇ `msqid` est l'identifiant de la file
- ◇ `cmd` est la commande à réaliser :
  - `IPC_RMID` ⇒ Suppression de la file de message
  - `IPC_STAT` ⇒ Consultation de l'état de la file de message (→ `buf`)
  - `IPC_SET` ⇒ Modification de l'état de la file de message (← `buf`)
- ◇ `buf` est un pointeur sur une structure de type `struct msqid_d` permettant de consulter ou de modifier l'état de la file...





## Exemple d'utilisation de file de messages (1)

Des fonctions de manipulation/saisie de lignes de caractères :

```
                         /* manipLine.h */

#ifndef    MANIPLINE_H
#define    MANIPLINE_H

extern char *scanLine(char *str, size_t size);
extern void  removeNewLine(char *str);

#endif  /* MANIPLINE_H */
```





## Exemple d'utilisation de file de messages (2)

```
#include <string.h>             /* scanLine.c */
#include <stdio.h>

#include "scanLine.h"

char *scanLine(char *str, size_t size)
{
 char *returnValue = fgets(str,size,stdin);

 if (returnValue!=NULL) removeNewLine(str);

 return returnValue;
}

void  removeNewLine(char *str)
{
 char *posNL = strrchr(str,'\n');

 if (posNL!=NULL) *posNL='\0';
}
```

## Exemple d'utilisation de file de messages (3)

```
#include <sys/types.h>
#include <sys/ipc.h>                    /* createur.c */
#include <sys/msg.h>
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
 key_t   msg_cle;
 int     msg_id;

 msg_cle = ftok("MESSAGE",1);
 if (msg_cle == -1) { perror("ftok"); exit(EXIT_FAILURE);   }

 msg_id  = msgget(msg_cle, IPC_CREAT | IPC_EXCL | 0666);
 if (msg_id == -1)  { perror("msgget"); exit(EXIT_FAILURE); }

 return EXIT_SUCCESS;
}
```





## Exemple d'utilisation de file de messages (4)

```
#include <sys/types.h>
#include <sys/ipc.h>               /* destructeur.c */
#include <sys/msg.h>
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
 key_t   msg_cle;
 int     msg_id;  int ok;

 msg_cle = ftok("MESSAGE",1);
 if (msg_cle == -1) { perror("ftok"); exit(EXIT_FAILURE);   }

 msg_id  = msgget(msg_cle, IPC_CREAT | 0666);
 if (msg_id == -1)  { perror("msgget"); exit(EXIT_FAILURE); }

 ok = msgctl(msg_id,IPC_RMID,NULL);
 if (ok == -1)      { perror("msgctl"); exit(EXIT_FAILURE); }

 return EXIT_SUCCESS;
}
```





## Exemple d'utilisation de file de messages (5)

Un peu plus d'info sur l'exemple... le type de messages échangés.

```
#define TAILLE  256       /*  message.h  */

#define DATA   (long)1

typedef struct {
  char texte [TAILLE];
} data;

typedef struct {
                long  mtype;
                data  mdata;
                            }   msgtext ;
```





## Exemple d'utilisation de file de messages (6)

```
#include <sys/types.h>
#include <sys/ipc.h>             /* clavier.c */
#include <sys/msg.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

#include "scanLine.h"
#include "message.h"

int main(void)
{
 key_t   msg_cle;
 int     msg_id;
 msgtext message;

 msg_cle = ftok("MESSAGE",1);
 if (msg_cle == -1) { perror("ftok"); exit(EXIT_FAILURE);   }

 msg_id  = msgget(msg_cle, 0);
 if (msg_id == -1)  { perror("msgget"); exit(EXIT_FAILURE); }
```

## Exemple d'utilisation de file de messages (7)

```
                                /* clavier.c (suite) */

 printf("Entrez des lignes et terminez par fin : \n");

 message.mtype = DATA;

 do {
     scanLine(message.mdata.texte,TAILLE);
     if (msgsnd(msg_id,&message, sizeof(data),0)==-1) { perror("msgsnd");
                                                         exit(EXIT_FAILURE);
                                                       }
    }
 while (strcmp(message.mdata.texte,"fin")!=0);

 return EXIT_SUCCESS;
}
```





## Exemple d'utilisation de file de messages (8)

```
#include <sys/types.h>
#include <sys/ipc.h>                /*  ecran.c */
#include <sys/msg.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

#include "message.h"

int main(void)
{
 key_t   msg_cle;
 int     msg_id;
 msgtext message;

 msg_cle = ftok("MESSAGE",1);
 if (msg_cle == -1) { perror("ftok"); exit(EXIT_FAILURE);   }

 msg_id  = msgget(msg_cle, 0);
 if (msg_id == -1)  { perror("msgget"); exit(EXIT_FAILURE); }
```





## Exemple d'utilisation de file de messages (9)

```
                        /*  ecran.c (suite) */

 do {
     if (msgrcv(msg_id,&message,sizeof(data),DATA,0)==-1)
         { perror("msgrcv");
           exit(EXIT_FAILURE) ;
         }
     printf("Recu %s\n",message.mdata.texte);
    }
 while (strcmp(message.mdata.texte,"fin")!=0);

 return EXIT_SUCCESS;
}
```





## Les tubes de communication (pipe)

### Les tubes de communication (pipe)

*Introduction*
*Les tubes sans nom*
*Les tubes nommés*
*Les tubes et le recouvrement de processus (*`exec`*)*

## Introduction

▷ Un tube (ou pipe) est un moyen de communication qui permet à des processus d'échanger une suite d'octets.

▷ Un tube est géré comme une mémoire tampon de type FIFO. La taille d'un tube est égale à `PIPE_BUF`

(voir `limits.h` ⇒ en général 4Ko ou 8Ko)

▷ Un tube est temporaire

▷ Les processus peuvent utiliser un tube via les fonctions classiques de lecture/écriture dans un fichier (`read`/`write` : <`unistd.h`>). Mais un tube n'est pas un fichier !





## Introduction

*Lecteur* : processus possédant un descripteur en lecture sur le tube
*Ecrivain* : processus possédant un descripteur en écriture sur le tube

```
ssize_t read(int fd, void *buf, size_t len);
```

◇ Retourne le nombre d'octets lus ou `-1` si erreur
◇ `fd` : descripteur de fichier,
◇ `buf` : buffer de lecture,
◇ `len` : taille du buffer de lecture.

```
ssize_t write(int fd, const void *buf, size_t count);
```

◇ Retourne le nombre d'octets écrits ou `-1` si erreur
◇ `fd` : descripteur de fichier,
◇ `buf` : message à écrire (buffer d'écriture),
◇ `count` : taille du message.

**Remarque** : un descripteur se ferme avec `int close(int fd);`





## Introduction

**Synchronisation des lectures/écritures dans un tube :**

▷ `read` est bloquant si :

◇ le tube est vide et

◇ il existe encore au moins un *écrivain* dans le tube

▷ `write` est bloquant si le tube est plein (`PIPE_BUF`)

▷ La *fin du tube* est atteinte si :

◇ le tube est vide et

◇ il n'existe plus d'*écrivain* dans le tube

Dans ce cas, `read` retourne `0` pour indiquer la *fin du tube*.





## Introduction

**Il existe deux types de tube :**

▷ les tubes sans nom

▷ les tubes nommés

**Différences :**

▷ Un tube sans nom est accessible uniquement par le processus créateur de ce tube et ses héritiers ...

▷ Un tube nommé possède un *nom externe* apparaissant dans l'arborescence de fichier (`ls -l` ⇒ `p`).

⟹ communication entre processus indépendants

## Les tubes sans nom

**Création :**

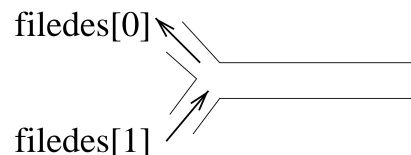
`int pipe(int filedes[2]);` <unistd.h>

- ◇ Retourne `0` si ok ou `-1` si erreur
- ◇ `filedes` : tableau de 2 descripteurs de fichier.

  Après l'appel à `pipe`, `filedes` contient les descripteurs de fichier associés au tube :

  `filedes[0]` : descripteur ouvert en lecture sur le tube,

  `filedes[1]` : descripteur ouvert en écriture sur le tube.








## Les tubes sans nom, exemple (1)

```
                    /* pipe.c */

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

void fils(int tube[2]);

void pere(int tube[2]);
```





## Les tubes sans nom, exemple (2)

```
int main(void)                       /* pipe.c (suite) */
{
 int   tube[2];
 pid_t pid = 0;

 if (pipe(tube) != 0) { perror("pipe"); exit(EXIT_FAILURE); }

 pid = fork();

 switch (pid)
 {
  case -1 : perror("fork"); exit(EXIT_FAILURE);
            break;
  case  0 : fils(tube);
            break;
  default : pere(tube);
            break;
 }

 return EXIT_SUCCESS;
}
```





## Les tubes sans nom, exemple (3)

```
void pere(int tube[2])              /* pipe.c (suite) */
{
 int ok = 0 ;
 int ch;

 close(tube[0]);    /* Le pere ne lit pas dans le tube */

 printf("Entrez des caracteres et terminez par CTRL+D (^D): \n");

 while ((ch=getchar())!=EOF)
 {
  ok = write(tube[1],&ch,1);
  if (ok==-1) { perror("write"); break; }
 }

 close(tube[1]);

 wait(NULL);         /* Pour attendre "explicitement" le processus fils */
}
```

## Les tubes sans nom, exemple (4)

```
void fils(int tube[2])             /* pipe.c (suite et fin ) */
{
 int   ok = 0;
 char  ch;
 int   compteur=0;

 close(tube[1]); /* Le fils n'ecrit pas dans le tube               */
                 /* FONDAMENTAL pour atteindre la "fin du tube" ! */

 ok=read(tube[0],&ch,1);
 while ((ok!=-1) && (ok!=0)) {
                              compteur++;
                              ok=read(tube[0],&ch,1);
 }

 if (ok==-1) { perror("read"); exit(EXIT_FAILURE); }

 printf("Recu %d caracteres\n",compteur);

 close(tube[0]);
}
```





## Les tubes nommés

**Création :**

`int mkfifo(const char *pathname, mode_t mode);`

- ◇ Retourne `0` si ok ou `-1` si erreur
- ◇ `pathname` : nom externe du tube
  - ⟹ création de la référence `pathname`
- ◇ `mode` : droits d'accès au tube nommé

pathname

**Ouverture :**

`int open(const char *pathname, int flags);`

- ◇ Retourne un descripteur ouvert sur le tube ou `-1` si erreur
- ◇ `pathname` : nom externe du tube à ouvrir
- ◇ `flags` : O_RDONLY ⇒ lecture ; O_WRONLY ⇒ écriture





## Les tubes nommés, exemple (1)

**Création du tube dans un programme "externe" :**

```
                             /* creePipe.c */
#include <stdlib.h>
#include <sys/stat.h>
#include <stdio.h>

int main(void)
{
 if (mkfifo("Dialogue", 0666) != 0) { perror("mkfifo");
                                      exit(EXIT_FAILURE);
 }

 return EXIT_SUCCESS;
}
```





## Les tubes nommés, exemple (2)

```
                               /* clavier.c */
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <stdio.h>

int main(void)
{
 int   ok = 0 , tube1 = 0;

 char  ch;

 if ((tube1 = open("Dialogue", O_WRONLY)) == -1) { perror("open");
                                                   exit(EXIT_FAILURE);
 }
```

## Les tubes nommés, exemple (3)

```
                                /* clavier.c (suite) */

 printf("Entrez des caracteres et terminez par CTRL+D (^D): \n");

 while ((ch=getchar())!=EOF)
 {
  ok = write(tube1,&ch,1);
  if (ok==-1) { perror("write"); break; }
 }

 close(tube1);

 return EXIT_SUCCESS;
}
```





## Les tubes nommés, exemple (4)

```
                                /* ecran.c */
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <stdio.h>

int main(void)
{
 int   ok = 0, tube0 = 0;
 char  ch;
 int   compteur=0;

 if ((tube0 = open("Dialogue", O_RDONLY)) ==-1) { perror("open");
                                                  exit(EXIT_FAILURE);
 }
```





## Les tubes nommés, exemple (5)

```
                                /* ecran.c (suite) */
 ok=read(tube0,&ch,1);
 while ((ok!=-1) && (ok!=0)) {
                              compteur++;
                              ok=read(tube0,&ch,1);
 }

 if (ok==-1) { perror("read");
              exit(EXIT_FAILURE);
 }

 printf("Recu %d caracteres\n",compteur);

 close(tube0);

 return EXIT_SUCCESS;
}
```





## Les tubes et le recouvrement de processus (`exec`)

**Introduction**

`exec`

`#include <unistd.h>`

`exec` est un nom générique désignant plusieurs appels système permettant à un processus d'exécuter un nouveau programme **sans création de nouveau processus**.

⟹
- ◇ Les descripteurs de fichiers ouverts restent ouverts.
- ◇ Le masque des signaux bloqués est conservé.
- ◇ Les signaux ignorés restent ignorés, les autres reprennent leurs comportements par défaut

## Les tubes et le recouvrement de processus (exec)

### Fonctions de la famille exec

```
int execl (const char *path,const char *arg, ...);

int execlp(const char *file,const char *arg, ...);

int execle(const char *path,const char *arg , ...,char * const envp[]);

int execv (const char *path,char *const argv[]);

int execvp(const char *file,char *const argv[]);

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





## Les tubes et le recouvrement de processus (exec)

### Fonctions de la famille exec

| Nom | Nb. paramètres | PATH | Environnement |
|---|---|---|---|
| execl | variable | non | ancien |
| execlp | variable | oui | ancien |
| execle | variable | non | nouveau |
| execv | fixe | non | ancien |
| execvp | fixe | oui | ancien |
| execve | fixe | non | nouveau |





## Les tubes et le recouvrement de processus (exec)

### Exemple d'utilisation (1)

```
/* pipeExec.c */

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

void fils(int tube[2]);

void pere(int tube[2]);
```





## Tube + recouvrement, Exemple d'utilisation (2)

```
int main(void)                    /* pipeExec.c (suite) */
{
 int   tube[2];
 pid_t pid = 0;

 if (pipe(tube) != 0) { perror("pipe"); exit(EXIT_FAILURE); }

 pid = fork();

 switch (pid)
 {
  case -1 : perror("fork"); exit(EXIT_FAILURE);
            break;
  case  0 : fils(tube);
            break;
  default : pere(tube);
            break;
 }

 return EXIT_SUCCESS;
}
```

## Tube + recouvrement, Exemple d'utilisation (3)

```
void pere(int tube[2])              /* pipeExec.c (suite) */
{
 int ok = 0 ;
 int ch;

 close(tube[0]);    /* Le pere ne lit pas dans le tube */

 printf("Entrez des caracteres et terminez par CTRL+D (^D): \n");

 while ((ch=getchar())!=EOF)
 {
  ok = write(tube[1],&ch,1);
  if (ok==-1) { perror("write"); break; }
 }

 close(tube[1]);

 wait(NULL);         /* Pour attendre "explicitement" le processus fils */
}
```





## Tube + recouvrement, Exemple d'utilisation (3)

```
                             /* pipeExec.c (suite et fin) */

void fils(int tube[2])
{
 dup2(tube[0],STDIN_FILENO);     /* tube[0] => Entree standard         */

 close(tube[0]);  /* Il est donc maintenant possible de fermer tube[0] */

 close(tube[1]);  /* ESSENTIEL : pour la fin du tube                   */

 execlp("wc","wc","-c",NULL);
 perror("execlp");
}
```





## Les signaux

### Les Signaux

*Introduction*
*Envoi d'un signal*
*Réception d'un signal – Handler d'interruption*
*Réception d'un signal... signal() : Non POSIX*
*Réception d'un signal... sigaction() : POSIX*
*Attente explicite d'un signal: pause()*





## Introduction

**Définition :**

Un signal est une interruption logicielle qui permet d'informer les processus destinataires qu'un événement particulier s'est produit

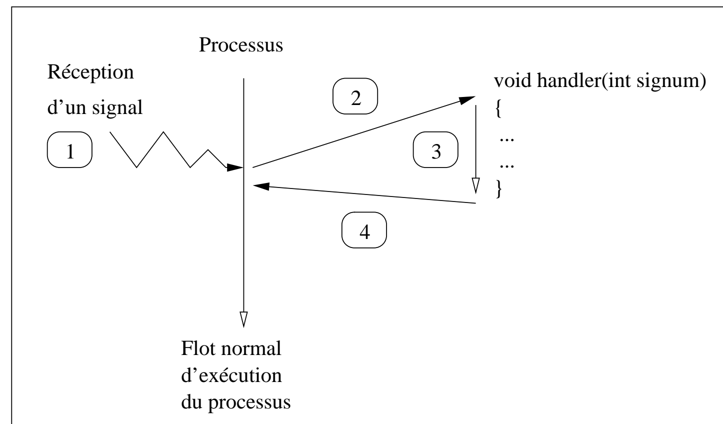
**Remarques :**

▷ En général, la réception d'un signal par un processus provoque la fin de ce processus.

▷ Il est possible de spécifier un handler d'interruption décrivant le comportement du processus lors de la réception d'un signal particulier.

  ◇ En général, à la fin du handler, le processus reprend le cours normal de son exécution.

## Introduction








## Voici quelques signaux

| **Signaux** | **Emission lorsque ...** |
|---|---|
| SIGUP | fermeture du terminal de contrôle du processus |
| SIGINT | interruption clavier (en général ˆC) |
| SIGQUIT | interruption clavier (en général ˆ\) |
| SIGABRT | lorsque la fonction C `abort()` est exécutée |
| SIGILL | détection d'instruction assembleur illégale |
| SIGPIPE | détection d'erreur d'écriture sur un tube ou sur une socket |
| SIGKILL | pour provoquer un arrêt du processus... l'arme absolue ! |
| SIGBUS | erreur sur le bus |
| SIGALRM | alarme d'horloge programmée via la fonction `alarm()` |
| SIGSEGV | erreur de segmentation |
| SIGUSR1 | signal utilisateur numéro 1 |
| SIGUSR2 | signal utilisateur numéro 2 |

**Il en existe d'autres, ... et des signaux temps réel, ...**





## Introduction

**Remarques complémentaires :**

▷ L'**émission d'un signal** par programme se fait via la fonction `kill()` ... et

▷ il existe 2 façons de gérer la **réception des signaux** :

- ◇ à la mode ANSI C (une norme ?) $\Longrightarrow$ `signal`
- ◇ à la mode POSIX (une norme !) $\Longrightarrow$ `sigaction`,...

**Fiabilité et comportements identiques sur plusieurs systèmes $\Longrightarrow$ POSIX**





## Envoi d'un signal

```
int kill(pid_t pid, int signum);                    <signal.h>
```

Envoie le signal `signum` à un ou plusieurs processus

- Retourne 0 si ok, -1 si ko
- `pid` représente le ou les processus destinataires:
  - ◇ si `pid` $\geq 1$ : le processus `pid`.
  - ◇ si `pid` $= 0$ : tous les processus (sauf 0) de même groupe que le processus émetteur.
  - ◇ si `pid` = -1 : indéfini sous POSIX.
  - ◇ si `pid` $\leq$ -2 : tous les processus appartenant au groupe identifié par |`pid`|.

```
void raise(int signum); ⟺ kill(getpid(), signum);
```

## Réception d'un signal – Handler d'interruption

Il est possible de spécifier un handler d'interruption décrivant le comportement du processus lors de la réception d'un signal particulier.

En général, ce handler est de la forme :

```
void handler(int signum)
{ ...
 switch(signum) {

   no1 : ...
   no2 : ...

