PRINCIPES DES SYSTEMES D'EXPLOITATION

Licence d'informatique

Table des matières

Chapitre 1

Intro. aux systèmes d'exploitation

1.1 Notion de système d'exploitation

1.1.1 Vue externe (vue qu'un utilisateur a du système)

- machine virtuelle (les services que peut utiliser l'utilisateur sont rajoutés)
- on cache les entrées/sorties bas niveau
- plusieurs utilisateurs **au même instant**
- les différentes couches

4	couches des applications (DOS, LATEX ...)
3	outils de développement (permet d'éditer des sources de programmes, de compiler)
2	système d'exploitation
1	machine physique (ensemble des circuits, clavier, écran, souris, HP, micro ...)

- Les couches 1 et 2 représentent la machine virtuelle du point de vue de l'utilisateur

1.1.2 Vue interne

- Elle s'intéresse aux couches dans le sens inverse (vue ascendante, de 1 → 4)
- elle gère les ressources (travail principal du système d'exploitation)

  CPU : *Central Processing Unit* ou *Unité centrale de traitement* : c'est le processeur

  UAL : *Unité Arithmétique et Logique*

  Mémoire Centrale

  Entrées/Sorties
- problèmes de protection

1.1.3 Conclusion

- La vue externe répond à la question : de quels services peut on disposer ?

- La vue interne répond à la question : comment fonctionnent ces services ?

1.2 Historique

L'histoire des systèmes d'exploitation est liée à l'évolution de la technologie.

1.2.1 Première génération (pas de système d'exploitation)

- Génération des tubes à vide
- période 1945-1955
- ENIAC en 1946 (5000 additions par seconde)

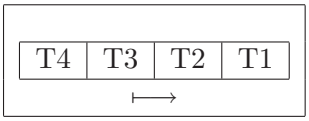
1.2.2 Deuxième génération : transistors et traitement par lots

- période 1955-1965

  transistors : réduction des surfaces et des énergies, plus grande fiabilité et augmentation de la puissance des machines ⇒ augmentation du niveau de raisonnement

- FORTRAN (1955 *FORmula TRANslator*) : traducteur de formules

  traitement par lot : lécart de vitesse entre le temps de calcul et les Entrées/Sorties augmente.



  En mémoire centrale on a les 4 travaux

  explication : Dès que T1 fait une opération d'E/S, T2 est passé à l'Unité Centrale en donnant les cartes/bandes à d'autres ordinateurs. Quand les 4 travaux sont terminés, on les retire de la mémoire et on passe aux autres trains de travaux ⇒ il faut organiser les trains de travaux par des moniteurs d'enchaînement

  avantages : pour des calculs scientifiques très gros, ou de l'informatique de gestion

- COBOL

1.2.3 Troisième génération : circuits intégrés et multiprogrammation

- Période 1965-1980

  circuit intégré : regroupe des transistors dans une même puce ⟹ diminution des volumes, augmentation de la fiabilité

  microprocesseur

  multiprogrammation : plusieurs activités en mémoire centrale (à un même instant, plusieurs tâches s'exécutent)

  système du temps partagé : à chaque tâche est réservé un petit temps de travail (quantum), cela se fait par roulement

- exemple : MULTICS (1965-1969)

1.2.4 Quatrième génération : micro-ordinateurs en réseaux

- VLSI (*Very Large Scale Integration*) circuit très complexe et place très réduite

  micro-ordinateur : marché grand public de l'informatique

  système d'exploitation : succès de MS-DOS, puis de WINDOWS

  développement des réseaux : tous les ordinateurs sont connectés ensembles (LAN : Local Area Network, de Netware)

  système distribué : le système d'exploitation est réparti sur plusieurs machines physiques (AMOEBIA, MACH, CHORUS)

1.2.5 Historique UNIX

DELL, BSD, ATT

1.3 Principales fonctions d'un système d'exploitation

1.3.1 Gestion de la mémoire

- attribuer de la mémoire à ceux qui en demandent
- protéger la mémoire qui a été attribuée

1.3.2 Gestion des activités (= processus)

- un processus est une exécution d'un ou plusieurs programme(s) exécutable(s)
- contrôler l'attribution du CPU et gestion des évènements externes (ex. : on appuie sur une touche)
- gestion des processus utilisateurs (ex. : création/destruction)
- partage de code ou de données entre processus
- protection entre les processus
- communication

1.3.3 Gestion des Entrées/Sorties

disques, bandes

1.3.4 Gestion des fichiers

rôle principal du système d'exploitation

1.3.5 Environnement de développement (entre les couches 2 et 3)

gestion des erreurs et langage de commande

1.3.6 Gestion des utilisateurs

- création ou suppression des utilisateurs
- modification des droits d'accès
- mécanisme de gestion des comptes (commerciaux)
- statistiques (vérifier le comportement des utilisateurs)

1.3.7 Gestion du système lui-même

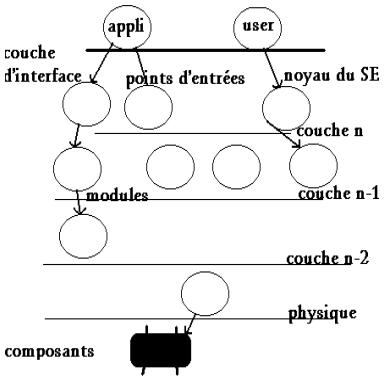
- maintenance (contrôle des équipements)
- sauvegarde + procédure associée
- optimisation de la réponse du système

1.4 Structures d'un système d'exploitation

Comment sont implémentées les fonctions du système d'exploitation ?