 }
  ...
}
```





## Réception d'un signal ... signal()

**... Non POSIX ... et donc comportement pas très sûr...**

```
void (*signal(int signum, void (*handler)(int)))(int);
```

On peut réécrire ce prototype:

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

La fonction `signal` permet de décrire, via la fonction `handler`, le comportement d'un processus lors de la réception du signal `signum`.

- le numéro du signal est passé à la fonction lors de son activation
- fonctions prédéfinies : ◇ SIG_IGN : ignore le signal
  ◇ SIG_DFL : comportement par défaut
- `signal` retourne l'ancien handler ou SIG_ERR si il y a un problème...





## Réception d'un signal ... signal(), exemple no 1

```
#include <signal.h>        /* ignore.c */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main(void)
{
 int i = 0;

 if (signal(SIGINT,SIG_IGN) == SIG_ERR)
     fprintf(stderr,"Signal SIGINT non ignore\n");

 for(i=1;i<=3;i++) { fprintf(stdout,"hello %d\n",i); sleep(2); }

 return EXIT_SUCCESS;
}
```

```
$ ./ignore
hello 1
^C
hello 2
^C
hello 3
$
```





## Réception d'un signal ... signal(), exemple no 2

```
#include <signal.h>
#include <stdio.h>         /* intercepte.c */
#include <unistd.h>
#include <stdlib.h>

void interception(int numero_signal) {
 fprintf(stdout,
         "Recu signal %d\n", numero_signal);
}

int main(void)
{
 int i = 0;

 if (signal(SIGINT,interception) == SIG_ERR)
     fprintf(stderr,"Signal SIGINT non capte\n");

 for(i=1;i<=3;i++) { fprintf(stdout,"hello %d\n",i); sleep(2); }

 return EXIT_SUCCESS;
}
```

```
$ ./intercepte
hello 1
^CRecu signal 2
hello 2
^CRecu signal 2
hello 3
^CRecu signal 2
$
```

## Réception d'un signal ... sigaction()

**... POSIX ...**

```
int sigaction(int signum, const struct sigaction *act,
                           struct sigaction *oldact);
```

`sigaction()` permet de décrire précisément le comportement d'un processus lors de la réception du signal `signum`.

- ◇ `act` est un pointeur sur une structure décrivant un nouveau comportement. Si `act==NULL`, permet de sauvegarder le comportement courant dans `oldact`.
- ◇ `oldact` est un pointeur sur une structure permettant de sauvegarder l'ancien comportement. Si `oldact==NULL`, pas de sauvegarde.
- ◇ Retourne 0 si ok, -1 si problème.





## Réception d'un signal ... sigaction()

Via la structure pointée par `act` il est possible :

- ▷ d'indiquer le handler à appeler lors de la réception de `signum`. Eventuellement `SIG_DFL` ou `SIG_IGN`.
- ▷ de préciser la liste des signaux bloqués lors de l'exécution du handler. Par défaut, `signum` est bloqué lors de l'exécution du handler.
- ▷ décrire finement le comportement d'un processus lors de la réception du signal.

  Citons, par exemples :

  - ◇ le fait qu'un signal ne soit pas bloqué à l'intérieur de son propre handler, ou bien que,
  - ◇ le comportement par défaut soit réinstallé lors de l'activation du handler.





## Réception d'un signal ... sigaction()

```
struct sigaction {

    sighandler_t     sa_handler;     /* - Le handler ou SIG_DFL, SIG_IGN */

    sigset_t         sa_mask;        /* - L'ensemble des signaux bloques */
                                     /*   lors de l'execution du handler */

    int              sa_flags;       /* - Ou binaire entre differentes   */
                                     /*   constantes decrivant finement  */
    .....                            /*   le comportement du handler     */
};
```

Voici quelques constantes binaires utilisées pour `sa_flags` :

| | |
|---|---|
| SA_RESTART | Les appels systèmes lents interrompus par le signal sont automatiquement relancés. Evite de tester `errno==EINTR` . |
| SA_INTERRUPT | Comportement inverse à SA_RESTART |
| SA_NODEFER | Signal non bloqué à l'intérieur de son propre handler |
| SA_RESETHAND | Comportement par défaut réinstallé lors de l'activation du handler |
| SA_ONESHOT | Identique à SA_RESETHAND |





## Réception d'un signal ... sigaction(), exemple no 1

```
#include <signal.h>
#include <stdio.h>       /* ignorePOSIX.c */
#include <unistd.h>
#include <stdlib.h>

int main(void)
{
 int i = 0;
 struct sigaction action;

 action.sa_handler = SIG_IGN;
 sigemptyset( &(action.sa_mask) );
 action.sa_flags = 0;

 if (sigaction(SIGINT,&action, NULL) != 0)       /* SIGINT: ^C */
     fprintf(stderr,"Signal SIGINT non ignore\n");

 for(i=1;i<=3;i++) { fprintf(stdout,"hello %d\n",i); sleep(2); }

 return EXIT_SUCCESS;
}
```

```
$ ./ignorePOSIX
hello 1
^C
hello 2
^C
hello 3
^C
$
```

## Réception d'un signal...sigaction(), exemple no 2 (1)

```
#include <signal.h>
#include <stdio.h>        /* interceptePOSIX.c */
#include <unistd.h>
#include <stdlib.h>

void interception(int numero_signal)
{
 fprintf(stdout,
       "Recu signal %d\n", numero_signal);
}

int main(void)
{
 int i = 0;
 struct sigaction action;
```

```
$ ./interceptePOSIX
hello 1
^CRecu signal 2
hello 2
^CRecu signal 2
hello 3
^CRecu signal 2
$
```





## Réception d'un signal...sigaction(), exemple no 2 (2)

```
                                  /* interceptePOSIX.c (suite) */

 action.sa_handler = interception;
 sigemptyset( &(action.sa_mask) );
 action.sa_flags = 0;

 if (sigaction(SIGINT,&action, NULL) != 0)      /* SIGINT: ^C */
     fprintf(stderr,"Signal SIGINT non capte\n");

 for(i=1;i<=3;i++) { fprintf(stdout,"hello %d\n",i); sleep(2); }

 return EXIT_SUCCESS;
}
```





## Attente explicite d'un signal: pause()

```
int pause(void);                    #include<unistd.h>
```

La fonction `pause` force le processus appelant à sendormir jusqu'à ce qu'un signal soit reçu.

A la réception d'un signal :

- ▷ le processus se termine (par défaut) ou
- ▷ va exécuter un handler d'interruption.
  $\Longrightarrow$ dans ce cas, à la fin du handler, le processus reprend son exécution à l'instruction se trouvant juste après `pause();`





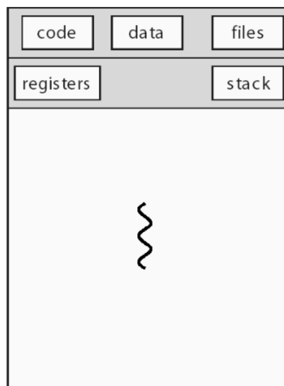
## Activités et modes d'ordonnancement

### Activités et modes d'ordonnancement

*Des processus aux activités*
*Description des activités*
*Gestion des activités*
*Politiques d'ordonnancement*
*Mises en œuvre Windows NT4 et UNIX SVR4*

---

## Des processus aux activités : processus




Contexte d'un processus (mono-activité)





## Des processus aux activités : processus

Problèmes :

▷ la définition d'un contexte de processus est complexe,

▷ les communications entre 2 processus font souvent intervenir le noyau.

Conséquences :

▷ le temps de commutation de contexte est significatif
→ le rendement du système se détériore en cas de processus pléthoriques

▷ le coût des communications inter-processus tend à limiter l'usage des processus aux masquages des entrées/sorties bloquantes

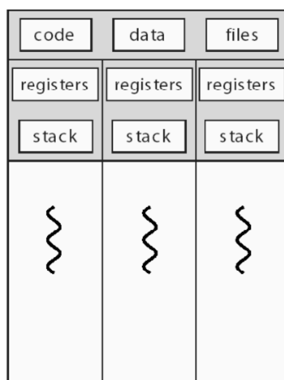
⇒ structuration du code : des opérations naturellement concurrentes doivent être séquentialisées

⇒ pas d'expression du parallélisme interne aux processus pour une machine multi-processeurs





## Des processus aux activités : activités




Contexte d'un processus multi-threads (multi-activités) :
contexte du processus + contextes des activités





## Des processus aux activités : activités

À l'intérieur d'un contexte, on peut distinguer :

▷ l'allocation de ressources mémoires, et d'entrées/sorties
→ elle se fait sur la base du processus,

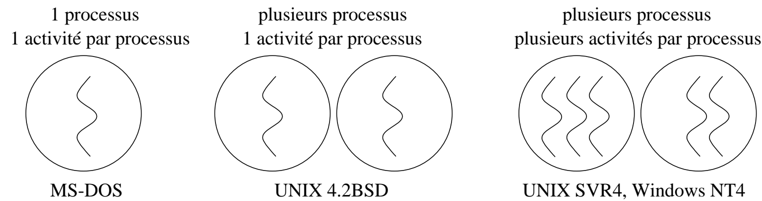
▷ l'allocation de ressources de calcul
→ elle se fait sur la base de l'activité (en anglais *thread*).

Une activité est liée à un processus, dont elle utilise les ressources.

Un processus contient une ou plusieurs activités.

Le processus est vu comme un environnement de travail passif, les calculs étant réalisés par les activités à l'intérieur de cet environnement.

## Des processus aux activités : processus et activités




Systèmes multi-activités (*multithreaded systems*) = un système d'exploitation en mesure de gérer plusieurs activités.





## Des processus aux activités : avantages des activités

- ▷ Réactivité
- ▷ Partage de ressources
- ▷ Economie
- ▷ Utilisation d'architectures multi-processeurs





## Des processus aux activités : inconvénients des activités

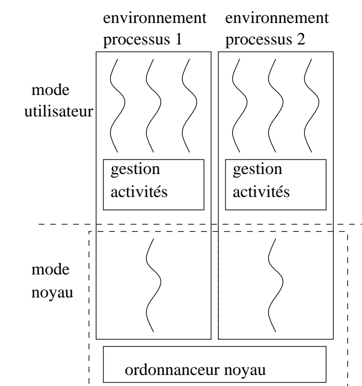
- ▷ Parfois complexes à mettre en œuvre
- ▷ En fonction des systèmes, comportements différents
- ▷ Comprendre les notions de **thread utilisateur** et **thread noyau**....





## Activités en mode utilisateur

- ▷ Les activités utilisateurs sont gérées indépendamment du noyau.
- ▷ Les commutations entre les activités ne nécessitent pas de passage en mode noyau ;
  elles sont gérées par les fonctions de bibliothèques.
- ▷ Par contre,
  si une activité fait appel à un service bloquant du noyau, le processus, et donc toutes ses activités, est bloqué.
- ▷ Pthreads, Threads Java, Threads Win32



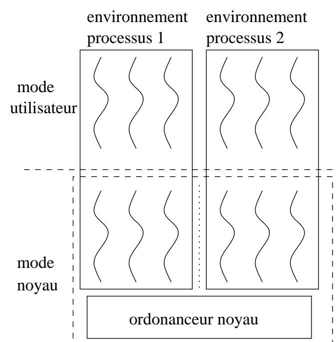
## Activités en mode noyau

▷ Chaque activité est vue comme une entité par le noyau.

Une activité noyau est associée à chaque activité.

▷ Les commutations entre les activités nécessitent un passage en mode noyau.

Gestion des activités par l'ordonnanceur noyau.




▷ Un appel à un service bloquant du noyau, ne bloque pas toutes les activités appartenant au même processus.

▷ Windows XP/2000, Solaris, Linux, Tru64 Unix, Mac OS X



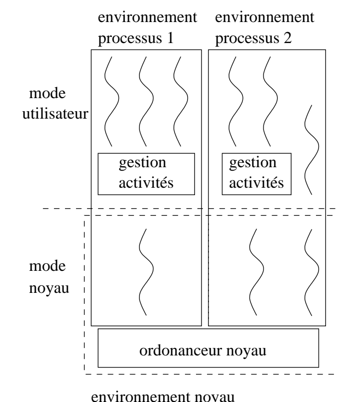


## Activités combinées

▷ Synthèse des deux approches précédentes.

▷ À chaque processus est associé un nombre variable d'activités noyau. Ce nombre peut être ajusté dynamiquement en fonction des besoins.

▷ L'unité d'ordonnancement est l'activité noyau.








## Modèles de Multi-threading

Ainsi, nous avons plusieurs modèles de multi-threading :

▷ Modèle : Plusieurs–à–Un

▷ Modèle : Un–à–Un

▷ Modèle : Plusieurs–à–Plusieurs

▷ Modèle : A deux niveaux

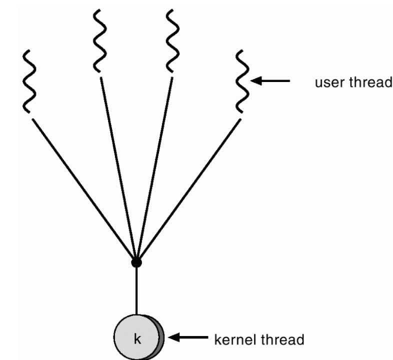




## Modèle Plusieurs–à–Un

▷ Plusieurs threads utilisateurs attachés à un seul thread noyau

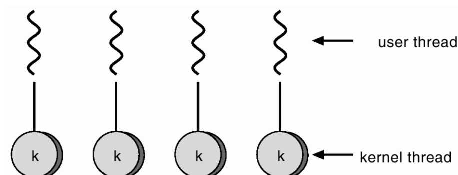
▷ Exemples : Solaris Green Threads, GNU Portable threads



## Modèle Un–à–Un

▷ Chaque thread utilisateur attaché à un thread noyau

▷ Exemples : Windows/NT/XP/2000, Linux, Solaris 9 et +





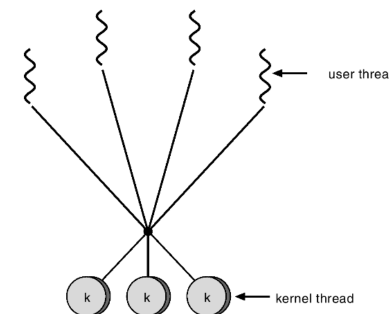



## Modèle Plusieurs–à–Plusieurs

▷ Permet à plusieurs threads utilisateurs d'être attachés à plusieurs threads noyaux

▷ Permet au système d'exploitation de créer un nombre suffisant de threads noyau.

▷ Exemples : Solaris < 9, Windows NT/2000 (package ThreadFiber)



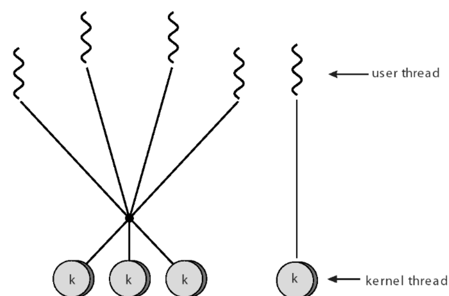





## Modèle à Deux niveaux

▷ Similaire à Plusieurs–à–Plusieurs, sauf qu'il permet à un thread utilisateur d'être **lié** à un thread noyau

▷ Exemples : IRIS, HP–UX, Tru64 Unix, Solaris < 9








## Politiques d'ordonnancement : objectifs

▷ utilisation maximale des ressources (CPU et périphériques) :
→ pourcentage du temps d'activité processeur,
→ surcoût minimum de la gestion de l'ordonnancement.

▷ performances utilisateur :
→ débit, en nombre de processus exécutés par période de temps,
→ temps de terminaison d'un processus,
→ temps de réponse : intervalle entre la soumission d'une requête et **le début** de la réponse (processus interactifs),
→ sûreté et équité : tous les processus doivent "avancer" et obtenir les ressources de calcul de façon équitable.

## Politiques d'ordonnancement : perspectives

▷ **Long terme** :
autorisation de création d'une activité ou d'un processus
→ éviter la saturation du système.

▷ **Moyen terme** :
un processus est échangé entre la mémoire secondaire et principale (*swap*).

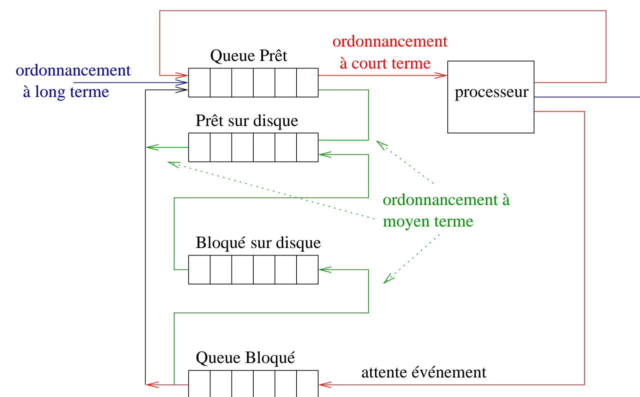
▷ **Court terme** :
mise en œuvre à chaque événement (appel système, temps, E/S) pour donner l'accès aux ressources de calcul.

**Ici, on ne considère que l'ordonnancement à court terme.**





## Politiques d'ordonnancement : perspectives








## Politiques d'ordonnancement : sans priorité

Pas de priorité : toutes les activités ont *a priori* les mêmes privilèges d'accès à la ressource processeur.

Critère de choix parmi les activités prêtes :

▷ Premier venu, premier servi (First Come, First Served)
Non préemptif, sélection=max(temps passé dans le système)

▷ Tourniquet (*Time sharing*)
Préemptif, bon temps de réponse

▷ Le plus court d'abord (Shortest Process Next)
Non préemptif, sélection=min(temps de service)

▷ Le plus près de la fin d'abord (Shortest Remaining Time)
Préemptif, sélection=min(temps de service - temps déjà exécuté)

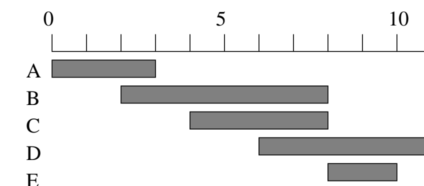




## Politiques d'ordonnancement : sans priorité

Exemples :

| Activités | date d'arrivée | durée |
|---|---|---|
| A | 0 | 3 |
| B | 2 | 6 |
| C | 4 | 4 |
| D | 6 | 5 |
| E | 8 | 2 |



## Politiques d'ordonnancement : sans priorité

Comparaison numérique des algorithmes :

| | Tâche | A | B | C | D | E | moyenne |
|---|---|---|---|---|---|---|---|
| | $t_{serv}$ | 3 | 6 | 4 | 5 | 2 | 4 |
| FCFS | $t_{term}$ | 3 | 7 | 9 | 12 | 12 | 8,60 |
| | $t_{term}/t_{serv}$ | 1,00 | 1,17 | 2,25 | 2,40 | 6,00 | 2,56 |
| Tourniquet | $t_{term}$ | 4 | 16 | 13 | 14 | 7 | 10,80 |
| | $t_{term}/t_{serv}$ | 1,33 | 2,67 | 3,25 | 2,80 | 3,50 | 2,71 |
| SPN | $t_{term}$ | 3 | 7 | 11 | 14 | 3 | 7,60 |
| | $t_{term}/t_{serv}$ | 1,00 | 1,17 | 2,75 | 2,80 | 1,50 | 1,84 |
| SRT | $t_{term}$ | 3 | 13 | 4 | 14 | 2 | 7,20 |
| | $t_{term}/t_{serv}$ | 1,00 | 2,17 | 1,00 | 2,80 | 1,00 | 1,59 |

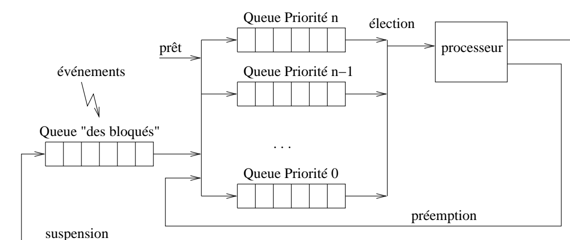




## Politiques d'ordonnancement : avec priorité

Priorités multiples : un ensemble d'activités prêtes par priorité possible.

**Une activité d'une priorité donnée n'a aucun accès au processeur tant qu'il existe des activités prêtes de priorité supérieure.**








## Politiques d'ordonnancement : avec priorité

Gestion des priorités :

- ▹ priorité fixe attribuée par le programmeur (ordonnancement dit *temps réel*)
- ▹ priorité décroissante avec le temps d'exécution (cf exemple priorité dégressive),
- ▹ priorité variable en fonction de l'état d'une activité.
- ▹ priorité variable en fonction de l'état d'avancement des différents processus (Fair-Share Scheduling).





## Politiques d'ordonnancement : avec priorité

**Ordonnancement à partage équitable**

Fair-Share Scheduling (FSS)

→ prendre en compte l'attribution des activités aux applications en cours sur la machine
→ équilibrer l'usage des ressources pour les différentes applications, quel que soit le nombre d'activités qui les composent.

Implanté sous une forme simplifiée dans certaines versions d'UNIX (SVR3, 4.3BSD)

## Politiques d'ordonnancement : avec priorité

### Ordonnancement à partage équitable (suite)

Calcul de la priorité $P_j$ d'une activité pour une tranche de temps $i$

- $i$ désigne la $i^{ème}$ tranche de temps
- $j$ désigne une activité appartenant à un groupe $k$ d'activités
- $U_j$ : temps d'utilisation du processeur par l'activité $j$
- $GU_k$ : temps d'utilisation du processeur par le groupe d'activité $k$
- $W_k$ : fraction de la ressource de calcul affectée à un groupe $k$

$$P_j(i) = Base_j + \frac{CPU_j(i-1)}{2} + \frac{GCPU_k(i-1)}{4.W_k}$$
$$CPU_j(i) = \frac{U_j(i)}{2} + \frac{CPU_j(i-1)}{2}$$
$$GCPU_k(i) = \frac{GU_k(i)}{2} + \frac{GCPU_k(i-1)}{2}$$

$CPU_j \simeq$ temps moyen d'utilisation du processeur par l'activité $j$
$GCPU_k \simeq$ temps moyen d'utilisation du processeur par le groupe d'activités $k$





## Politiques d'ordonnancement : avec priorité

### Ordonnancement à partage équitable : exemple

| | processus A | | | processus B | | | processus C | | |
|---|---|---|---|---|---|---|---|---|---|
| temps | priorité | proc. | groupe | priorité | proc. | groupe | priorité | proc. | groupe |
| 0 | 60 | 0 | 0 | 60 | 0 | 0 | 60 | 0 | 0 |
| | | 1 | 1 | | | | | | |
| | | ... | ... | | | | | | |
| | | 60 | 60 | | | | | | |
| 1 | 90 | 30 | 30 | 60 | 0 | 0 | 60 | 0 | 0 |
| | | | | | ... | ... | | | ... |
| | | | | | 60 | 60 | | | 60 |
| 2 | 74 | 15 | 15 | 90 | 30 | 30 | 75 | 0 | 30 |
| | | ... | ... | | | | | | |
| | | 75 | 75 | | | | | | |
| 3 | 96 | 37 | 37 | 74 | 15 | 15 | 67 | 0 | 15 |
| | | | | | | ... | | ... | ... |
| | | | | | | 75 | | 60 | 75 |
| 4 | 78 | 18 | 18 | 81 | 7 | 37 | 93 | 30 | 37 |
| | | ... | ... | | | | | | |
| | | 78 | 78 | | | | | | |
| 5 | 98 | 39 | 39 | 70 | 3 | 18 | 93 | 15 | 18 |

3 processus :
A, B et C

2 groupes :
Groupe 1: A
Groupe 2: B,C

$W_1$: 0.5
$W_2$: 0.5

Bases identiques : 60

Tranche de temps : 60

Séquence :
A,B,A,C,A,B...

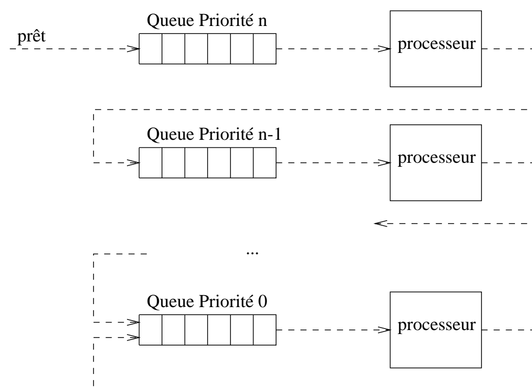
Remarque :
Ici, celui qui est sélectionné est celui qui a la valeur de $P_j(i)$ minimu.





## Politiques d'ordonnancement : avec priorité

### Ordonnancement avec queues de priorités dégressives








## Activités : mise en œuvre Unix Solaris (versions<9)

La notion d'activité n'existait pas à l'origine dans UNIX,
où l'unité d'ordonnancement était le processus.

Les activités ont été introduites dans UNIX SVR4, avec une révision des politiques d'ordonnancement du système.

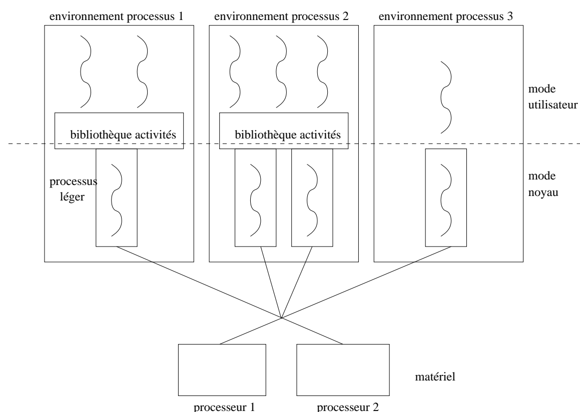
L'approche choisie dans Solaris était de type combiné. Les activités utilisateur sont gérées par des appels à une bibliothèque.

Les activités utilisateur sont affectées à des "processeurs virtuels", les processus légers (*lightweigth process*). Chaque processus léger est associé à une seule activité noyau. Le nombre de processus légers peut être adapté dynamiquement en fonction du degré de concurrence désiré.

Une activité utilisateur est liée à un ou plusieurs processus légers, qui s'exécutent dans l'environnement d'un processus.

**L'unité d'ordonnancement est le processus léger**

## Activités : mise en œuvre Unix Solaris (versions<9)








## Description des activités Unix Solaris (versions<9)

Dans le contexte du processus, on trouve un pointeur vers une liste de contextes de processus légers (c'est à dire d'activités noyau).

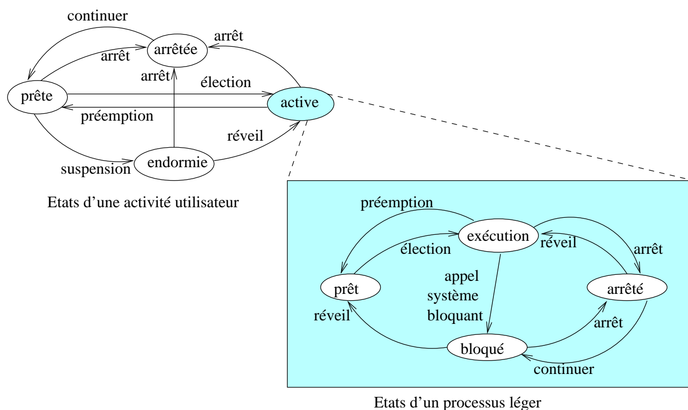
Contexte de processus légers :
- ▷ type `kthread_t` défini dans `/usr/include/sys/thread.h`,
- ▷ statistiques, contexte de l'activité noyau associée,
- ▷ accès au contexte du processus associé,
- ▷ environ 350 octets pour chaque contexte.

Contexte des activités utilisateur :
- ▷ registres processeurs, pile, masque de signaux, priorités d'ordonnancement,
- ▷ ne dépend pas directement du système d'exploitation, mais de la bibliothèque utilisée.





## Etats des activités et des processus léger Unix Solaris (versions<9)




Etats d'une activité utilisateur

Etats d'un processus léger





## Etats des activités et des processus léger Unix Solaris (versions<9)

### Activité utilisateur

- ▷ Une activité utilisateur active s'exécute lorsque le processus léger associé est actif. Mais la réciproque n'est pas vraie.
- ▷ L'état `endormie` correspond à une attente de synchronisation avec une autre activité utilisateur.
- ▷ L'état `arrêtée` est une mise en attente d'une activité, qui pourra éventuellement être réveillée par une autre.

### Processus léger (ou activité noyau)

- ▷ L'état `bloqué` correspondant à un appel système bloquant (E/S ou synchronisation inter-processus).

  ⇒ Une activité utilisateur peut être active alors que l'activité noyau associée est bloquée.

## Ordonnancement des activités Unix Solaris (versions<9)

Type : Tourniquet avec priorités, préemptif, 160 niveaux de priorité.

4 classes d'ordonnancement,
correspondant à une gestion différente des priorités :

- ▷ Classe "temps réel" : priorité fixe de 100 à 159. Ces activités sont prioritaires par rapport aux activités système.
- ▷ Classe "système" : priorité de 60 à 99. Utilisée pour les tâches du système.
- ▷ Classe "partage de temps" : priorité de 0 à 59. Utilisée pour les applications classiques.
- ▷ Classe "interactive" : priorité de 0 à 59. Utilisée pour les activités liées à une fenêtre graphique.

Le processeur est attribué à une activité pendant au maximum un quantum de temps. Le quantum de temps dépend de la priorité de l'activité (par exemple, 1 s pour la priorité 100, et 20 ms pour la priorité 59). Pour la classe "système", le quantum de temps est illimité.





## Ordonnancement des activités Unix Solaris (versions<9)

Dans le classe "partage de temps", une activité voit sa priorité temporairement augmentée :

- ▷ si elle est en attente d'événement d'entrées/sorties ou de synchronisation,
- ▷ si elle a été privée de la ressource processeur pendant trop longtemps.

Une activité voit sa priorité temporairement diminuée :

- ▷ si elle a épuisé complètement son quantum de temps.

Dans la classe "interactive", l'activité en cours dans la fenêtre graphique sélectionnée voit sa priorité augmentée.





## Activités : mise en œuvre Windows NT4

Modèle Un–à–Un :

- ▷ Toute activité Windows NT existe en mode noyau
- ▷ L'unité d'ordonnancement du noyau est l'activité

Pour implémenter le modèle Plusieurs–à–Plusieurs, la notion d'activités utilisateur (*fibres*) a été introduite dans NT, pour faciliter le portage d'applications UNIX.

Sous NT, on trouve généralement un processus multi-activités par application, à la différence d'UNIX où plusieurs processus participent souvent à une même application complexe.





## Description des activités Windows NT4

- ▷ Implémente donc le modèle Un–à–Un
- ▷ Chaque thread comporte :
  - ⋄ Un id
  - ⋄ Ensemble de registres
  - ⋄ Piles utilisateur et noyau séparés
  - ⋄ Espace de stockage de données séparé
- ▷ L'ensemble de registres, piles, et l'espace de stockage privé constituent le **contexte** d'un thread

## Description des activités Windows NT4

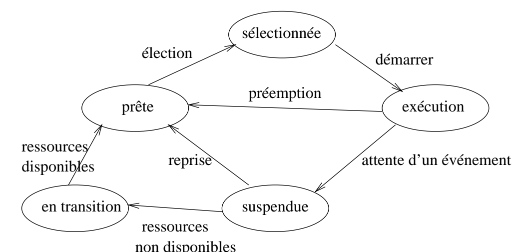
Les structures de données dun thread Windows NT4 sont :

- ▷ ETHREAD (executive thread block) : identité de l'activité et du processus associé, jeton d'accès sécurité, requête d'E/S en cours
- ▷ KTHREAD (kernel thread block) : ordonnancement et synchronisation (utilisé par le noyau uniquement)
- ▷ TEB (thread environment block) : dans l'espace d'adressage utilisateur, gestion de l'image mémoire





## Etats des activités Windows NT4




- ▷ En transition : l'environnement d'exécution n'est pas disponible en mémoire principale.
- ▷ Sélectionnée : état temporaire d'une activité avant d'accéder à une ressource processeur.
- ▷ Suspendue : activité en attente d'un événement.





## Algorithme d'ordonnancement Windows NT4

Algorithme de base : tourniquet avec priorités
→ une file d'attente des processus prêts par priorité

L'ordonnanceur est préemptif : une activité de haute priorité peut interrompre une activité de priorité inférieure avant la fin de sa tranche de temps.

En cas de préemption, l'activité suspendue est placée en tête de la liste de sa priorité.





## Algorithme d'ordonnancement Windows NT4 : priorités

32 niveaux de priorités, répartis en 2 classes :

- ▷ la classe temps réel, destinée principalement aux activités de services du système,
- ▷ la classe à priorité variable, destinée principalement aux applications.

La priorité des activités de classe temps réel reste fixe ; celle des activités de classe variable peut être temporairement modifiée par le système pour tenter de mieux utiliser les ressources de la machine.

Une activité voit sa priorité temporairement augmentée :

- ▷ si elle est en attente d'événement d'entrées/sorties ou de synchronisation,
- ▷ si elle a été privée de la ressource processeur pendant trop longtemps (plusieurs secondes).

## Algorithme d'ordonnancement Windows NT4 : "Quantum" de temps

Chaque activité dispose d'un *quantum* de temps d'accès au processeur.
La durée du quantum varie selon le type de la machine : de 10 ms à 200 ms.

L'avancement dans le quantum est codé par une valeur numérique, décrémentée périodiquement.

Lorsque le quantum est épuisé, une autre activité prête de même priorité est élue. L'activité précédente est placée en fin de liste des activités prêtes et son quantum est ré-initialisé.

Les activités associés au programme au premier plan (– interactif –) voient leur quantum augmenté automatiquement par le système.





## Programmation multi-threads

### Programmation multi-threads

*Quelques précautions (programmation système)*
*Ordonnancement*
*Principes*
*Threads* Posix
*Synchronisation*





## Quelques précautions

- ▷ **De la rigueur !**
  - ◇ *"C'est facile lorsque ça fonctionne !"*
    - ◦ Difficile d'expliquer les disfonctionnements *a posteriori*
  - ◇ Compiler le plus sévèrement possible
    - ◦ `$ g++ -W -Wall -pedantic -Werror ...`
    - ◦ Il est **toujours** possible d'éliminer un avertissement
  - ◇ Tester le succès ou l'échec des invocations
    - ◦ Lire attentivement les pages de manuel
    - ◦ En particulier les sections `RETURN VALUE` et `ERRORS`
  - ◇ Faciliter la portabilité
    - ◦ Vérifier l'origine (*ANSI-C*, *POSIX*, *SystemV*, *BSD* ...)
    - ◦ Éviter les extensions spécifiques à l'environnement
    - ◦ Ne pas compter sur un comportement spécifique





## Quelques précautions

- ▷ **Utilisation de** `errno` (`man 3 errno`)
  - ◇ `#include <errno.h>`
    `extern int errno;`
  - ◇ Indicateur positionné en cas d'erreur
    - ◦ Depuis les appels systèmes
    - ◦ Depuis quelques fonctions de bibliothèque
  - ◇ Un ensemble de constantes indiquant la nature de l'erreur
  - ◇ Utilisation classique
    - ◦ `errno=0;` (facultatif)
    - ◦ Effectuer l'appel
    - ◦ Si le résultat indique une erreur (`-1` généralement)
    - ◦ → Lire la valeur de `errno`

## Quelques précautions

▷ **Utilisation de** `errno`

- ◇ Description d'une erreur (`man 3 strerror`)
  ```
  #include <string.h>
  char * strerror(int errnum);
  ```
- ◇ Retourne une chaîne allouée statiquement décrivant l'erreur `errnum`
- ◇ Signaler une erreur (`man 3 perror`)
  ```
  #include <stdio.h>
  void perror(const char * msg);
  ```
- ◇ Écrit `msg` et une description de `errno` dans la sortie d'erreur
- ◇ `msg` peut être un pointeur nul





## Quelques précautions

▷ **Les appels sytèmes** *"lents"*

- ◇ Certains appels systèmes sont atomiques
  - ∘ Information immédiatement disponible dans le noyau
- ◇ D'autres peuvent prendre du temps (*"lents"*)
  - ∘ Il faut attendre les informations
- ◇ Les processus peuvent recevoir des signaux
- ◇ Si un processus reçoit un signal en cours d'appel système
  - ∘ Le processus est relancé
  - ∘ Les informations ne sont pas forcément disponibles !
  - ∘ L'appel système échoue
  - ∘ `errno` vaut `EINTR`
- ◇ Sans gravité → recommencer l'appel
- ◇ Précisé dans la section `ERRORS` du manuel de l'appel





## Quelques précautions : utilisation de `errno`

```
#include <sys/types.h>     /* utilisationErrno.c */
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
int status=EXIT_SUCCESS;
int fd=open("file.txt",O_RDONLY);
if(fd==-1)
  {
  perror("open()");
  status=EXIT_FAILURE;
  }
else
  {
  char buffer[256];
  ssize_t nb;
```

```
$ ./prog
open(): No such file or directory
$ echo Hello > file.txt
$ ./prog
Success ! --> Hello

$ chmod -r file.txt
$ ./prog
open(): Permission denied
$ rm file.txt
$ mkdir file.txt
$ ./prog
read(): Is a directory
$
```





## Quelques précautions : utilisation de `errno`

```
  do                                /* utilisationErrno.c (suite) */
    {
    nb=read(fd,buffer,255);
    } while((nb==-1)&&
            ((errno==EINTR)||(errno==EAGAIN)));
  if(nb==-1)
    {
    perror("read()");
    status=EXIT_FAILURE;
    }
  else
    {
    buffer[nb]='\0';
    fprintf(stderr,"Success ! --> %s\n",buffer);
    }
  close(fd);
  }
return(status);
}
```

## Quelques précautions : expérience vécue

▷ **Au delà du code strict**
- ◇ S'imaginer dans les pires conditions !
- ◇ Présentation, indentation
  - ○ 80 colonnes, espaces/tabulations ...
  - ○ *"Aurai-je toujours ma super-imprimante sous la main ?"*
- ◇ Capitaliser son expérience
  - ○ Une difficulté rencontrée → une nouvelle règle pour l'éviter
  - ○ Un problème de portabilité → retenir la solution commune
- ◇ De nombreuses opérations s'utilisent dans des contextes variés
  - ○ S'efforcer de les effectuer de la même façon
  - ○ Utilisation systématique de *patrons* de programmation
- ◇ Pas de règles universelles mais de l'homogénéité





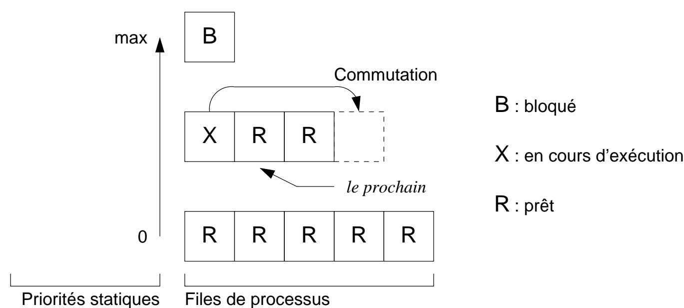
## L'ordonnancement des processus

▷ **Contrôlé par les fonctionnalités de** `sched.h`
- ◇ *"Comment choisir le processus à exécuter ?"*
- ◇ *"Jusqu'à quand ?"*
- ◇ Choix d'un mode d'ordonnancement par processus (pas pour l'ensemble)
- ◇ Géré par files de priorités (statiques)
- ◇ Éventuellement des priorités dynamiques





## L'ordonnancement des processus




B : bloqué

X : en cours d'exécution

R : prêt

▷ **Pour qu'un processus soit exécuté**
- ◇ Aucun processus de priorité statique > ne doit être prêt
- ◇ Il doit être le premier de sa file





## L'ordonnancement des processus

▷ **Commutations implicites**
- ◇ Appels systèmes bloquants
- ◇ Préemptions par l'ordonnanceur (horloge)
- ◇ Terminaison du processus

▷ **Commutations explicites**
- ◇ Appel système `sched_yield()` (`man 2 sched_yield`)
  `int sched_yield(void);`
- ◇ Cède le processeur à un autre processus
- ◇ Passe à la fin de la file de sa priorité statique
- ◇ Si aucun autre processus de priorité statique ≥ n'est prêt → continuer sans commuter
- ◇ Retour : `0` si ok, `-1` si erreur (cause ?)

## L'ordonnancement des processus

▷ **Trois modes d'ordonnancement** (`man 2 sched_setscheduler`)

◇ Choix du mode par processus, pas pour l'ensemble

◇ Choix d'une priorité statique (mini./maxi. dépendantes du système et du mode d'ordonnancement: `man 2 sched_get_priority_max` et `man 2 sched_get_priority_min`)

◇ `SCHED_FIFO`

- Mode à tendance *"temps-réel"*
- Uniquement des priorités statiques > `0`
- Pas de préemption par horloge → Multi-tâches coopératif
  Commutation si : `sched_yield`, bloqué par E/S, ou encore s'il existe un processus de priorié supérieure.
- Risque de bloquage (pour priorités statiques ≤) !
- Nécessite des privilèges (`root`)





## L'ordonnancement des processus

▷ **Trois modes d'ordonnancement**

◇ `SCHED_RR` (*Round Robin*)

- ≡ `SCHED_FIFO` + préemption par une horloge
- Mode à tendance *"temps-réel"*
- Uniquement des priorités statiques > `0`
- Préemption par une horloge → Multi-tâches préemptif
  Commutation régulière
- Risque de bloquage (pour priorités statiques <) !
- Nécessite des privilèges (`root`)





## L'ordonnancement des processus

▷ **Trois modes d'ordonnancement**

◇ `SCHED_OTHER`

- Pour la majorité des processus
- Optimisé pour un temps de réponse et un rendement global (pas un processus au détriment des autres)
- Priorité statique = `0`
- Préemption par une horloge → Multi-tâches préemptif
  Commutation régulière
- Repose sur une priorité *dynamique* dépendante du système (une quantité fixe et une autre évoluant selon les commutations)
- Procédé spécifique à chaque système
  (quelquefois ≡ `SCHED_RR` avec la priorité statique `0`)





## Généralités sur les threads

▷ **Donner plusieurs activités à un même processus**

◇ Mener plusieurs traitements bloquants

◇ Gagner du temps sur une machine multi-processeurs

▷ **Plus efficace que plusieurs processus**

◇ Commutation plus efficace (*processus légers*)

◇ Espace d'adressage commun (communication simplifiée)

▷ **Informations propres à chaque thread**

◇ Pile d'exécution, valeurs des registres

▷ **Informations partagées au sein du processus**

◇ Code, données, descripteurs de fichiers

▷ **Partage dépendant de l'implémentation**

◇ Signaux, propriétés . . .

## Plusieurs implémentations

- **Espace noyau**
  - Gérés par le système (comme des processus)
  - Propriétés bien séparées
  - Permet l'utilisation de plusieurs processeurs
  - Priorités et modes d'ordonnancement des processus
  - Commutation des threads $\simeq$ commutation des processus
- **Espace utilisateur**
  - Propriétés communes car inconnu du noyau
  - Ordonnancement *"applicatif"* à l'intérieur du processus
  - Appel bloquant dans un thread
    → risque de blocage du processus
  - Utilisation d'un seul processeur
  - Commutations plus légères que pour les processus





## Plusieurs implémentations

- **Les Pthreads**
  - Norme *Posix.1c*, c'est **la** référence !
  - Fonctionnalités de base portables
  - Influence sur les fonctionnalités habituelles (mono-tâche)
    → très dépendant de la plate-forme
- **Quelques exemples**
  - *Linux* : *Pthread* (noyau), *Pth* (utilisateur)
  - *IRIX* : *Pthread* (utilisateur), *sproc* (noyau)
  - *Solaris* : *Pthread* (noyau)
  - *SunOS* : *Pthread* (utilisateur), *LWP* (noyau)
  - *Windows* : threads noyau
  - *Java* : ???, très variable d'une plate-forme à l'autre
  - Certaines sont hybrides (noyau/utilisateur)





## Précautions

- **Exécutions indépendantes**
  - Nécessité d'une synchronisation explicite
- **Espace d'adressage commun**
  - Accès concurrents à des ressources communes
    - Mécanismes d'exclusion mutuelle
  - Écriture de traitements réentrants
    - Pas de données statiques ou globales
    - Arguments supplémentaires
  - Choix de primitives système réentrantes
    - Fonction ayant l'extension `_r` (*Posix.1c*)
    - Ex : `asctime()` et `asctime_r()`





## Précautions

- **Exemple de fonction non réentrante**
  - Contenu de `buffer` indéterminé si invocations simultanées

```
const char * writeHexa(int i)
{
static char buffer[0x10];
sprintf(buffer,"0x%.8x",i);
return(buffer);
}
```

- **Version réentrante de cette fonction**
  - Chaque invocation utilise des données différentes (transmises)