1.4.1 Systèmes d'exploitation monolytiques

MSDOS, UNIX : peu de structures.
Deux modes de fonctionnement : user ou superuser(ou mode noyau).



1.4.2 Systèmes d'exploitation hiérarchiques

exemples : MULTIX, VAX/VMS, THE(1968)

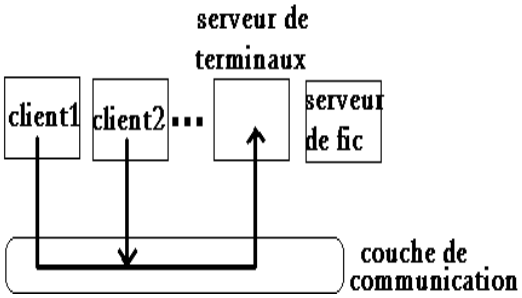
1.4.3 Systèmes d'exploitation à machine virtuelle

exemple : VM/CMS(IBM-1970)
Idée : on a plusieurs machines virtuelles.

CMS	CMS	CMS
	VM3710	
	materiel	

1.4.4 Systèmes d'exploitation clients/serveurs et à micro-noyau

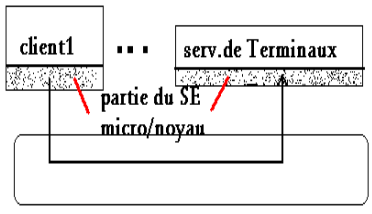
clients/serveurs



L'isolation est encore plus importante que dans le SE hiérarchique.

micro-noyau

Toutes les fonctions fondamentales vont être isolées.



Chapitre 2

Machines, processeurs et exécution

2.1 Machines et processeurs

2.1.1 Architecture générale

Une machine (dite de Von Neuman) est constituée de :
- mémoire centrale
- processeur
- système d'Entrées/Sorties
- liaisons entre les 3

Machine de Heichen : Programme en ROM non modifiable et non changeable (*machine à laver*)

2.1.2 Processeurs

Principe : 3 entités principales dans un processeurs :
- Contrôle
- U.A.L
- Registres

Organisation : CISC : *Complexe Instruction Set.* Travaille dans l'unité centrale (486, Pentium, 1 pération codé sur 4 octets) avec beaucoup d'appels en mémoire ⇒ le processeur a beaucoup de décodage à effectuer.

RISC : *Reduced Instruction Set.* Au départ les U.A.L contenaîent 1 (A) ou 2 (A, B) registre(s). Par la suite, en augmentant le nombre de registres, on a pu faire des opérations plus complexes (multiplication, ...). Il peut être intéressant d'avoir un jeu d'instruction réduit qui travaille en registres (avec plus de registres) ⇒ chemin réalisé par les données très court, mais les registres sont couteux et consommateurs d'énergie.

exemples d'instruction : traduire C ⟵ A+B.

si systèacccmmllttcumulateur : LOAD Acc,A ;*(charger dans Acc la valeur de A)*
ADD B ;*(addition de B)*
STORE C*(le mettre dans C).*

si registres : LOAD A1,A ;*(charger l'adresse de A dans le registre d'adresse A1)*
LOAD A2,B ;*(charger l'adresse de B dans le registre d'adresse A2)*
LOAD A3,C ;*(charger l'adresse de C dans le registre d'adresse A3)*
LOAD R1,A1 ;*(charger le regitre de calcul 1 avec le contenu de l'adresse de A1)*
ADD R1,A2 ;
STORE R1,A3.*(stocker le contenu de R1 dans l'adresse de A3)*

2.1.3 Etapes d'exécution d'une instruction :

- **chargement de l'instruction** (qui est dans la mémoire programmée). Elle se trouve dans le registre d'instruction.
- **décodage** (que doit on faire ?)
- **calcul de l'adresse des opérandes**
- **chargement des registres de l'UAL** (uniquement si cela est nécéssaire)
- **calcul dans l'UAL**
- **rangement du résultat**
- *(passer à l'instruction suivante)*

2.1.4 Contexte d'un processus

- Contexte processeur = mot d'état processeur

  compteur programme

  drapeaux

  information sur instruction en cours (pour les différents modes)
- Contexte UCT (=CPU)

  mot d'état processeur

  registre CPU
- Contexte Mémoire : concerne le procédé et se trouve dans la mémoire.