```
char * writeHexa_r(int i,char * buffer)
{
sprintf(buffer,"0x%.8x",i);
return(buffer);
}
```

## Conventions sur le nomage des *Pthreads*

▷ `#include <pthread.h>`

▷ **Les types :** `pthread[_objet]_t`

◇ *objet* :

- `attr`, `mutex` ou `cond`
- *thread* si omis

▷ **Les fonctions :** `pthread[_objet]_operation[_np]`

◇ *objet* : (voir les types)

◇ *operation* : traitement concernant le type désigné

- `init`, `destroy` ...
- `create`, `exit`, `join` ...
- `lock`, `unlock`, `signal`, `broadcast` ...

◇ L'extension `_np` signale un traitement non portable (spécifique à l'implémentation courante)





## Développer avec les *Pthreads*

▷ **Signalement des erreurs**

◇ Pour la majorité des fonctions `pthread` :

- `errno` n'est pas utilisé
- Retour valant `0` → ok
- Retour non nul → code d'erreur interprété comme `errno`

▷ **Compilation avec la macro** `_REENTRANT`

◇ `$ cc -c -D_REENTRANT prog.c`

- Influence sur les `.h` standards
- Implémentation différente de certains services

▷ **Édition de liens avec la bibliothèque** `pthread`

◇ `$ cc -o prog prog.o -lpthread`





## Identification de l'activité

▷ **Le processus dans sa globalité**

◇
```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);
```

◇ 1 PID par thread pour les threads en mode noyau !

▷ **Le thread courant**

◇ `pthread_t pthread_self(void);`

▷ **Le test d'égalité**

◇ `pthread_t` est un type opaque → pas de comparaison directe !

◇ `int pthread_equal(pthread_t t1,pthread_t t2);`

◇ Résultat non nul si `t1` ≡ `t2`, `0` sinon





## Création d'un thread

▷ **La fonction** `pthread_create()` (`man 3 pthread_create`)

◇
```
int pthread_create(pthread_t * id,
                   pthread_attr_t * attr,
                   void * (*fct)(void *),
                   void * fctArg);
```

◇ Crée un thread exécutant `fct` avec l'argument `fctArg`

◇ Stocke son identifiant dans `id`

◇ Le thread a les propriétés décrites par `attr` (propriétés par défaut si pointeur nul)

◇ Retour : `0` si ok, non nul si erreur (`EAGAIN`)

◇ Causes d'erreur :

- `PTHREAD_THREADS_MAX` atteint
- Plus assez de ressources système

## Terminaison d'un thread

▷ **Fin de sa fonction**
  ◇ Résultat transmis par la valeur de retour

▷ **La fonction** `pthread_exit()` (`man 3 pthread_exit`)
  ◇ `void pthread_exit(void * result);`
  ◇ Termine le thread courant en retournant `result`
  ◇ Résultat lisible par `pthread_join()` (voir plus loin)
  ◇ Appelle les traitements de `pthread_cleanup_push()` (voir plus loin)

▷ **Fin du programme principal**
  ◇ Fin de `main()` (ou `exit()`) ou recouvrement (par `exec()`) → destruction de tous les threads (il reste une activité)
  ◇ `pthread_exit()` dans `main()` → attente de tous les threads





## Attente d'un thread

▷ **La fonction** `pthread_join()` (`man 3 pthread_join`)
  ◇ `int pthread_join(pthread_t th,void ** result);`
  ◇ Attendre la terminaison de `th` et obtenir son résultat dans `result`
  ◇ `*result` vaut `PTHREAD_CANCELED` si `th` a été annulé (voir plus loin)
  ◇ Les ressources de `th` sont libérées
  ◇ Un seul `pthread_join()` sur un thread donné
  ◇ Le thread attendu ne doit pas être détaché (voir plus loin)
  ◇ Un thread ne peut s'attendre lui même
  ◇ Retour : `0` si ok, non nul si erreur





## Créer et attendre un thread

```
#include <pthread.h>            /* waitThread.c */
#include <stdio.h>
void * task(void * data)
{
int * nb=(int *)data; int i;
fprintf(stderr,"start 1\n");
for(i=0;i<*nb;i++) { } fprintf(stderr,"end 1\n");
return((void *)0);
}
int main(void)
{
pthread_t th;
int nb=100000000; int i;
void * result;
if(pthread_create(&th,(pthread_attr_t *)0,task,&nb))
  { fprintf(stderr,"Pb create()\n"); return(1); }
fprintf(stderr,"start 2\n");
for(i=nb/2;i<nb;i++) { } fprintf(stderr,"end 2\n");
if(pthread_join(th,&result))
  { fprintf(stderr,"Pb join()\n"); return(1); }
fprintf(stderr,"quit\n");
return(0);
}
```

```
$ ./prog
start 2
start 1
end 2
end 1
quit
$
```





## Effet de `sched_yield()`

```
#include <pthread.h>    /* effetYield.c */
#include <sched.h>
#include <stdio.h>
void * task(void * data)
{ int i;
  for(i=0;i<1000000;i++)
  {
   fprintf(stderr,"%d",*((int *)data));
  /* sched_yield(); */
  }
return((void *)0);
}
int main(void)
{
int n1=1,n2=2;
pthread_t t1,t2; void * result;
if(pthread_create(&t1,(pthread_attr_t *)0,task,&n1))
  { fprintf(stderr,"Pb thread 1\n"); return(1); }
if(pthread_create(&t2,(pthread_attr_t *)0,task,&n2))
  { fprintf(stderr,"Pb thread 2\n"); return(1); }
pthread_join(t1,&result); pthread_join(t2,&result);
return(0);
}
```

```
$ ./prog # sans sched_yield()
1111111111111111111111111111111111111111
1111 ... 1112222222222222222222222222222222
2222222222222222222222 ... 11111111111111
111111111111111111111111111111111111 ... 11
1111122222222 ... 22222222222222222222222
2222
$
$ ./prog # avec sched_yield()
1212121212121212121212121212121212121212121
2121212121212121212121212 ... 2121212121212
1212
$
```

## Annulation d'un thread

▷ **La fonction** `pthread_cancel()` (`man 3 pthread_cancel`)
- `int pthread_cancel(pthread_t th);`
- Demande au thread `th` de se terminer
- Pas forcément pris en compte immédiatement (voir plus loin)
- Retour : `0` si ok, non nul si erreur

▷ **La fonction** `pthread_testcancel()` (`man 3 pthread_testcancel`)
- `void pthread_testcancel(void);`
- Le thread courant teste s'il a reçu une demande d'annulation
- Le thread se termine à ce point si le test est positif
- Le retour du thread vaut `PTHREAD_CANCELED`
- Il existe d'autres points d'annulation (certains appels systèmes, certaines fonctions *pthread* … dépend de l'implémentation)





## Annulation d'un thread

▷ **La fonction** `pthread_setcancelstate()` (`man 3 pthread_setcancelstate`)
- `int pthread_setcancelstate(int state,int * oldState);`
- Change l'état d'annulation du thread courant à `state`
- Indique l'ancien état si `oldState` est non nul
- `PTHREAD_CANCEL_ENABLE` : annulations autorisées (défaut)
- `PTHREAD_CANCEL_DISABLE` : annulations ignorées
- Retour : `0` si ok, non nul si erreur





## Annulation d'un thread

▷ **La fonction** `pthread_setcanceltype()` (`man 3 pthread_setcanceltype`)
- `int pthread_setcanceltype(int type,int * oldType);`
- Change le type d'annulation du thread courant à `type`
- Indique l'ancien type si `oldType` est non nul
- `PTHREAD_CANCEL_DEFERRED` :
  annulation prise en compte aux points d'annulation (défaut)
- `PTHREAD_CANCEL_ASYNCHRONOUS` :
  annulation prise en compte immédiatement (dangereux !)
- Retour : `0` si ok, non nul si erreur





## Les attributs d'un thread

▷ **Création des attributs** (`man 3 pthread_attr_init`)
- `int pthread_attr_init(pthread_attr_t * attr);`
- Initialise à leur valeur par défaut les attributs désignés par `attr`
  - `detachstate` : `PTHREAD_CREATE_JOINABLE`
  - `schedpolicy` : `SCHED_OTHER`
  - `schedparam` : `0` (priorité)
  - `inheritsched` : `PTHREAD_EXPLICIT_SCHED`
  - `scope` : `PTHREAD_SCOPE_SYSTEM` (dépend de la plate-forme)
- Modifications éventuelles avant l'appel à `pthread_create()`
- Retour : `0` si ok, non nul sinon (cause ?)

Après création avec `pthread_create()`, les attributs d'un thread sont modifiables avec : `pthread_detach`, `pthread_setschedpolicy`, `pthread_setschedparam`.
Le nombre d'activités "noyau" peut, éventuellement, être modifié avec : `pthread_setconcurrency`

## Les attributs d'un thread

▷ **Destruction des attributs** (`man 3 pthread_attr_destroy`)

- ⋄ `int pthread_attr_destroy(pthread_attr_t * attr);`
- ⋄ Libère les attributs désignés par `attr`
- ⋄ Peut avoir lieu après `pthread_create()`
- ⋄ Les valeurs sont recopiées à la création du thread
- ⋄ Retour : `0` si ok, non nul sinon (cause ?)





## Les attributs d'un thread

▷ **Le détachement d'un thread**
(`man 3 pthread_attr_setdetachstate`)

- ⋄ `int pthread_attr_setdetachstate(`
  `pthread_attr_t * attr,int state);`
- ⋄ Interprétation de `state` :
  - ∘ `PTHREAD_CREATE_JOINABLE` :
    libération des ressources après `pthread_join()`
  - ∘ `PTHREAD_CREATE_DETACHED` :
    libération des ressources dès la terminaison du thread
- ⋄ Retour : `0` si ok, non nul sinon
- ⋄ Intervention sur un thread en cours
  - ∘ `int pthread_detach(pthread_t th);`
  - ∘ Le thread désigné passe dans l'état détaché
  - ∘ Retour : `0` si ok, non nul sinon (inconnu ou déjà détaché)





## Les attributs d'un thread

▷ **Le mode d'ordonnancement d'un thread**
(`man 3 pthread_attr_setschedpolicy`)
(`man 3 pthread_attr_setschedparam`)
(`man 3 pthread_attr_setinheritsched`)

- ⋄ `int pthread_attr_setschedpolicy(`
  `pthread_attr_t * attr,`
  `int policy);`
- ⋄ `int pthread_attr_setschedparams(`
  `pthread_attr_t * attr,`
  `const struct sched_param * params);`
- ⋄ Voir l'ordonnancement des processus
  - ∘ `schedpolicy` : `SCHED_FIFO`, `SCHED_RR`, `SCHED_OTHER`
  - ∘ `schedparams` : champ `sched_priority`





## Les attributs d'un thread

▷ **Le mode d'ordonnancement d'un thread**

- ⋄ `int pthread_attr_setinheritsched(`
  `pthread_attr_t * attr,`
  `int inherit);`
- ⋄ Transmission des paramètres d'ordonnancement
  - ∘ `PTHREAD_INHERIT_SCHED` :
    reprendre ceux du thread parent
  - ∘ `PTHREAD_EXPLICIT_SCHED` :
    initialisation explicite (ou valeur par défaut)
- ⋄ Concerne uniquement `schedpolicy` et `schedparams`
- ⋄ Retour : `0` si ok, non nul sinon

## Les attributs d'un thread

▷ **L'implémentation du thread** (`man 3 pthread_attr_setscope`)

- `int pthread_attr_setscope(pthread_attr_t * attr,int scope);`
- Interprétation de `scope` :
  - `PTHREAD_SCOPE_SYSTEM` : thread noyau
  - `PTHREAD_SCOPE_PROCESS` : thread utilisateur
- Choix réellement possible pour les implémentations hybrides
- Retour : `0` si ok, non nul sinon





## Attributs/Annulation d'un thread

```
#include <pthread.h>                /* attrThread.c */
#include <stdio.h>

void * task(void * data)
{
int * flag=(int *)data;
unsigned long i;
// pthread_setcancelstate(PTHREAD_CANCEL_ENABLE,(int *)0);
pthread_setcancelstate(PTHREAD_CANCEL_DISABLE,(int *)0);
// pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS,(int *)0);
pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED,(int *)0);
fprintf(stderr,"task begin\n");
for(i=0;i<200*1000*1000;i++) {}
(*flag)++;
pthread_testcancel();
(*flag)++;
fprintf(stderr,"task end\n");
return((void *)0);
}
```





## Attributs/Annulation d'un thread

```
/* attrThread.c (suite) */

int main(void)
{
pthread_attr_t attr;
pthread_t th;
int flag=0; int pb=0;
unsigned long i;
if(!pb) pb|=pthread_attr_init(&attr);
if(!pb) pb|=pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED);
if(!pb) pb|=pthread_create(&th,&attr,task,&flag);
if(!pb) pb|=pthread_attr_destroy(&attr);
for(i=0;i<100*1000*1000;i++) {}
pthread_cancel(th);
for(i=0;i<200*1000*1000;i++) {}
fprintf(stderr,"quit (flag=%d)\n",flag);
return(pb);
}
```

```
$ ./prog # CANCEL DISABLE
task begin
task end
quit (flag=2)
$
$ ./prog # CANCEL ENABLE & DEFERRED
task begin
quit (flag=1)
$
$ ./prog # CANCEL ENABLE & ASYNCHRONOUS
task begin
quit (flag=0)
$
```





## La pile de nettoyage

▷ **Empiler un traitement** (`man 3 pthread_cleanup_push`)

- `int pthread_cleanup_push(void (*fptr)(void *), void * arg);`
- Macro contenant `{`
- Empile une fonction et son argument
- Appel automatique (en dépilant) à la fin du thread (fin de fonction, `pthread_exit()` ou `pthread_cancel()`)
- Permet de fermer proprement ce qui a été initialisé

▷ **Dépiler un traitement** (`man 3 pthread_cleanup_pop`)

- `int pthread_cleanup_pop(int execute);`
- Macro contenant `}`
- Dépile une fonction et l'appelle si `execute` est non nul
- Doit être associé à un `pthread_cleanup_push()` (même bloc)

## La pile de nettoyage

```
{                                           /* pushPop.c */
// ...
void * data=malloc(0x1000);
if(data)
  {
  FILE * input;
  pthread_cleanup_push(free,data);
  input=fopen("file.txt","r");
  if(input)
    {
    pthread_cleanup_push((void (*)(void *))fclose,input);
    doSomethingWithThat(input,data);
    pthread_cleanup_pop(1); // fclose(input)
    }
  pthread_cleanup_pop(1); // free(data)
  }
// ...
}
```





## Réaction aux signaux

- ▷ **Émission explicite en interne**
  - ◇ `int pthread_kill(pthread_t th,int signum);`
  - ◇ Reçu par le thread désigné
- ▷ **Émission implicite par le système** (`SIGSEGV`, `SIGBUS` ...)
  - ◇ Mode noyau : reçu par le thread en cause
  - ◇ Mode utilisateur : reçu par le processus
- ▷ **Émission explicite depuis l'extérieur**
  - ◇ Mode noyau : reçu par le thread désigné par le PID
  - ◇ Mode utilisateur : reçu par un des threads (lequel ?)
- ▷ **Appel** `pthread` **depuis un gestionnaire → risque de blocage !**
- ▷ → **Éviter d'utiliser les signaux avec les threads !**





## Réaction à `fork()`

- ▷ **Comportement normal d'un** `fork()`
  - ◇ Les données des différents threads existent dans l'enfant
  - ◇ Le processus enfant n'a qu'une activité : le thread qui l'a créé
  - ◇ *"Que deviennent ces données, les verrous ... ?"*
- ▷ **La fonction** `pthread_atfork()` (`man 3 pthread_atfork`)
  - ◇ 
```
int pthread_atfork(void (*prepare)(void),
                   void (*parent)(void),
                   void (*child)(void));
```
  - ◇ Enregistre des traitements à effectuer automatiquement :
    - ○ avant le `fork()` (`prepare`)
    - ○ après le `fork()` dans le parent (`parent`)
    - ○ après le `fork()` dans l'enfant (`child`)
- ▷ **Éviter d'utiliser** `fork()` **avec les threads** (sauf pour `exec()`)





## Synchronisation

- ▷ **Attendre la fin d'un thread**
  - ◇ `pthread_join()` (déjà vu)
- ▷ **Effectuer un traitement unique**
  - ◇ Plusieurs threads ont besoin qu'un traitement soit effectué
  - ◇ Celui-ci doit être effectué une seule fois
- ▷ **Sémaphores d'exclusion mutuelle** (verrous)
  - ◇ Limiter l'accès à une donnée
  - ◇ Un seul accès à la fois
- ▷ **Variables conditions**
  - ◇ Attendre qu'une condition soit vérifiée dans un autre thread
  - ◇ Blocage du traitement en attendant

## Les traitements uniques

- ▷ **Synchronisation autour d'un** `pthread_once_t` (`man 3 pthread_once`)
  - ◇ `int pthread_once(pthread_once_t * control, void (*fptr)(void));`
  - ◇ Avant usage, `control` doit être initialisé à `PTHREAD_ONCE_INIT`
  - ◇ Premier accès à `control` → appel de la fonction `fptr`
  - ◇ Accès ultérieurs à `control` → aucun effet
  - ◇ Retour : toujours `0` !





## Les traitements uniques

```
#include <pthread.h>     /* once.c */
#include <stdio.h>

void initFunc(void)
{
fprintf(stderr,"initFunc() in %d\n",
              (int)pthread_self()); // ugly !
}

void * task(void * data)
{
unsigned long i;
fprintf(stderr,"begin task(%p) in %d\n",
              data,(int)pthread_self()); // ugly !
for(i=0;i<50*1000*1000;i++) {}
pthread_once((pthread_once_t *)data,initFunc);
for(i=0;i<50*1000*1000;i++) {}
fprintf(stderr,"end task(%p) in %d\n",
              data,(int)pthread_self()); // ugly !
return((void *)0);
}
```

```
$ ./prog
begin task(bffff744) in 1026
begin task(bffff748) in 2051
begin task(bffff744) in 3076
begin task(bffff748) in 4101
begin task(bffff744) in 5126
begin task(bffff748) in 6151
initFunc() in 1026
initFunc() in 2051
end task(bffff744) in 1026
end task(bffff744) in 3076
end task(bffff744) in 5126
end task(bffff748) in 4101
end task(bffff748) in 2051
end task(bffff748) in 6151
$
```





## Les traitements uniques

```
int main(void)              /* once.c (suite) */
{
pthread_once_t ctrl1=PTHREAD_ONCE_INIT;
pthread_once_t ctrl2=PTHREAD_ONCE_INIT;
pthread_t th[6];
int i, j;
for(i=0;i<6;i++)
  {
  if(pthread_create(&th[i],(pthread_attr_t *)0,task,i%2 ? &ctrl1 : &ctrl2))
    { fprintf(stderr,"Pb pthread_create()\n"); return(1); }
  }
for(j=0;j<6;j++)
  {
  void * result;
  if(pthread_join(th[j],&result))
    { fprintf(stderr,"Pb pthread_join()\n"); return(1); }
  }
return(0);
}
```





## Les sémaphores d'exclusion mutuelle

- ▷ **Création d'un verrou** (`man 3 pthread_mutex_init`)
  - ◇ Type : `pthread_mutex_t`
  - ◇ Initialisation statique
    - ○ `pthread_mutex_t myMutex=PTHREAD_MUTEX_INITIALIZER;`
  - ◇ Initialisation dynamique
    - ○ `int pthread_mutex_init(pthread_mutex_t * mtx, pthread_mutexattr_t * attr);`
    - ○ Généralement `attr` est nul (initialisation par défaut)
- ▷ **Destruction d'un verrou** (`man 3 pthread_mutex_destroy`)
  - ◇ `int pthread_mutex_destroy(pthread_mutex_t * mtx);`
  - ◇ Le verrou n'est plus utilisable
  - ◇ Retour : `0` si ok, non nul si erreur
  - ◇ Erreur si le verrou est monopolisé → erreur `EBUSY`

## Les sémaphores d'exclusion mutuelle

▷ **Opération de verrouillage** (`man 3 pthread_mutex_lock`)
- ◇ `int pthread_mutex_lock(pthread_mutex_t * mtx);`
- ◇ Si le verrou est libre
  - ○ il est monopolisé par le thread courant
  - ○ le thread courant poursuit son traitement
- ◇ Si le verrou n'est pas libre
  - ○ le thread courant est bloqué jusqu'à la libération du verrou
- ◇ Retour : `0` si ok, non nul si erreur





## Les sémaphores d'exclusion mutuelle

▷ **Opération de déverrouillage** (`man 3 pthread_mutex_unlock`)
- ◇ `int pthread_mutex_unlock(pthread_mutex_t * mtx);`
- ◇ Libère le verrou
- ◇ Si des threads sont bloqués en attente sur ce verrou
  - ○ l'un d'eux est débloqué et monopolise le verrou
- ◇ Retour : `0` si ok, non nul si erreur

▷ **Tentative de verrouillage** (`man 3 pthread_mutex_trylock`)
- ◇ `int pthread_mutex_trylock(pthread_mutex_t * mtx);`
- ◇ Si le verrou est libre
  - ○ il est monopolisé par cet appel qui retourne `0`
- ◇ Si le verrou n'est pas libre
  - ○ cet appel retourne immédiatement un résultat non nul
  - ○ il ne faut pas utiliser les données protégées par le verrou





## Les sémaphores d'exclusion mutuelle

```
#include <pthread.h>        /* mutex.c */
#include <stdio.h>

void * task(void * data)
{
pthread_mutex_t * mtx=(pthread_mutex_t *)data; int n;
fprintf(stderr,"begin task() in %d\n",
               (int)pthread_self()); // ugly cast !
for(n=0;n<2;n++)
 { unsigned long i;
   pthread_mutex_lock(mtx); // begin critical section
   fprintf(stderr,"  thread %d -->\n",
                (int)pthread_self()); // ugly cast !
   for(i=0;i<10*1000*1000;i++) {}
   fprintf(stderr,"  thread %d <--\n",
                (int)pthread_self()); // ugly cast !
   pthread_mutex_unlock(mtx); // end critical section
  }
fprintf(stderr,"end task() in %d\n",
               (int)pthread_self()); // ugly cast !
return((void *)0);
}
```

```
$ ./prog
begin task() in 1026
  thread 1026 -->
begin task() in 2051
begin task() in 3076
  thread 1026 <--
  thread 2051 -->
  thread 2051 <--
  thread 3076 -->
  thread 3076 <--
  thread 1026 -->
  thread 1026 <--
  thread 3076 -->
end task() in 1026
  thread 3076 <--
  thread 2051 -->
  thread 2051 <--
end task() in 2051
end task() in 3076
$
```





## Les sémaphores d'exclusion mutuelle

```
int main(void)            /* mutex.c (suite) */
{
pthread_mutex_t mtx=PTHREAD_MUTEX_INITIALIZER;
pthread_t th[3];
int i;
void * result;
for(i=0;i<3;i++)
  {
  if(pthread_create(&th[i],(pthread_attr_t *)0,task,&mtx))
    { fprintf(stderr,"Pb create\n"); return(1); }
  }
for(i=0;i<3;i++)
  {
  if(pthread_join(th[i],&result))
    { fprintf(stderr,"Pb join\n"); return(1); }
  }
if(pthread_mutex_destroy(&mtx))
  { fprintf(stderr,"Pb mutex destroy\n"); return(1); }
return(0);
}
```

## Les variables conditions

▷ **Création d'une condition** (`man 3 pthread_cond_init`)

◇ Type : `pthread_cond_t` (doit être associé à un `mutex`)

◇ Initialisation statique

∘ `pthread_cond_t myCond=PTHREAD_COND_INITIALIZER;`

◇ Initialisation dynamique

∘
```
int pthread_cond_init(pthread_cond_t * cond,
                      pthread_condattr_t * attr);
```

∘ Généralement `attr` est nul (initialisation par défaut)

▷ **Destruction d'une condition** (`man 3 pthread_cond_destroy`)

◇ `int pthread_cond_destroy(pthread_cond_t * cond);`

◇ La condition n'est plus utilisable

◇ Retour : `0` si ok, non nul si erreur

◇ Erreur si la condition est utilisée → erreur `EBUSY`





## Les variables conditions

▷ **Attente d'une condition** (`man 3 pthread_cond_wait`)

◇
```
int pthread_cond_wait(pthread_cond_t * cond,
                      pthread_mutex_t * mtx);
```

◇ Bloque le thread courant
→ débloqué quand une modification est signalée sur `cond`
→ évite l'attente active

◇ `cond` n'a aucune valeur logique ! (sert à la synchronisation)

◇ `mtx` doit être monopolisé avant et libéré après

◇ Interruptible par les signaux → relance

◇ Démarche usuelle :

```
pthread_mutex_lock(&mtx);
while(!conditionIsSatisfied())
  pthread_cond_wait(&cond,&mtx);
pthread_mutex_unlock(&mtx);
```





## Les variables conditions

▷ **Attente temporisée d'une condition**
(`man 3 pthread_cond_timedwait`)

◇
```
int pthread_cond_timedwait(pthread_cond_t * cond,
                           pthread_mutex_t * mtx,
                           const struct timespec * date);
```

◇ Même principe que `pthread_cond_wait()`

◇ Renvoie `ETIMEDOUT` si `date` est dépassée

◇ `date` est une limite, pas un délai !

∘ Prendre la date courante (`time()`, `gettimeofday()`)

∘ Ajouter un délai (structure `timespec` de `nanosleep()`)

∘ Basé sur le temps universel (pas de fuseau horaire)





## Les variables conditions

▷ **Signaler une condition** (`man 3 pthread_cond_broadcast`)

◇ `int pthread_cond_broadcast(pthread_cond_t * cond);`

◇ Débloque tous les threads attendant la condition `cond`

◇ Le verrou associé à `cond` doit être monopolisé avant et libéré après

◇ Démarche usuelle :

```
pthread_mutex_lock(&mtx);
/* make this condition become true */
pthread_cond_broadcast(&cond);
pthread_mutex_unlock(&mtx);
```

◇ `int pthread_cond_signal(pthread_cond_t * cond);`

- Ne débloque qu'un thread parmi ceux qui attendent `cond`
- Un peu plus efficace que `pthread_cond_broadcast`
- Bien moins général (incohérence si plusieurs threads en attente !)

## Les variables conditions

```
#include <pthread.h>     /* cond.c */
#include <stdio.h>

pthread_mutex_t mtx;
pthread_cond_t cond;
unsigned long x,y;

void * task(void *)
{
pthread_mutex_lock(&mtx);
while(x<=y)
  {
  fprintf(stderr,"until now %lu <= %lu\n",x,y);
  pthread_cond_wait(&cond,&mtx);
  }
pthread_mutex_unlock(&mtx);
fprintf(stderr,"and then %lu > %lu\n",x,y);
return((void *)0);
}
```

```
$ ./prog
..until now 115245 <= 10000000
...........................................
...........................................
.........and then 10000001 > 10000000
...........................................
...........................................
.........$
$
```





## Les variables conditions

```
int main(void)              /* cond.c (suite) */
{
pthread_t th;
void * result;
unsigned long t;
x=0; y=10000000;
pthread_mutex_init(&mtx,(pthread_mutexattr_t *)0);
pthread_cond_init(&cond,(pthread_condattr_t *)0);
if(pthread_create(&th,(pthread_attr_t *)0,task,(void *)0))
  { fprintf(stderr,"Pb create\n"); return(1); }
for(t=0;t<20000000;t++)
  {
  if(!(t%100000)) fputc('.',stderr);
  pthread_mutex_lock(&mtx);
  if((x=t)>y) pthread_cond_broadcast(&cond);
  pthread_mutex_unlock(&mtx);
  }
if(pthread_join(th,&result))    { fprintf(stderr,"Pb join\n"); return(1); }
if(pthread_cond_destroy(&cond)) { fprintf(stderr,"Pb cond destroy\n"); return(1); }
if(pthread_mutex_destroy(&mtx)) { fprintf(stderr,"Pb mtx destroy\n"); return(1); }
return(0);
}
```





## Les données privées

- ▷ **Principe**
  - ◇ Créer une clef unique accessible par tous les threads
  - ◇ Chaque thread associe ses propres données à cette clef
  - ◇ Ces données sont tout à fait indépendantes d'un thread à un autre
  - ◇ Type de la clef : `pthread_key_t`
- ▷ **Création de la clef** (`man 3 pthread_key_create`)
  - ◇ `int pthread_key_create(pthread_key_t * key,`<br>`void (*destrFnct)(void *));`
  - ◇ Initialise la clef désignée par `key`
  - ◇ Si `destrFnct` est non nul → fonction de destruction des données
    - ◦ À la terminaison ou à l'annulation de chaque thread
  - ◇ Retour : `0` si ok, non nul si erreur
  - ◇ Erreur si nombre de clef maxi atteint (`PTHREAD_KEYS_MAX`)





## Les données privées

- ▷ **Destruction de la clef** (`man 3 pthread_key_delete`)
  - ◇ `int pthread_key_delete(pthread_key_t key);`
  - ◇ La clef devient inutilisable (n'appelle pas la fonction de destruction)
  - ◇ Retour : `0` si ok, non nul si erreur
- ▷ **Écriture d'une donnée** (`man 3 pthread_key_setspecific`)
  - ◇ `int pthread_key_setspecific(pthread_key_t key,`<br>`const void * data);`
  - ◇ Associe la donnée `data` à la clef `key` pour le thread courant
  - ◇ Retour : `0` si ok, non nul si erreur
- ▷ **Lecture d'une donnée** (`man 3 pthread_key_getspecific`)
  - ◇ `void * pthread_key_getspecific(pthread_key_t key);`
  - ◇ Retourne la donnée associée à `key` et au thread courant
  - ◇ Retourne un pointeur nul en cas d'erreur

## Les données privées

```
#include <pthread.h>       /* data.c */
#include <stdio.h>

pthread_key_t key;

void destroyFunc(void * data)
{
fprintf(stderr,"destroying [%s]\n",
               (const char *)data);
}

void * task(void * data)
{
int i;
if(pthread_setspecific(key,((int)data)%2 ? "data is odd" : "data is even"))
  { fprintf(stderr,"Pb set specific\n"); return((void *)0); }
for(i=0;i<100*1000*1000;i++);
fprintf(stderr,"stored in thread %d : %s\n",
               (int)pthread_self(), // ugly cast !
               (const char *)pthread_getspecific(key));
return((void *)0);
}
```

```
$ ./prog
stored in thread 1026 : data is odd
destroying [data is odd]
stored in thread 2051 : data is even
destroying [data is even]
stored in main thread : first thing stored
$
```





## Les données privées

```
int main(void)            /* data.c (suite) */
{
pthread_t th1,th2;
void * result;
if(pthread_key_create(&key,destroyFunc))
  { fprintf(stderr,"Pb key create\n"); return(1); }
if(pthread_setspecific(key,"first thing stored"))
  { fprintf(stderr,"Pb set specific\n"); return(1); }
if(pthread_create(&th1,(pthread_attr_t *)0,task,(void *)1))
  { fprintf(stderr,"Pb create\n"); return(1); }
if(pthread_create(&th2,(pthread_attr_t *)0,task,(void *)2))
  { fprintf(stderr,"Pb create\n"); return(1); }
if(pthread_join(th1,&result))
  { fprintf(stderr,"Pb join\n"); return(1); }
if(pthread_join(th2,&result))
  { fprintf(stderr,"Pb join\n"); return(1); }
fprintf(stderr,"stored in main thread : %s\n",
               (const char *)pthread_getspecific(key));
if(pthread_key_delete(key))
  { fprintf(stderr,"Pb key destroy\n"); return(1); }
return(0);
}
```





## Les systèmes de fichiers

### Les systèmes de fichiers

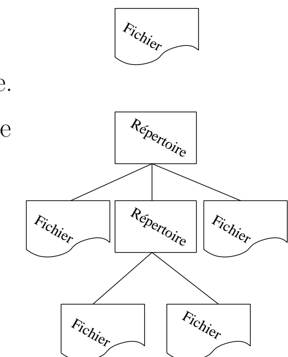
*Définitions*
*Les répertoires*
*Objectifs*
*Méthodes d'accès (séquentiel, direct, indexé)*
*Architecture logicielle de la gestion de fichiers*
*Organisation de l'espace disque*
*Exemples de mise en œuvre : NTFS, UFS, ...*
*Fiabilité et récupération*





## Définitions

- ▷ Un fichier est un "conteneur" d'informations en relation, enregistrées sur une mémoire secondaire.
- ▷ Un système de fichiers organise les fichiers en une structure logique, qui est, la plupart du temps, hiérarchique.
- ▷ Un répertoire est un fichier qui enregistre des informations relatives à d'autres fichiers, et qui définit leur position dans la structure logique.
- ▷ Une partition contient un groupe, normalement étendu, de répertoires.



## Les répertoires

- ▷ Les répertoires sont des fichiers dont le contenu organise les fichiers au sein d'une hiérarchie logique, et leur associe des attributs.
- ▷ Un "fichier répertoire" contient une entrée pour chacun des fichiers qui lui "appartiennent".

**Remarque** : l'organisation logique des fichiers présentée à l'utilisateur ne se reflète pas dans l'enregistrement physique des fichiers sur les disques.





## Les répertoires : contenu

Pour chaque fichier du répertoire, sont enregistrés :

- ▷ le nom (unique dans le répertoire) et le type du fichier,
- ▷ des informations d'adressage (périphérique de stockage, taille, position d'enregistrement sur le périphérique),
- ▷ des information de contrôle d'accès (propriétaire, permissions d'accès)
- ▷ des informations d'usage (estampilles de lecture et de modification, activités en cours sur le fichier, identité des derniers utilisateurs).





## Objectifs d'un système de gestion de fichiers

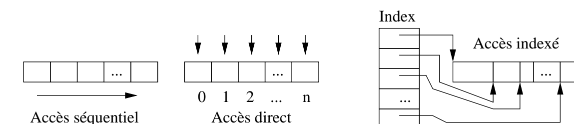
- ▷ permettre le stockage des données, et les opérations utilisateurs sur celles-ci (relecture complète ou partielle, séquentielle ou aléatoire, insertion, suppression, mise à jour des données),
- ▷ optimiser les performances d'accès,
- ▷ permettre une implantation du système de fichiers sur un maximum de supports d'enregistrements différents, tout en essayant de standardiser les accès,
- ▷ gérer les fichiers dans un environnement multi-utilisateurs,
- ▷ sécuriser le système, pour limiter les risques de pertes d'informations.





## Méthodes d'accès au contenu d'un fichier

- ▷ **Accès séquentiel** : les données composant un fichier sont accessibles dans leur ordre logique d'enregistrement (comme sur une bande magnétique)
  → simplicité, modèle très courant d'accès, données hétérogènes.
- ▷ **Accès direct** : les données sont de longueur fixe et désignées par un numéro d'ordre dans le fichier
  → efficacité, contrainte d'organisation.
- ▷ **Accès indexé** : les enregistrements d'index, de taille fixe, donnent la position des données, qui peuvent être de taille variable, dans le fichier. L'accès débute par un accès direct à l'index.



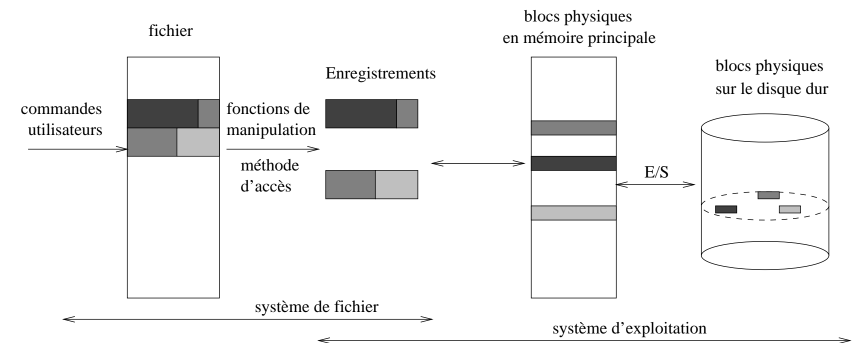
## Architecture logicielle de la gestion de fichiers

- **Entrées/sorties logiques** :
  traitent les données au niveau de la structure logique (utilisateur) des fichiers.
- **Superviseurs d'entrées/sorties** :
  sélectionnent un support d'enregistrement, ordonnancent les accès en mode multi-tâches.
- **Entrées/sorties physiques** :
  échangent des blocs de données entre la mémoire principale et le support d'enregistrement, allouent des blocs d'enregistrement.
- **Pilotes de périphériques** :
  contrôlent le périphérique associé au niveau matériel.





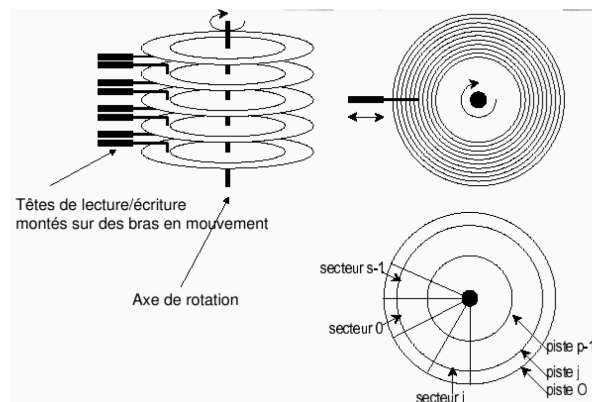
## Architecture logicielle et fonctionnalités de la gestion de fichiers








## Organisation de l'espace disque : le disque physique








## Organisation de l'espace disque

Pour décrire l'utilisation du disque, le système d'exploitation gère 2 tables :

- une table d'allocation qui associe des portions du disque à l'enregistrement des fichiers (souvent appelée FAT *File Allocation Table*),
- une table des portions libres sur le disque.

**Remarques** :

- L'unité d'allocation sur le disque peut être de taille variable, ou de taille fixe (on parle alors de blocs).
- La grande majorité des systèmes utilise la seconde solution.

## Organisation de l'espace disque : allocation contiguë

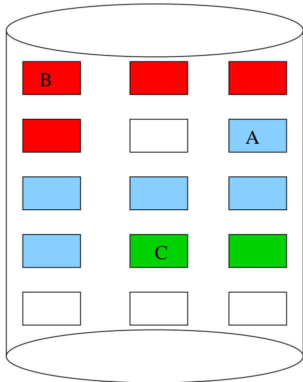



Table d'allocations

| nom | début | longueur |
|---|---|---|
| A | 5 | 5 |
| B | 0 | 4 |
| C | 10 | 2 |

◇ Accès séquentiel rapide

◇ Accès direct à un bloc individuel possible

◇ Problème de fragmentation
→ nécessité de "défragmenter"





## Organisation de l'espace disque : allocation chaînée

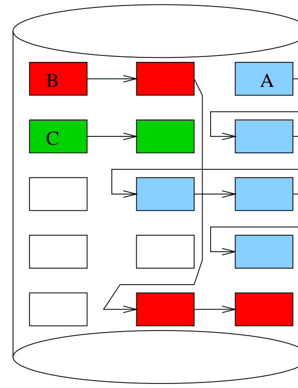



Table d'allocations

| nom | début | longueur |
|---|---|---|
| A | 2 | 5 |
| B | 0 | 4 |
| C | 3 | 2 |

◇ Accès séquentiel lent (faible localité spatiale, nécessité d'interpréter les informations de chaînage)

◇ Accès direct à un bloc individuel impossible

◇ La taille utile des blocs n'est plus une puissance de 2





## Organisation de l'espace disque : allocation indexée (index vers les blocs)

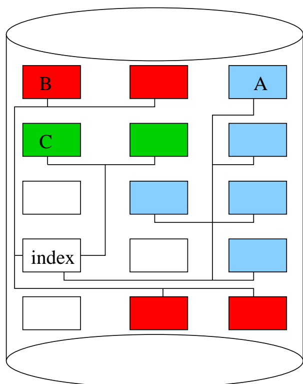



Table d'allocations

| nom | bloc index | bloc |
|---|---|---|
| A | 9 | 2,5,7,8,11 |
| B | 9 | 0,1,13,14 |
| C | 9 | 3,4 |

◇ Le bloc *index* peut contenir les index associés à plusieurs fichiers.

◇ Accès direct à un bloc individuel possible

◇ De tailles réduites pour de petits fichiers, les index peuvent être copiés en mémoire principale.





## Organisation de l'espace disque : allocation indexée (index vers une suite de blocs)

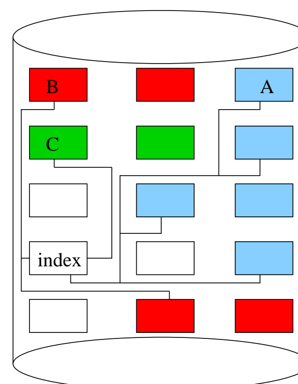



Table d'allocations

| nom | bloc index | bloc, longueur |
|---|---|---|
| A | 9 | (2,1), (5,1), (7,2), (11,1) |
| B | 9 | (0,2), (13,2) |
| C | 9 | (3,2) |

◇ Exploitation simple de la localité spatiale : les blocs adjacents sont mis en évidence par la structure de l'index.

◇ Accès direct à un bloc individuel possible

◇ La fragmentation externe augmente la taille de l'index

## Exemple de mise en œuvre : NTFS (NT File System)

Notions clefs :

- **Cluster** : regroupement de blocs contigus, unité d'allocation minimun

  Taille cluster ↑ ⇒ Localité spatiale ↑ ⇒ Performance ↑ ⇒ Mais fragmentation interne ↑

  → adapter la taille de l'unité d'allocation en fonction de la taille du disque.
- **Volume** : partition (disque logique).
- **Fichier d'initialisation** : programme d'amorçage du système, situé à un emplacement pré-défini sur le disque. Il contient aussi l'emplacement de la table des fichiers maîtres.
- **Table des fichiers maîtres (MFT : *Master File Table*)** : un fichier MFT par volume, situé à un emplacement pré-défini.
  La table contient des **enregistrements de fichiers** décrivant les fichiers.





## Exemple de mise en œuvre : NTFS (2)

Notions clefs (suite) :

- **Enregistrements de fichiers** : listes de couples attribut/valeur
  - Attribut : valeurs numériques conventionnelles caractérisant la valeur associée.
  - Valeur : sa sémantique est donnée par l'attribut.
  - Exemples d'attributs : nom de fichier, données, descripteur de sécurité, permissions, nom de fichiers dans un répertoire, ...

  La taille d'un enregistrement de fichiers est de 1KOctets.

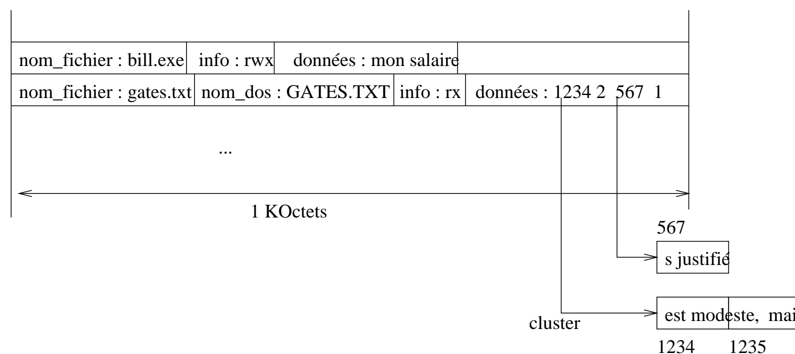
  Un fichier de petite taille peut y être entièrement enregistré
  → accès direct à l'information (fichier dit *résident*).





## Exemple de mise en œuvre : NTFS (3)

Un fichier de taille supérieure est accessible indirectement (fichier dit *non-résident*).




L'allocation indexée (vers suite de blocs) est utilisée pour enregistrer le contenu du fichier sur le disque.





## Exemple de mise en œuvre : NTFS (4) Compression et fichiers creux

- NTFS dispose de possibilités de compression/décompression "à la volée" du contenu des fichiers.

  La compression/décompression est réalisée sur la base d'une division en zones de 16 "clusters" du contenu du fichier ;
  les zones sont traitées indépendamment les unes des autres.

  ⇒ Les zones "utiles" sont décompressées, les autres non.
- NTFS peut détecter les "clusters vides" (c'est–à–dire de longues suites de valeurs égales à 0 dans le fichier), et ne pas les enregistrer réellement sur le disque.

## Exemple de mise en œuvre : UFS (Unix File System)

UFS : version constructeur (SUN) de FFS *Fast File System*.

Organisation physique d'une partition UFS

| | | | | |
|---|---|---|---|---|
| cylindre 0 | boot | superbloc | i–noeuds | blocs de données |
| | ... | | | |
| cylindre i | superbloc | blocs de données | | |
| | ... | | | |

Les 16 premiers secteurs du cylindre 0 contiennent un programme d'amorçage du système (*boot*).

Chaque cylindre contient une copie du "superbloc" :
◇ données sur le système de fichiers lui-même ◇ plusieurs copies par sécurité





## Exemple de mise en œuvre : UFS Les i–nœuds

▷ Un i-nœud (*index node* ou *inode*) est associé à chaque fichier. Il décrit les attributs du fichier et adresse son contenu sur le disque.

▷ Une zone est réservée sur le cylindre 0 du disque pour enregistrer tous les i-nœuds du système. Ils sont désignés par leur numéro d'ordre dans cette zone.

▷ La taille de la zone "i-nœuds" détermine le nombre maximum de fichiers qui peuvent être enregistrés dans une partition.

▷ L'i-nœud numéro 2 désigne le répertoire racine de la hiérarchie de fichiers de la partition.





## Exemple de mise en œuvre : UFS Les i–nœuds (2)

▷ Les informations contenues dans un i-nœud sont de deux types :
- ◇ les attributs du fichier (nature, propriétaire, ...),
- ◇ le contenu du fichier (chaînage des blocs de données, ou référence à un pilote par un nombre majeur ou mineur).

▷ Une entrée dans un répertoire associe un nom à un i-nœud. Un fichier peut donc avoir plusieurs noms (ou liens).

▷ La structure `icommon`, définie dans `/usr/include/sys/fs/ufs_inode.h` représente un i-nœud.

▷ Lorsqu'un fichier est utilisé, son i-nœud est chargé en mémoire. L'i-nœud est alors complété par d'autres champs, qui ne sont pas enregistrés sur le disque : périphérique d'origine, contrôle d'accès au fichier, ...

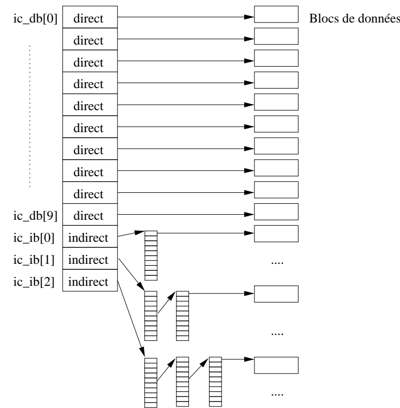




## Exemple de mise en œuvre : UFS Les i–nœuds (3) : structure `icommon`

```
#define   NDADDR   10    /* direct addresses in inode */
#define   NIADDR   3     /* indirect addresses in inode */
struct   icommon {
    o_mode_t ic_smode;   /*  mode and type of file */
    short    ic_nlink;   /*  number of links to file */
    o_uid_t  ic_suid;    /*  owner's user id */
    o_gid_t  ic_sgid;    /*  owner's group id */
    quad  ic_size;       /*  number of bytes in file */
    ...
    struct timeval ic_atime; /* time last accessed */
    struct timeval ic_mtime; /* time last modified */
    struct timeval ic_ctime; /* last time inode changed */
    ...
    daddr_t  ic_db[NDADDR];  /* disk block addresses */
    daddr_t  ic_ib[NIADDR];  /* indirect blocks */
    ... };
```

## Exemple de mise en œuvre : UFS
## Les i–nœuds (3) : Adressage des blocs physiques








## Exemple de mise en œuvre : UFS
## Le superbloc

Le superbloc contient des données sur le système de fichiers lui-même :

▷ taille du système de fichiers,

▷ taille de la zone des i-nœuds, nombre d'i-nœuds libres,

▷ liste des i-nœuds libres,

▷ liste des blocs libres.

**Remarque :** le superbloc est recopié en mémoire principale lorsque un disque est "monté" dans le système de fichier. L'arrêt brutale d'une machine peut provoquer la perte de modifications réalisées sur la copie mémoire du superbloc.





## Exemple de mise en œuvre : autres systèmes de fichiers

▷ ext, ext2, ext3 : systèmes de fichiers Linux.

▷ HPFS, FAT32, VFAT : systèmes de fichiers Microsoft.

▷ NFS (Network File System), DFS (Distributed File System), RFS (Remote File System) : interface vers des systèmes de fichiers distants.

▷ tmpfs : disque "mémoire", géré par des mécanismes de mémoire virtuelle
→ stockage non permanent, mais accès très rapide.

▷ procfs : image dans le système de fichiers de certaines structures de données internes au noyau.
Utilisé principalement par les outils de débogage.





## Fiabilité et récupération

▷ Les supports d'enregistrement de type disque dur ont une durée de vie limitée.

▷ Le système de fichiers est décrit par un ensemble de méta-données : l'accès aux données repose sur la cohérence de ces méta-données.

⇒ une confiance aveugle dans le système de fichiers expose l'utilisateur à de grandes catastrophes... :(

Cependant, les systèmes modernes proposent des procédures de récupération permettant de remédier à certaines défaillances... :)

## Performances contre fiabilité

**Caches disques :** les *caches disques* contiennent une copie des fichiers les plus récemment utilisés.

Ils s'appuient sur les mêmes principes que les caches mémoires des processeurs (principes de localité spatiale et temporelle).
Ils limitent le nombre d'accès aux périphériques
→ performances.

Les modifications du cache disque sont périodiquement reportées sur le disque, mais peuvent être perdues en cas d'arrêt brutal de la machine.

Sous Unix, la commande `sync` permet de forcer une mise à jour du disque par recopie du cache.





## Fiabilité et récupération : les écritures attentives

Une écriture dans le système de fichiers peut nécessiter la mise à jour de plusieurs fichiers (le fichier modifié lui-même, et les fichiers de méta-données).

En cas d'interruption inopinée, le système de fichiers peut se trouver dans un état incohérent.

Les systèmes à écriture attentive cherchent à ordonner les opérations de modifications sur le disque de manière à minimiser l'impact d'une interruption brutale.





## Fiabilité et récupération : transactions vers le disque

Modèle transactionnel : une transaction est une opération considérée comme "atomique", qui ne peut être interrompue.

En cas d'interruption inopinée (panne de courant par exemple), une transaction initiée doit pouvoir être annihilée ou refaite.

Le modèle transactionnel permet théoriquement de conserver un système de fichiers dans un état cohérent, même en cas de transactions concurrentes.





## Fiabilité et récupération : mise en journal des transactions

Toutes les opérations de modification du système de fichiers sont enregistrées dans un journal. Les enregistrements du journal indiquent comment faire ou comment défaire les transactions.

La modification du disque est autorisée lorsque la transaction a été entièrement décrite et enregistrée dans le journal (transaction validée).

En cas de problème, le journal permet soit de refaire les transactions qui ont été perdues, ou de défaire des transactions incomplètes (système de fichiers incohérent).

## Fiabilité et récupération : système NTFS – récupération

NTFS gère un modèle transactionnel, qui garantit les méta-données du système de fichiers
→ la structure logique du système de fichiers est protégée.

Le système n'est pas protégé contre la perte des données utilisateurs.

Séquence d'actions pour modifier un fichier :

1. description de la transaction dans le journal "cache",
2. modification du volume dans le "cache",
3. vidage du journal "cache" sur le disque, puis validation de la transaction,
4. vidage des modifications du cache sur le disque.





## Fiabilité et récupération : système UFS – logiciel `fsck`

Le logiciel `fsck` est chargé de vérifier l'intégrité du système ou des systèmes de fichiers.

Opérations effectuées :

- ▷ vérification de l'accès aux blocs du disque et aux fichiers décrivant le système de fichiers,
- ▷ vérification des i-nœuds (taille, format, allocation de blocs, nombre de liens, . . . ),
- ▷ vérification de la structure logique du système (contenu des fichiers répertoires, liaisons entre les répertoires, . . . )
- ▷ vérification de la liste des blocs libres.
- ▷ diagnostic et état du système (fragmentation, intégrité)





## Fiabilité et récupération : système UFS – Cohérence au niveau des blocs

En parcourant la liste des blocs libres,
on les note dans le tableau `LIBRE`.

En parcourant tous les blocs d'information,
on note les blocs utilisés dans le tableau `UTILISE`.

- ▷ si `LIBRE[i]>1`,
  on supprime une des références au bloc `i` dans `LIBRE`
- ▷ si `LIBRE[i]>0` et `UTILISE[i]>0`,
  on supprime la référence au bloc `i` dans `LIBRE`
- ▷ `UTILISE[i]>1`,
  on duplique le bloc `i` et on l'insère dans un des fichiers qui le référencient





## Fiabilité et récupération : système UFS – Cohérence au niveau des fichiers

En parcourant tous les répertoires,
on note pour chaque i–nœud le nombre d'entrées (`n`) qui pointent sur ce nœud.

- ▷ si `n=0`,
  le fichier est mis dans un répertoire spécial : `lost+found`
- ▷ si `n` correspond au nombre de liens déclarés dans l'i–nœud,
  c'est correct !

# Les moyens d'entrée/sortie (programmeur)

## Les moyens d'entrée/sortie (programmeur)

*Descripteurs de fichiers*
*Flux*
*Parcours des répertoires*





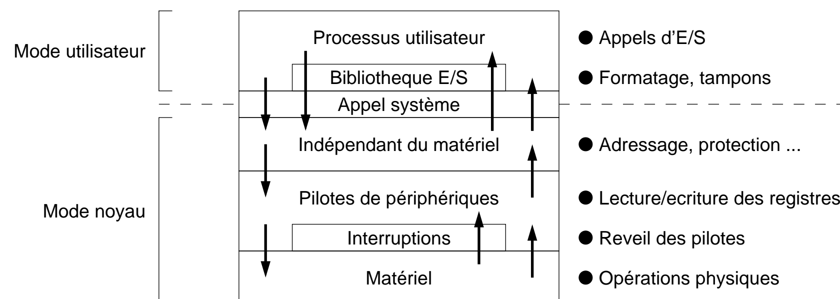
# Principes

- ▷ **De manière générale :** *"tout est fichier"* **!**
  - ◇ Mise en œuvre homogène des entrées/sorties
  - ◇ Se résume à des lectures/écritures dans des flots
  - ◇ Réutilisation des traitements dans divers contextes
  - ◇ Généralisation du terme *"fichier"*
    - ○ Fichier *"réel"* d'un système de fichiers
    - ○ Terminal
    - ○ Tube de communication
    - ○ *Socket*
    - ○ ...
- ▷ **Deux notions principales**
  - ◇ Les descripteurs de fichiers (système)
  - ◇ Les flux de données (espace utilisateur)





# Les couches d'entrées/sorties



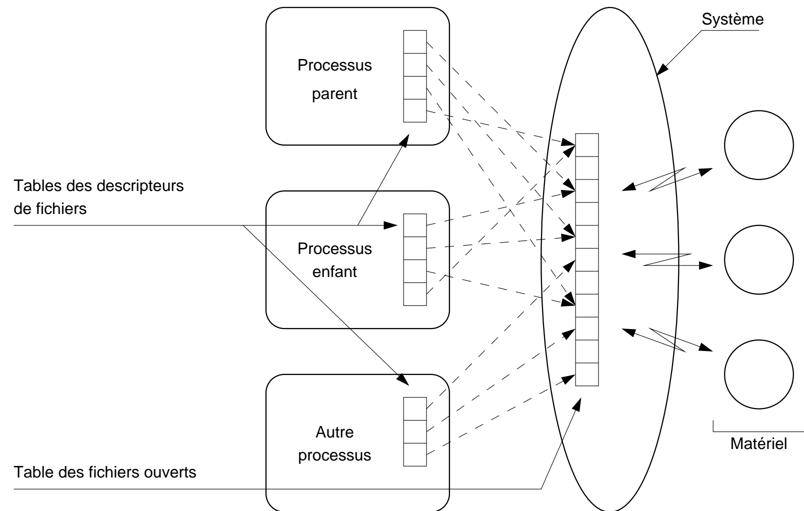





# Descripteurs de fichiers

- ▷ **Identifiant d'un flot** (*File Descriptor*)
  - ◇ Un simple entier (type `int`) désignant un flot ouvert
  - ◇ *API* relativement indépendante du procédé physique
  - ◇ Abstraction plus élevée que les pilotes de périphériques
- ▷ **Rôle, utilisation**
  - ◇ Envoyer/obtenir des blocs de données
  - ◇ Données stockées dans l'espace d'adressage du processus
  - ◇ Données non formatées, binaires ou textuelles
  - ◇ Pas de tampon (le minimum)
  - ◇ À chaque opération → appel système

## Descripteurs de fichiers








## Descripteurs de fichiers

- ▷ **Opérations spécifiques**
  - ◇ Ouverture du flot
  - ◇ Traitements particuliers (positionnement . . . )
- ▷ **Opérations génériques** (*"un simple tuyau"*)
  - ◇ Lecture/écriture
  - ◇ Scrutation
  - ◇ Paramétrage
  - ◇ Fermeture





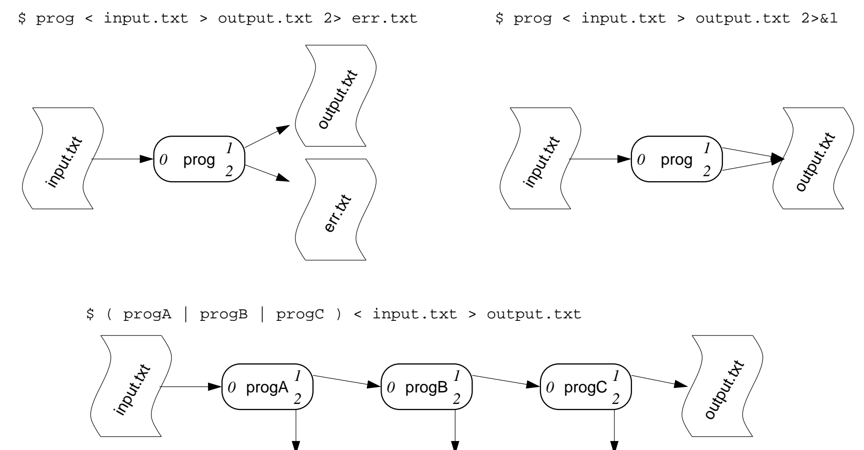
## Flots par défaut

- ▷ **Ouverture implicite de 3 flots pour chaque processus**
  - ◇ Entrée standard → lecture depuis le terminal
    - ∘ Descripteur de fichier `0` (`STDIN_FILENO` dans `unistd.h`)
  - ◇ Sortie standard → écriture vers le terminal
    - ∘ Descripteur de fichier `1` (`STDOUT_FILENO` dans `unistd.h`)
  - ◇ Sortie d'erreurs → écriture vers le terminal
    - ∘ Descripteur de fichier `2` (`STDERR_FILENO` dans `unistd.h`)
- ▷ **Modifications explicites**
  - ◇ Redirection depuis la ligne de commande
  - ◇ Redirection/fermeture par le processus parent
  - ◇ Redirection/fermeture par le processus lui-même





## Flots par défaut



## Ouverture/fermeture d'un flot

▷ **Cas particulier ici pour l'ouverture**
- ◇ Un fichier *"réel"* dans un système de fichiers
- ◇ Plus général qu'il n'y parait (`/dev/ttyS0`, `/dev/audio` ...)

▷ **Ouverture d'un fichier existant**
- ◇ Appel système `open()` (`man 2 open`)

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char * path,int flags,
          ...  /* mode_t mode */);
```

- ◇ `path` : chemin absolu ou relatif du fichier à ouvrir





## Ouverture/fermeture d'un flot

▷ **Ouverture d'un fichier existant**
- ◇ `flags` : type d'ouverture et paramètres (combinaison bit à bit)
  - ○ Ouverture `O_RDONLY`, `O_WRONLY` ou `O_RDWR`
  - ○ `O_CREAT` : création si inexistant
  - ○ `O_EXCL` : échec si `O_CREAT` mais existant
  - ○ `O_APPEND` : ajout en fin de fichier
  - ○ `O_NONBLOCK`, `O_SYNC`, `O_TRUNC`, `O_NOCTTY` ...
- ◇ `mode` : droits du fichier si `O_CREAT`
  - ○ Valeur entière (octale) ou combinaison bit à bit
  - ○ `S_ISUID`, `S_ISGID`, `S_ISVTX`
    `S_IRWXU`, `S_IRUSR`, `S_IWUSR`, `S_IXUSR`
    `S_IRWXG`, `S_IRGRP`, `S_IWGRP`, `S_IXGRP`
    `S_IRWXO`, `S_IROTH`, `S_IWOTH`, `S_IXOTH`





## Ouverture/fermeture d'un flot

▷ **Ouverture d'un fichier existant**
- ◇ Retour de `open()`
  - ○ Un descripteur de fichier si ouverture correcte
  - ○ `-1` si une erreur survient
- ◇ Nombreuses causes d'erreurs (consulter `errno`)
  - ○ Fichier inexistant, droits insuffisants, `flags` incorrects ...

▷ **Création d'un nouveau fichier**
- ◇ Appel système `creat()` (`man 2 creat`)
  - ○ `int creat(const char * path,mode_t mode);`
    ≡ `open(path,O_CREAT|O_WRONLY|O_TRUNC,mode);`





## Ouverture/fermeture d'un flot

▷ **Synchronisation d'un flot**
- ◇ Appel système `fsync()` (`man 2 fsync`)
  - ○ `#include <unistd.h>`
    `int fsync(int fd);`
- ◇ `fd` : descripteur de fichier quelconque
- ◇ Retour : `0` si OK, `-1` si mauvais `fd` ou flot inadapté

▷ **Fermeture d'un flot**
- ◇ Très général (pas uniquement les fichiers)
- ◇ Appel système `close()` (`man 2 close`)
  - ○ `#include <unistd.h>`
    `int close(int fd);`
- ◇ `fd` : descripteur de fichier quelconque
- ◇ Retour : `0` si OK, `-1` si mauvais `fd`

## Lecture/écriture dans un flot

▷ **Lecture depuis un flot**

- Très général (pas uniquement les fichiers)
- Appel système `read()` (`man 2 read`)
  - `#include <unistd.h>`
    `ssize_t read(int fd,void * buf,size_t count);`
- Extraction de `count` octets de `fd` vers `buf` (préallablement alloué !)
- Retour : nombre d'octets extraits, `0` si fin de fichier, ou `-1` si erreur
- Nombreuses causes d'erreurs (consulter `errno`)
  - `EBADF` si mauvais `fd`
  - `EINTR` si interrompu → relance
  - `EAGAIN` si ouverture `O_NONBLOCK` (ou vérouillage) et rien à lire
  - Dépendant de la nature exacte de `fd` ...
- Gestion *"subtile"* du volume extrait !





## Lecture/écriture dans un flot

▷ **Écriture vers un flot**

- Très général (pas uniquement les fichiers)
- Appel système `write()` (`man 2 write`)
  - `#include <unistd.h>`
    `ssize_t write(int fd,const void * buf,size_t count);`
- Envoi des `count` octets de `buf` dans `fd`
- Retour : nombre d'octets envoyés ou `-1` si erreur
- Nombreuses causes d'erreurs (consulter `errno`)
  - `EBADF` si mauvais `fd`
  - `EPIPE` si extémité fermée (*tube*, *socket*)
  - `EINTR` si interrompu → relance
  - `EAGAIN` si ouverture `O_NONBLOCK` (ou vérouillage) → relance
  - Dépendant de la nature exacte de `fd` ...





## Lecture/écriture dans un flot

```
#include <sys/types.h>              /* copie.c */
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <stdio.h>
#define MY_BUFFER_SIZE 1000

/* Extraire count octets maxi   */
/* en evitant les interruptions */

ssize_t myRead(int fd,char * buf,
               size_t count)
{
ssize_t nb;
do
  {
  nb=read(fd,buf,count);
  } while((nb==-1)&&
          ((errno==EINTR)||
           (errno==EAGAIN)));
return(nb);
}
```

```
/* Transmettre count octets exactement */
/* en evitant les interruptions        */

ssize_t myWrite(int fd,const char * buf,
                size_t count)
{
ssize_t nb;
size_t remaining=count;
const char * ptr=buf;
do
  {
  nb=write(fd,ptr,remaining);
  if(nb>0) { remaining-=nb; ptr+=nb; }
  } while(remaining&&(nb==-1)&&
          ((errno==EINTR)||
           (errno==EAGAIN)));
return(nb==-1 ? -1 : (ssize_t)(count-remaining));
}
```





## Lecture/écriture dans un flot

```
int main(int argc,char ** argv)          /* copie.c (suite) */
{
int input, output, stop;
char buffer[MY_BUFFER_SIZE];
if(argc<=2)
  {
  fprintf(stderr,"usage: %s input output\n",argv[0]);
  return(1);
  }
input=open(argv[1],O_RDONLY);
if(input==-1)
  {
  fprintf(stderr,"Can't open input file %s\n",argv[1]);
  return(1);
  }
output=open(argv[2],O_WRONLY|O_CREAT|O_TRUNC,0644);
if(output==-1)
  {
  close(input);
  fprintf(stderr,"Can't open output file %s\n",argv[2]);
  return(1);
  }
```

## Lecture/écriture dans un flot

```
stop=0;                  /* copie.c (suite) */
do
  {
  ssize_t nbR=myRead(input,buffer,MY_BUFFER_SIZE);
  switch(nbR)
    {
    case -1:
      fprintf(stderr,"Error while reading\n"); stop=1; break;
    case 0: /* end of file */
      stop=1; break;
    default:
      {
      ssize_t nbW=myWrite(output,buffer,nbR);
      if(nbW==-1) { fprintf(stderr,"Error while writing\n"); stop=1; }
      }
    }
  } while(!stop);
close(input);
close(output);
return(0);
}
```





## Scrutation des flots

- **Mécanisme d'attente passive**
  - Attendre simultanément des données sur plusieurs flots
  - Éviter une boucle d'attente active en `O_NONBLOCK`
  - Généralement utilisé en lecture mais même principe en écriture
  - Principe
    - Préciser au système les flots à surveiller
    - Bloquer le processus
    - Réveil lorsqu'un des flots surveillés a évolué ou après un délais
  - Deux primitives disponibles
    - Famille *BSD* : appel système `select()` (la plus utilisée)
    - Famille *System V* : appel système `poll()` (quelquefois émulée par `select()`)





## Scrutation des flots

- **L'appel système** `select()` (`man 2 select`)
  - ```
    #include <sys/time.h>
    #include <sys/types.h>
    #include <unistd.h>
    int select(int n,fd_set * rdfds,fd_set * wrfds,
               fd_set * exfds,struct timeval * timeout);
    ```
  - Ensemble de descripteurs `fd_set` et macros associées
    ```
    FD_ZERO(fd_set * set);
    FD_SET(int fd,fd_set * set);
    FD_CLR(int fd,fd_set * set);
    FD_ISSET(int fd,fd_set * set);
    ```
  - `n` : descripteur maxi `+ 1` (dimensionnement de masques internes)
  - `rdfds`, `wrfds`, `exfds` : descripteurs à surveiller





## Scrutation des flots

- **L'appel système** `select()`
  - `timeout` : attente maximale avant retour
    - Infinie si pointeur nul
    - Voir le cours sur la mesure du temps (structure `timeval`)
    - Test et retour immédiat si durée nulle
  - Retour :
    - Nombre de descripteurs dans les conditions attendues
    - `0` si `timeout` écoulé et rien détecté
    - `-1` si erreur
  - Causes d'erreur (consulter `errno`) :
    - Mauvais arguments
    - `EINTR` → relance
    ```

## Scrutation des flots

```
#include <sys/time.h>                  /* select.c */
#include <sys/types.h>
#include <unistd.h>
#include <errno.h>

#include <stdio.h>
                                            /* Surveiller les descripteurs */
void selectExample(int * fdArray,size_t nbFd) /* passes dans le tableau      */
{
size_t i;
struct timeval timeout;
int result;
int fdMax=-1;
fd_set rdFds;    /* Ensemble des descripteurs a lire */
FD_ZERO(&rdFds); /*   initialement vide              */
for(i=0;i<nbFd;i++) /* y inserer les descripteus du tableau */
  {
  FD_SET(fdArray[i],&rdFds);
  if(fdArray[i]>fdMax) fdMax=fdArray[i]; /* rechercher le plus grand */
  }
timeout.tv_sec=10; /* attente maximale de 10 secondes */
timeout.tv_usec=0; /*   et 0 microseconde            */
```





## Scrutation des flots

```
                            /* select.c (suite) */

do  /* bloquer le processus en attendant une possibilite de lecture  */
  {
  result=select(fdMax+1,&rdFds,(fd_set *)0,(fd_set *)0,&timeout);
  } while((result==-1)&&(errno==EINTR)); /* eviter les interruptions */
switch(result)
  {
  case -1:
    fprintf(stderr,"Pb in select() !\n");
    break;
  case 0:
    fprintf(stderr,"Timeout expired in select() !\n");
    break;
  default:
    for(i=0;i<nbFd;i++)
      {
      if(FD_ISSET(fdArray[i],&rdFds))
        fprintf(stderr,"I can read from %d\n",fdArray[i]);
      }
  }
}
```





## Scrutation des flots

- **L'appel système** `poll()` (`man 2 poll`)
  - ```
    #include <sys/poll.h>
    int poll(struct pollfd * fds,unsigned long nfds,
             int timeout);
    ```
  - Champ `fd` de `pollfd` : flot à surveiller
  - Champ `events` de `pollfd` : conditions attendues
    - `POLLIN` : prêt pour lecture
    - `POLLOUT` : prêt pour écriture
    - `POLLPRI` : donnée urgente
  - Champ `revents` de `pollfd` : résultat (conditions survenues)
    - `POLLIN`, `POLLOUT`, `POLLPRI`
    - `POLLERR` : erreur sur descripteur
    - `POLLHUP` : déconnexion de l'extrémité
    - `POLLNVAL` : descripteur invalide





## Scrutation des flots

- **L'appel système** `poll()`
  - `timeout` : attente maximale avant retour (millisecondes)
    - Infinie si négatif
    - Test et retour immédiat si durée nulle
  - Retour :
    - Nombre de descripteurs dans les conditions attendues
    - `0` si `timeout` écoulé et rien détecté
    - `-1` si erreur
    - Erreurs sur les flots ≠ erreur dans l'appel
  - Causes d'erreur (consulter `errno`) :
    - Mauvais arguments
    - `EINTR` → relance
  - Plus simple et plus général que `select()` mais moins utilisé !
    ```

## Scrutation des flots

```
#include <sys/poll.h>      /* poll.c */
#include <alloca.h>
#include <errno.h>

#include <stdio.h>
                                            /* Surveiller les descripteurs */
void pollExample(int * fdArray,size_t nbFd) /* passes dans le tableau      */
{
size_t i;
int result;
struct pollfd * fds=(struct pollfd *)alloca(nbFd*sizeof(struct pollfd));
for(i=0;i<nbFd;i++)
  {
  fds[i].fd=fdArray[i];           /* inserer le flot                  */
  fds[i].events=POLLIN | POLLOUT; /* attendre pour lecture ou ecriture */
  }
do  /* bloquer le processus en attendant une possibilite de lecture/ecriture */
  {
  result=poll(fds,nbFd,10000); /* attente maximale de 10 secondes */
  } while((result==-1)&&(errno==EINTR)); /* eviter les interruptions */
```





## Scrutation des flots

```
switch(result)             /* poll.c (suite) */
  {
  case -1:
    fprintf(stderr,"Pb in poll() !\n");
    break;
  case 0:
    fprintf(stderr,"Timeout expired in poll() !\n");
    break;
  default:
    for(i=0;i<nbFd;i++)
      {
      if(fds[i].revents&POLLIN)
        fprintf(stderr,"I can read from %d\n",fdArray[i]);
      if(fds[i].revents&POLLOUT)
        fprintf(stderr,"I can write to %d\n",fdArray[i]);
      }
  }
}
```





## Paramétrage des flots

- **L'appel système** `fcntl()` (`man 2 fcntl`)
  - `#include <unistd.h>`<br>`#include <fcntl.h>`<br>`int fcntl(int fd,int cmd,...);`
  - `fd` : le flot à paramétrer
  - `cmd` : la commande à appliquer (peut nécessiter des arguments)
  - Influer sur les paramètres d'un flot ouvert<br>`F_DUPFD`, `F_GETFD`, `F_SETFD`, `F_GETFL`, `F_SETFL`
  - Vérouiller des portions de fichier<br>`F_GETLK`, `F_SETLK`, `F_GETLKW`
  - Communication asynchrone<br>`F_GETOWN`, `F_SETOWN`, `F_GETSIG`, `F_SETSIG`





## Positionnement dans un flot

- **L'appel système** `lseek()` (`man 2 lseek`)
  - Déplacer/consulter la position courante dans un flot de données
  - Position : le type `off_t` ($\simeq$ entier long)<br>`#include <sys/types.h>`
  - `#include <unistd.h>`<br>`off_t lseek(int fd,off_t offset,int whence);`
  - Se déplacer de `offset` octets dans le flot `fd` selon `whence` :
    - `SEEK_SET` : depuis le début du flot
    - `SEEK_CUR` : depuis la position courante
    - `SEEK_END` : depuis la fin du flot
  - Retour : nouvelle position(depuis le début) ou `-1` si erreur

## Positionnement dans un flot

- **L'appel système** `lseek()`
  - Causes d'erreur (consulter `errno`) :
    - Mauvais arguments
    - Flot inadapté (*tube*, *socket* ...)
    - Position finale négative
  - Le déplacement ne modifie pas le contenu !
  - On peut dépasser la fin !
    - On peut écrire à cette position
    - `read()` donne des `0` dans l'espace
  - Consulter la position courante : `pos=lseek(fd,0,SEEK_CUR);`
  - Consulter la taille du flot : `size=lseek(fd,0,SEEK_END);`





## Positionnement dans un flot

```
#include <sys/types.h>              /* readwrite.c */
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main(void) /* inverser les octets d'un fichier */
{
int fd=open("file.txt",O_RDWR);           /* ouverture en lecture/ecriture */
off_t i, halfSize=lseek(fd,0,SEEK_END)/2; /* determiner le milieu          */
lseek(fd,0,SEEK_SET);                     /* revenir au debut              */
for(i=0;i<halfSize;i++)
  {
  char c1,c2;
  read(fd,&c1,sizeof(char));                            /* lire c1 (debut)   */
  lseek(fd,-(i+1),SEEK_END); read(fd,&c2,sizeof(char)); /* lire c2 (fin)     */
  lseek(fd,-sizeof(char),SEEK_CUR);                     /* reculer d'1 octet */
  write(fd,&c1,sizeof(char));                           /* ecrire c1 (fin)   */
  lseek(fd,i,SEEK_SET); write(fd,&c2,sizeof(char));     /* ecrire c2 (debut) */
  }
close(fd);
return(0);
}
```





## Duplication de descripteur

- **L'appel système** `dup()` (`man 2 dup`)
  - Attribuer un autre descripteur au même *"fichier"*
  - `#include <unistd.h>`
    `int dup(int oldfd);`
    `int dup2(int oldfd, int newfd);`
  - Retour : nouveau flot (`newfd` pour `dup2()`) ou `-1` si erreur
  - Causes d'erreur (consulter `errno`) :
    - Mauvais arguments
    - Trop de fichiers ouverts (très rare !)
  - `newfd` est fermé avant d'être utilisé
  - Les opérations sur l'un influent sur l'autre !





## Duplication de descripteur

```
#include <sys/types.h>       /* dup.c */
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

void print(const char * txt,int length) /* Fonction prevue pour ecrire */
{                                       /* sur la sortie standard      */
write(STDOUT_FILENO,txt,length);        /* (la console)                */
}

int main(void)
{
int output;
print("BEFORE DUP\n",11); /* ecriture dans la console */
output=open("output.txt",
            O_WRONLY|O_CREAT|O_TRUNC,0644); /* ouvrir un fichier */
if(dup2(output,STDOUT_FILENO)==-1) /* sortie standard --> fichier */
  {
  write(STDERR_FILENO,"Pb in dup()\n",12);
  return(1);
  }
print("AFTER DUP\n",10); /* ecriture dans le fichier ! */
return(0);
}
```

## Les flux

- ▷ *API* **de haut niveau pour les entrées/sorties**
  - ◇ Bibliothèque de fonctions déclarées dans `stdio.h`
  - ◇ Type *opaque* `FILE` manipulé exclusivement par pointeur
  - ◇ Dissimuler les subtilités des appels systèmes (relance sur `EINTR` ...)
  - ◇ Améliorer la portabilité des programmes
    - ∘ Fait partie de la norme *ANSI* du langage *C*
  - ◇ Puissantes fonctionnalités de formatage des données
  - ◇ Inconnu du système (espace utilisateur)
  - ◇ Compatible avec les opérations *"bas-niveau"*





## Les flux

- ▷ *API* **de haut niveau pour les entrées/sorties**
  - ◇ Tampons internes → moins d'appels au système (plus efficace)
    - ∘ Fonction `setvbuf()` (`man 3 setvbuf`)
    - ∘ Tampon complet, vidé quand il est plein
    - ∘ Tampon par ligne, `'\n'` → vidange (pour les terminaux)
  - ◇ Flux prédéfinis :
    - ∘ Variables globales de type `FILE *`
    - ∘ `stdin` : sur `STDIN_FILENO`
    - ∘ `stdout` : sur `STDOUT_FILENO`, tampon par ligne
    - ∘ `stderr` : sur `STDERR_FILENO`, sans tampon)





## Ouverture/fermeture d'un flux

- ▷ **Ouverture sur un fichier**
  - ◇ Fonction `fopen()` (`man 3 fopen`)
    `FILE * fopen(const char * path,const char * mode);`
  - ◇ Ouvrir le fichier `path` selon `mode` :
    - ∘ `"r"` : lecture depuis le début
    - ∘ `"r+"` : lecture/écriture depuis le début
    - ∘ `"w"` : écriture (fichier vidé ou créé)
    - ∘ `"w+"` : lecture/écriture (fichier vidé ou créé)
    - ∘ `"a"` : écriture à partir de la fin (fichier complété ou créé)
    - ∘ `"a+"` : lecture/écriture à partir de la fin (fichier complété ou créé)
    - ∘ `"b"` peut être ajouté pour indiquer un fichier binaire
      (conversion des fins de lignes, indispensable sous `MSDOS`)





## Ouverture/fermeture d'un flux

- ▷ **Ouverture sur un fichier**
  - ◇ Retour : pointeur sur un nouveau `FILE` ou pointeur nul si erreur
  - ◇ Causes d'erreur (consulter `errno`) :
    - ∘ Mauvais mode
    - ∘ Erreurs de `open()` et `fcntl()`
- ▷ **Ouverture sur un descripteur donné**
  - ◇ Fonction `fdopen()` (`man 3 fdopen`)
    `FILE * fdopen(int fd,const char * mode);`
  - ◇ Le `mode` doit être compatible avec les propriétés du flot déjà ouvert
  - ◇ Commence à la position courante du flot
  - ◇ Mêmes retour et erreurs que `fopen()`
  - ◇ Utile pour simplifier la communication sur *tubes*, *sockets* ...

## Ouverture/fermeture d'un flux

▷ **Réouverture sur un fichier**

- ◇ Fonction `freopen()` (`man 3 freopen`)
  `FILE * freopen(const char * path,const char * mode,`
  `FILE * stream);`
- ◇ Comme `fopen()` mais on remplace un flux existant
  - ○ fermeture du flux `stream`
  - ○ Mise à jour selon `path` et `mode`
- ◇ Mêmes retour et erreurs que `fopen()`
- ◇ Le reste de l'application continue à utiliser le flux `stream`
- ◇ Limité à un fichier → `dup2()` sur le descripteur est plus général





## Réouverture d'un flux sur un fichier

```
#include <stdio.h>              /* freopen.c */

void print(const char * txt) /* Fonction prevue pour ecrire */
{                            /* sur la sortie standard      */
fprintf(stdout,"%s",txt);    /* (la console)                */
}

int main(void)
{
print("BEFORE FREOPEN\n"); /* ecriture dans la console */
if(!freopen("output.txt","w",stdout)) /* sortie standard --> fichier */
  {
  fprintf(stderr,"Pb in freopen()\n");
  return(1);
  }
print("AFTER FREOPEN\n"); /* ecriture dans le fichier ! */
return(0);
}
```





## Ouverture/fermeture d'un flux

▷ **Vidange d'un flux**

- ◇ Fonction `fflux()` (`man 3 fflush`)
  `int fflush(FILE * stream);`
- ◇ Force l'écriture du tampon du flux `stream` (tous si pointeur nul)
- ◇ Ne synchronise pas le flot associé !
- ◇ Retour : `0` si ok, constante `EOF` si erreur
- ◇ Erreurs : comme `write()`

▷ **Fermeture d'un flux**

- ◇ Fonction `fclose()` (`man 3 fclose`)
  `int fclose(FILE * stream);`
- ◇ Ferme le descripteur associé et détruit la structure
- ◇ Retour : `0` si ok, constante `EOF` si erreur
- ◇ Erreurs : comme `fflush()` et `close()`





## Opération sur les flux

▷ **Les erreurs**

- ◇ `int ferror(FILE * stream);` (`man 3 ferror`)
  - ○ Résultat non nul si un erreur est survenue sur `stream`
- ◇ `void clearerr(FILE * stream);` (`man 3 clearerr`)
  - ○ Efface l'indicateur d'erreur sur le flux `stream`

▷ **La fin de fichier**

- ◇ `int feof(FILE * stream);` (`man 3 feof`)
  - ○ Résultat non nul si la fin de fichier a été rencontrée sur `stream`
  - ○ Seulement après lecture de `EOF`

▷ **Le descripteur de fichier**

- ◇ `int fileno(FILE * stream);` (`man 3 fileno`)
  - ○ Le descripteur de fichier associé à un flux
  - ○ Utilisable par les appels système (cohérence !)

## Lecture/écriture dans un flux

▷ **Lecture de données**
- ◇ Fonction `fread()` (`man 3 fread`)

```
size_t fread(const void * buf,size_t size,
             size_t nb,FILE * stream);
```

- ◇ Lit depuis `stream` `nb` éléments de taille `size`
- ◇ Les stocke dans `buf` devant être alloué au préalable
- ◇ Retour : Le nombre d'éléments correctement lus

▷ **Écriture de données**
- ◇ Fonction `fwrite()` (`man 3 fwrite`)

```
size_t fwrite(const void * buf,size_t size,
              size_t nb,FILE * stream);
```

- ◇ Écrit dans `stream` le bloc `buf` contenant `nb` éléments de taille `size`
- ◇ Retour : Le nombre d'éléments correctement écrits





## Lecture/écriture dans un flux

▷ **Lecture formatée (texte)**
- ◇ Fonction `fscanf()` (`man 3 fscanf`)

```
int fscanf(FILE * stream,const char * format,...);
```

- ◇ Extraire depuis `stream` selon les indications de `format`
  - ○ Correspondance entre les symboles `%` et les arguments variables
  - ○ Arguments passés par **adresse** pour leur mise à jour
  - ○ Documenté dans de nombreux ouvrages sur le langage *C*
  - ○ `"%d"` (entier), `"%g"` (réel) ...
- ◇ Retour :
  - ○ Le nombre de conversions réussies (`%`)
  - ○ Constante `EOF` si pb de lecture (fin de fichier ou erreur) avant la première conversion





## Lecture/écriture dans un flux

▷ **Lecture d'une chaîne (texte)**
- ◇ Fonction `fgets()` (`man 3 fgets`)

```
char * fgets(char * buf,int size,FILE * stream);
```

- ◇ Extrait depuis `stream` une ligne de texte et la place dans `buf`
- ◇ Lecture jusqu'à `'\n'` ou la fin de fichier (`'\n'` est inscrit)
- ◇ Le caractère `'\0'` est ajouté à la fin de la chaîne
- ◇ Lecture de `size-1` caractères maximum
- ◇ `buf` doit être alloué à `size` octets minimum
- ◇ Retour : `buf` si ok, pointeur nul si erreur ou fin de fichier et rien lu
- ◇ Ne **jamais** utiliser la fonction

```
char * gets(char * buf);
```

  Pas de limite à la saisie → débordement possible





## Lecture/écriture dans un flux

▷ **Lecture d'un caractère (texte ou donnée)**
- ◇ Fonction `fgetc()` (`man 3 fgetc`)

```
int fgetc(FILE * stream);
```

- ◇ Extrait un caractère depuis `stream`
- ◇ Retour : Caractère extrait ou `EOF` en cas d'erreur ou fin de fichier
- ◇ Le résultat est un `int` et non un `char` !
  - ○ Caractères valables [0;255] → `EOF` est codé sur plus d'un octet
  - ○ Convention : un caractère unique est transmis dans un `int`
- ◇ Macro `getc()` (`man 3 getc`)

```
int getc(FILE * stream);
```

- ◇ Exactement comme `fgetc()` mais sous forme de macro
  - ○ Plus efficace
  - ○ Évaluations multiples de l'argument → effets de bord !

## Lecture/écriture dans un flux

- **Réinjection d'un caractère (texte ou donnée)**
  - Fonction `ungetc()` (`man 3 ungetc`)
    `int ungetc(int c,FILE * stream);`
  - Remet le caractère `c` (normalement le dernier extrait) dans `stream`
  - La prochaine lecture fournira `c`
  - Ne fonctionne qu'une fois de suite
  - Retour : Caractère réinjecté ou `EOF` en cas d'erreur
  - Exemple d'utilisation : analyse lexicale
    - Fin d'un lexème → lecture caractère suivant
    - Premier caractère du lexème suivant
    - Dans `123<=i` , `<` marque la fin de `123` et fait partie de `<=`
      → le replacer dans le flot





## Lecture/écriture dans un flux

- **Écriture formatée (texte)**
  - Fonction `fprintf()` (`man 3 fprintf`)
    `int fprintf(FILE * stream,const char * format,...);`
  - Écrit dans `stream` selon les indications de `format`
    - Correspondance entre les symboles `%` et les arguments variables
    - Documenté dans de nombreux ouvrages sur le langage *C*
    - `"%d"` (entier), `"%g"` (réel), `"%s"` (chaîne) ...
  - Retour : Le nombre de caractères écrits
- **Écriture d'une chaîne (texte)**
  - Fonction `fputs()` (`man 3 fputs`)
    `int fputs(const char * str,FILE * stream);`
  - Écrit la chaîne `str` dans `stream` sans `'\0'`
  - Retour : non négatif si ok, `EOF` en cas d'erreur





## Lecture/écriture dans un flux

- **Écriture d'un caractère (texte ou donnée)**
  - Fonction `fputc()` (`man 3 fputc`)
    `int fputc(int c,FILE * stream);`
  - Écrit le caractère `c` dans `stream`
  - Caractère passé dans un `int`
  - Retour : Caractère transmis ou `EOF` en cas d'erreur
  - Macro `putc()` (`man 3 putc`)
    `int putc(int c,FILE * stream);`
  - Exactement comme `fputc()` mais sous forme de macro
    - Plus efficace
    - Évaluations multiples des arguments → effets de bord !





## Configuration du terminal

- **Caractéristiques en amont du flot**
  - Fonctionnement en mode ligne (canonique) ou caractère (brut)
  - Touches ou combinaisons spéciales
  - Paramètres de transmission sur les ports
  - Accessible par la structure `termios` (`man 3 termios`)
    - Fonctions de `termios.h`
  - Accessible par l'appel système `ioctl()` (`man 2 ioctl`)
    - Complexe et peu portable
    - De moins en moins utilisé
  - Accessible par la commande `stty` (`man 1 stty`)
    - Configuration externe au processus

## Configuration du terminal

```
#include <stdio.h>   /* fgetc.c */

int main(void)
{
int c;
do
  {
  c=fgetc(stdin);
  fprintf(stdout,"-->%.8x [%c]\n",c,c);
  } while(c!=EOF);
return(0);
}

$ echo "abcd" | ./prog
-->00000061 [a]
-->00000062 [b]
-->00000063 [c]
-->00000064 [d]
-->0000000a [
]
-->ffffffff []
```

```
$ ./prog
a
-->00000061 [a]
-->0000000a [
]
bcd
-->00000062 [b]
-->00000063 [c]
-->00000064 [d]
-->0000000a [
]
^D-->ffffffff []
$ stty -icanon
$ ./prog
a-->00000061 [a]
b-->00000062 [b]
c-->00000063 [c]
d-->00000064 [d]
```





## Parcours des répertoires

- ▷ **Parcours séquentiel (principe)**
  - ◇ Ouverture du répertoire par `opendir()` → structure `DIR`
  - ◇ Lecture des entrées sucessives par `readdir()` → structure `dirent`
  - ◇ Lecture du nom de l'entrée par le champ `d_name`
  - ◇ Retour éventuel à la première entrée par `rewinddir()` (seule fonctionnalité de déplacement *POSIX*)
  - ◇ Fermeture du répertoire par `closedir()`





## Parcours des répertoires

- ▷ **La fonction** `opendir()` (`man 3 opendir`)
  - ◇ 
    ```
    #include <sys/types.h>
    #include <dirent.h>
    DIR * opendir(const char * path);
    ```
  - ◇ Crée la structure `DIR` associée au répertoire désigné par `path`
  - ◇ Retour : adresse de cette structure ou pointeur nul si erreur
  - ◇ Causes d'erreur (consulter `errno`) :
    - ○ Répertoire inexistant
    - ○ Droits insuffisants
    - ○ ...





## Parcours des répertoires

- ▷ **La fonction** `readdir()` (`man 3 readdir`)
  - ◇ 
    ```
    #include <sys/types.h>
    #include <dirent.h>
    struct dirent * readdir(DIR * dir);
    ```
  - ◇ Initialise dans `dir` la structure `dirent` décrivant l'entrée suivante
    - ○ Seul champ *POSIX* : `char d_name[]`
    - ○ Ne pas libérer cette structure
    - ○ Contenu écrasé par le prochain `readdir()`
  - ◇ `.` et `..` toujours présents mais pas forcément indiqués
  - ◇ Retour :
    - ○ Adresse de cette structure si ok
    - ○ Pointeur nul si erreur ou dernière entrée dépassée
  - ◇ Causes d'erreur (consulter `errno`) : `dir` incorrect

## Parcours des répertoires

▷ **La fonction** `rewinddir()` (`man 3 rewinddir`)

◇
```
#include <sys/types.h>
#include <dirent.h>
void rewinddir(DIR * dir);
```
◇ Repositionne la lecture du répertoire au début

◇ Les fonctions `seekdir()` et `telldir()` ne sont pas *POSIX*

▷ **La fonction** `closedir()` (`man 3 closedir`)

◇
```
#include <sys/types.h>
#include <dirent.h>
int closedir(DIR * dir);
```
◇ Ferme le répertoire et libère la structure `dir`

◇ Retour : `0` si ok, `-1` si erreur (`dir` incorrect)





## Parcours des répertoires

▷ **Parcours plus élaborés**

◇ La fonction `scandir()` (`man 3 scandir`)
- Rechercher les entrées d'un répertoire
- Un pointeur de fonction pour inclure/exclure une entrée
- Un pointeur de fonction pour trier les entrées
- Fonction non *POSIX* (*BSD*)

◇ La fonction `ftw()` (`man 3 ftw`)
- Parcours récursif d'une arborescence
- Un pointeur de fonction à invoquer sur chaque entrée
- Une profondeur limitée
- Fonction non *POSIX* (*System V*)

◇ La fonction `glob()` (`man 3 glob`)
- Utilisation de *"Jokers"* (`*.c` ...)





## Parcours des répertoires

▷ **Déterminer le répertoire courant**

◇ Point de départ des chemins relatifs

◇ La fonction `getcwd()` (`man 3 getcwd`)
```
#include <unistd.h>
char * getcwd(char * buf,size_t size);
```
◇ Stocker dans `buf` le chemin désignant le répertoire courant

◇ Écriture de `size` octets maxi (y compris `'\0'`)

◇ `buf` doit être préalablement alloué à `size` octets minimum

◇ Retour : `buf` si ok, pointeur nul si erreur

◇ Causes d'erreur (consulter `errno`) :
- `ERANGE` → `size` insuffisant
- Droits d'accès ...





## Parcours des répertoires

▷ **Changer le répertoire courant**

◇ Les fonctions `chdir()` et `fchdir()` (`man 2 chdir / fchdir`)
```
#include <unistd.h>
int chdir(const char * path);
int fchdir(int fd);
```
◇ Le point de départ des chemins relatifs devient `path` ou `fd`

◇ `path` ou `fd` doivent désigner un répertoire

◇ Retour : `0` si ok, `-1` si erreur

◇ Causes d'erreur (consulter `errno`) :
- Mauvais arguments, droits d'accès ...

## Attributs des fichiers

▷ **L'appel système** `lstat()` (`man 2 lstat`)

◇ 
```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int lstat(const char * path,struct stat * infos);
```
◇ Écrit dans la structure `infos` selon les attributs du fichier `path`

◇ Informations de diverses nature :
Type de fichier, droits d'accès, taille, propriétaire, groupe, nombre de références, dates d'accès, de modification des attributs et du contenu . . .

◇ Retour : `0` si ok, `-1` si erreur

◇ Causes d'erreur (consulter `errno`) :

- Fichier inexistant, droits insuffisants . . .





## Attributs des fichiers

▷ **Le champ** `st_mode` **de la structure** `stat`

◇ De type `mode_t` (voir appel système `open()`)

◇ Lecture des droits : et bit à bit avec les constantes de `mode_t`

◇ Décodage du type de fichier à l'aide de macros :

- `S_ISBLK(infos->st_mode)` : périphérique en mode bloc
- `S_ISCHR(infos->st_mode)` : périphérique en mode caractère
- `S_ISDIR(infos->st_mode)` : répertoire
- `S_ISFIFO(infos->st_mode)` : *tube* de communication
- `S_ISLNK(infos->st_mode)` : lien symbolique
- `S_ISREG(infos->st_mode)` : fichier régulier
- `S_ISSOCK(infos->st_mode)` : *socket*





## Attributs des fichiers

```
#include <sys/types.h>       /* dir.c */
#include <sys/stat.h>
#include <dirent.h>
#include <unistd.h>

#include <stdio.h>
#include <string.h>

int main(int argc,char ** argv)
{
const char * path=(argc>1 ? argv[1] : ".");
DIR * dir=opendir(path);
struct dirent * entry;
if(!dir)
  {
  fprintf(stderr,"Can't open directory %s !\n",path);
  return(1);
  }
do
  {
  entry=readdir(dir);
```





## Attributs des fichiers

```
  if(entry)                     /* dir.c (suite) */
    {
    struct stat infos;
    char buffer[0x100];
    strcpy(buffer,path); strcat(buffer,"/"); strcat(buffer,entry->d_name);
    if(lstat(buffer,&infos)!=-1)
      {
      if(S_ISBLK(infos.st_mode))      strcat(buffer,": BLOCK DEVICE");
      else if(S_ISCHR(infos.st_mode))  strcat(buffer,": CHARACTER DEVICE");
      else if(S_ISDIR(infos.st_mode))  strcat(buffer,": DIRECTORY");
      else if(S_ISFIFO(infos.st_mode)) strcat(buffer,": FIFO");
      else if(S_ISLNK(infos.st_mode))  strcat(buffer,": SYMBOLIC LINK");
      else if(S_ISREG(infos.st_mode))  strcat(buffer,": REGULAR FILE");
      else if(S_ISSOCK(infos.st_mode)) strcat(buffer,": SOCKET");
      else strcat(buffer,": ???");
      }
    else strcat(buffer,": ???");
    fprintf(stderr,"%s\n",buffer);
    }
  } while(entry);
closedir(dir);
return(0);
}
```

## Attributs des fichiers

- **L'appel système** `stat()` (`man 2 stat`)
  - Exactement comme `lstat()` mais suit les liens symboliques
  - `S_ISLNK(infos->st_mode)` est inutile
- **L'appel système** `fstat()` (`man 2 fstat`)
  - Exactement comme `stat()` mais sur un descripteur de fichier
  - `int fstat(int fd,struct stat * infos);`
  - On peut ouvrir un répertoire en lecture avec `open()`





## Attributs des fichiers

```
   /* stat.c */
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>
int main(void)
{
struct stat infos;
if(fstat(STDIN_FILENO,&infos)!=-1)
  {
  if(S_ISBLK(infos.st_mode))       fprintf(stderr,"BLOCK DEVICE\n");
  else if(S_ISCHR(infos.st_mode))  fprintf(stderr,"CHARACTER DEVICE\n");
  else if(S_ISDIR(infos.st_mode))  fprintf(stderr,"DIRECTORY\n");
  else if(S_ISFIFO(infos.st_mode)) fprintf(stderr,"FIFO\n");
  else if(S_ISLNK(infos.st_mode))  fprintf(stderr,"SYMBOLIC LINK\n");
  else if(S_ISREG(infos.st_mode))  fprintf(stderr,"REGULAR FILE\n");
  else if(S_ISSOCK(infos.st_mode)) fprintf(stderr,"SOCKET\n");
  else fprintf(stderr,"???\n");
  }
else fprintf(stderr,"???\n");
return(0);
}
```

```
$ ./prog
CHARACTER DEVICE
$ ./prog < /dev/console
CHARACTER DEVICE
$ ./prog < /dev/hda
BLOCK DEVICE
```

```
$ ./prog < file.txt
REGULAR FILE
$ ./prog < /tmp
DIRECTORY
$ ./prog < /tmp/xmms_rodin.0
SOCKET
$ ls | ./prog
FIFO
```





## Attributs des fichiers

- **Modification des droits d'accès** (`man 2 chmod`)
  - ```
    #include <sys/types.h>
    #include <sys/stat.h>
    int chmod(const char * path,mode_t mode);
    int fchmod(int fd,mode_t mode);
    ```
  - Attribuer les droits `mode` au fichier
    - Désigné par le chemin `path`
    - Associé au descripteur `fd`
  - Voir le paramètre `mode` de l'appel système `open()`
  - Retour : `0` si ok, `-1` si erreur (consulter `errno`)





## Attributs des fichiers

- **Modification de la propriété** (`man 2 chown`)
  - ```
    #include <sys/types.h>
    #include <unistd.h>
    int chown(const char * path,uid_t user,gid_t group);
    int fchown(int fd,uid_t user,gid_t group);
    int lchown(const char * path,uid_t user,gid_t group);
    ```
  - Modifier le propriétaire et le groupe du fichier
    - Désigné par le chemin `path`
    - Associé au descripteur `fd`
  - Le lien symbolique lui-même avec `lchown()`
  - Retour : `0` si ok, `-1` si erreur (consulter `errno`)

## Attributs des fichiers

▷ **Lecture et modification de la taille** (`man 2 ftruncate`)
- ◇ Champ `st_size` de la structure `stat`
  - ○ Voir les appels `stat()`, `fstat()`, `lstat()`
  - ○ Type `off_t`
  - ○ Taille en octets d'un fichier, lien symbolique ou répertoire
- ◇
```
#include <unistd.h>
int ftruncate(int fd,off_t length);
```
- ◇ Réduire à `length` la taille du fichier désigné par `fd`
- ◇ Évite la recopie et le renommage
- ◇ Retour : `0` si ok, `-1` si erreur (consulter `errno`)





## Références sur les fichiers

▷ **Ajout d'un lien physique** (`man 2 link`)
- ◇
```
#include <unistd.h>
int link(const char * oldPath,
         const char * newPath);
```
- ◇ Le fichier `oldPath` est également désigné par `newPath`
  - ○ `newPath` de doit pas déjà exister
  - ○ Le champ `st_nlink` de la structure `stat` compte les références
  - ○ La destruction de l'un d'eux n'entraine pas la destruction du fichier
  - ○ Rester dans le même système de fichiers
- ◇ Retour : `0` si ok, `-1` si erreur (consulter `errno`)





## Références sur les fichiers

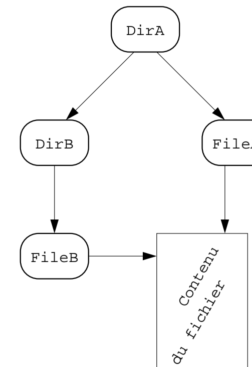
▷ **Ajout d'un lien symbolique** (`man 2 symlink`)
- ◇
```
#include <unistd.h>
int symlink(const char * oldPath,
            const char * newPath);
```
- ◇ Le fichier `oldPath` est également désigné par `newPath`
  - ○ `newPath` de doit pas déjà exister
  - ○ Le fichier n'existe que sous le nom `oldPath`
  - ○ `newPath` ne contient que **le nom** `oldPath`
- ◇ Retour : `0` si ok, `-1` si erreur (consulter `errno`)



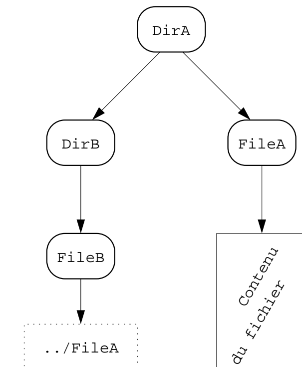


## Références sur les fichiers

`link("DirA/FileA","DirA/DirB/FileB");`




`symlink("DirA/FileA","DirA/DirB/FileB");`



## Références sur les fichiers

▷ **Création d'un répertoire** (`man 2 mkdir`)

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

◇ Crée le répertoire `path` avec les droits `mode`

∘ Il ne doit pas exister

∘ Voir l'argument `mode` de `open()`

◇ Retour : `0` si ok, `-1` si erreur (consulter `errno`)

▷ **Suppression d'un répertoire** (`man 2 rmdir`)

◇ 
```
#include <unistd.h>
int rmdir(const char * path);
```

◇ Supprime le répertoire `path` (doit être vide)

◇ Retour : `0` si ok, `-1` si erreur (consulter `errno`)





## Références sur les fichiers

▷ **Suppression d'une référence** (`man 2 unlink`)

◇ 
```
#include <unistd.h>
int unlink(const char * path);
```

◇ Supprime la référence sur le fichier `path`

◇ Le contenu du fichier peut toujours exister après

∘ Autres liens dans le système de fichier

∘ `open()` référence et `close()` déréférence

∘ `0` références → destruction

◇ Retour : `0` si ok, `-1` si erreur (consulter `errno`)

▷ **La fonction** `remove()` (`man 3 remove`)

◇ 
```
#include <stdio.h>
int remove(const char * path);
```

◇ Appelle `unlink()` ou `rmdir()` selon la nature de `path`





## Références sur les fichiers

▷ **Déplacement/renommage** (`man 2 rename`)

◇ 
```
#include <stdio.h>
int rename(const char * oldPath,
           const char * newPath);
```

◇ Remplace la référence `oldPath` par `newPath`

∘ Écrase la référence `newPath` si elle existe

∘ Si `newPath` est un répertoire, il doit être vide

∘ Rester dans le même système de fichiers

◇ Retour : `0` si ok, `-1` si erreur (consulter `errno`)





## Projection d'un fichier en mémoire

▷ **Principe**

◇ Accéder au contenu d'un fichier comme à un simple bloc mémoire

◇ Pas d'allocation préalable

∘ Le système charge/décharge les pages de manière transparente

∘ Volume de données > mémoire du système

◇ Bien plus rapide qu'une lecture/écriture classique

◇ Moyen de communication

∘ Synchronisation avec les projections d'autres processus

∘ Synchronisation avec le fichier physique

◇ Projection partagée avec les processus fils

◇ Fichier physique référencé tout au long de la projection

## Projection d'un fichier en mémoire

▷ **L'appel système** `mmap()` (`man 2 mmap`)

◇
```
#include <unistd.h>
#include <sys/mman.h>
void * mmap(void * start,size_t length,int prot,
            int flags,int fd,off_t offset);
```
◇ Projette en mémoire le contenu du fichier désigné par `fd`

◇ `start` : adresse désirée pour la projection
(limite de page, géneralement pointeur nul → choix du système)

◇ `offset` : début du fichier à ignorer

◇ `length` : taille de la quantité à projeter

◇ `prot` : protection de la zone mémoire (ou bit à bit)

- `PROT_NONE`, `PROT_READ`, `PROT_WRITE`, `PROT_EXEC`
- Mode d'ouverture de `fd` !
- Dépendant du système





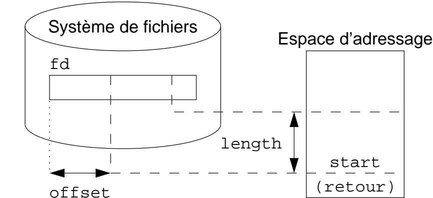
## Projection d'un fichier en mémoire

▷ **L'appel système** `mmap()`

◇ `flags` : options (ou binaire)

- `MAP_FIXED` : échec si `start` non respecté
- `MAP_SHARED` : Synchronisation entre les processus et le fichier
- `MAP_PRIVATE` : copie privée de la projection (taille limitée)
- `MAP_SHARED` ou `MAP_PRIVATE` obligatoire

◇ Retour : Adresse de la projection ou `MAP_FAILED` si erreur
(consulter `errno` : `fd` incompatible, mauvais arguments . . . )








## Projection d'un fichier en mémoire

▷ **L'appel système** `munmap()` (`man 2 munmap`)

◇
```
#include <unistd.h>
#include <sys/mman.h>
int munmap(void * start,size_t length);
```
◇ La zone mémoire [`start`;`start+length`[ n'est plus accessible

- Zone totale de projection fichier

◇ La synchronisation avec le fichier est forcée (si `MAP_SHARED`)

◇ Le descripteur de fichier peut être fermé avant cet appel

◇ Retour : `0` si ok, `-1` si erreur (mauvais arguments)





## Projection d'un fichier en mémoire

▷ **L'appel système** `msync()` (`man 2 msync`)

◇
```
#include <unistd.h>
#include <sys/mman.h>
int msync(void * start,size_t length,int flags);
```
◇ Mettre à jour dans le fichier la zone mémoire [`start`;`start+length`[

- Zone partielle → plus rapide
- Utile pour les projections `MAP_SHARED`

◇ `FLAGS` : type de synchronisation (ou bit à bit)

- `MS_ASYNC` : demande de mise à jour dès que possible
- `MS_SYNC` : mise à jour immédiate
- `MS_INVALIDATE` : provoquer le rechargement des autres processus
- `MS_ASYNC` ou `MS_SYNC` obligatoire

◇ Retour : `0` si ok, `-1` si erreur (mauvais arguments)

# Projection d'un fichier en mémoire

```
#include <sys/types.h>
#include <sys/stat.h>                  /* mmap.c */
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>

int main(void)
{
int fd=open("file.txt",O_RDWR);
struct stat infos;
if(!fstat(fd,&infos))
  {
  off_t sz=infos.st_size, i;
  char * p=(char *)mmap((void *)0,sz,PROT_READ|PROT_WRITE,MAP_SHARED,fd,0);
  close(fd);
  if(p!=(char *)MAP_FAILED)
    {
     for(i=0;i<sz/2;i++) { char c=p[i]; p[i]=p[sz-i-1]; p[sz-i-1]=c; }
     munmap(p,sz);
    }
  }
return(0);
}
```







# Projection d'un fichier en mémoire



```
$ du -m file.txt       # 256 Mo de memoire, swap desactive
257     file.txt
$
$ head -2 file.txt
00-0000 01-0000 02-0000 03-0000 04-0000 05-0000 06-0000 07-0000 08-0000 09-0000
0A-0000 0B-0000 0C-0000 0D-0000 0E-0000 0F-0000 10-0000 11-0000 12-0000 13-0000
$
$ tail -2 file.txt
F4-0100 F5-0100 F6-0100 F7-0100 F8-0100 F9-0100 FA-0100 FB-0100 FC-0100 FD-0100
FE-0100 FF-0100
$
$ ./prog
$
$ head -3 file.txt

0010-FF 0010-EF
0010-DF 0010-CF 0010-BF 0010-AF 0010-9F 0010-8F 0010-7F 0010-6F 0010-5F 0010-4F
$
$ tail -2 file.txt
0000-31 0000-21 0000-11 0000-01 0000-F0 0000-E0 0000-D0 0000-C0 0000-B0 0000-A0
0000-90 0000-80 0000-70 0000-60 0000-50 0000-40 0000-30 0000-20 0000-10 0000-00$
$
```