2.2 Problème de commutation de contexte

2.2.1 Insuffisance de l'éxécution séquentielle

L'éxécution est une suite d'activités.

problème de protection mutuelle

Il faut trouver un moyen pour que les processus se "partagent" le processeur quand il y en a plusieurs .

Asynchronisme des évènements

Ce sont des évènements extérieurs au programme.
C'est pour cela qu'il faut faire des mécanismes de commutations.

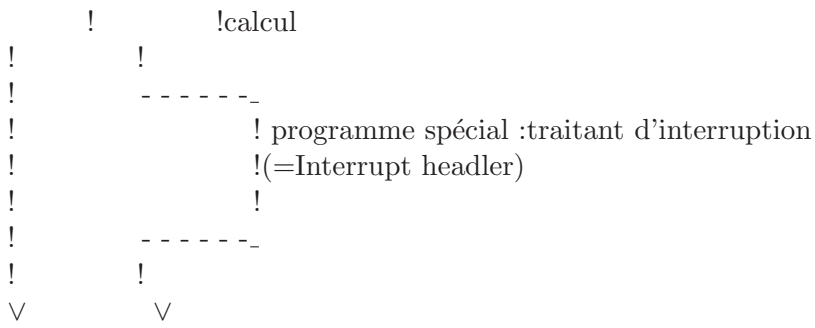
2.2.2 Les 3 mécanismes de commutation de contexte

nom	cause	emploi
Interruption (=Interrupt)	extérieure à l'instruction en cours	réagir à des évènements asynchrones par rapport au programme
Déroulement (=Trap)	éxécution de l'instruction elle même (ex : division par zéro)	emploi de protection de sécurité du système pour le processeur et son environnement
Appel au superviseur (=svc)	offrir le moyen de commuter son contexte (explicite dans le prog utilisateur)	la possibilité d'exécuter des instructions priviligiées.

2.2.3 Les interruptions

Le mécanisme fondamental de réaction aux évènements asynchrones.

principe


!
!calcul
!
!
!
! programme spécial :traitant d'interruption
!(=Interrupt headler)
!
!
!
!
∨
∨

Chapitre 3

Processus et relations entre processus

3.1 Processus

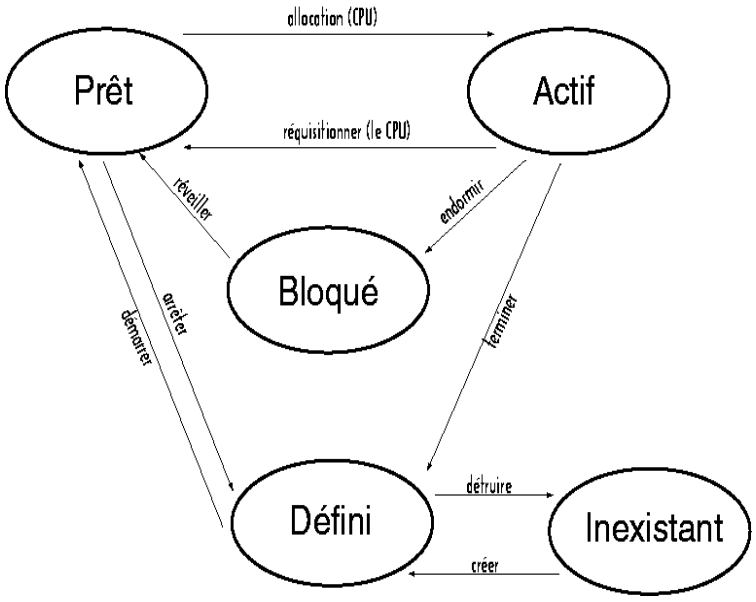
3.1.1 Définition :

- Processus : c'est **une** exécution d'un ou plusieurs programme(s) séquentiels.
- Contexte : états du CPU, de la mémoire et des E/S lors de l'éxécution d'un processus donné.

3.1.2 Noyau d'un système d'exploitation

C'est la partie responsable de la gestion des processus et du matériel. C'est le coeur du SE. Il est monolytique(= micro-noyau)et gère aussi la valeur du quantum (ex. : sur VSD43 : 1 quantum = 0.1 s).

3.1.3 État du processus



3.2 Relations entre processus

3.2.1 Compétition - Coopération

- Compétition : plusieurs processus ont des buts différents et disputent les ressources machine.
- Coopération : plusieurs processus ont un but commun.

3.2.2 Coopération

La situation de base est la suivante :
- P1 produit A
- P2 produit B

- P3 assemble A et B

↪ graphe de précédence (voir section 3.2.4)

3.2.3 Compétition

Toutes les ressources sont mises en compétition : CPU, mémoire, E/S. En particulier, quand 2 processus sont en compétition, des variables peuvent être partagées. Par exemple :
V=10
P1 : V ← V+5
P2 : V ← V*2
Valeur de V après exécution de P1 et P2 ? ? 25 ou 30 selon l'ordre d'exécution des processus !

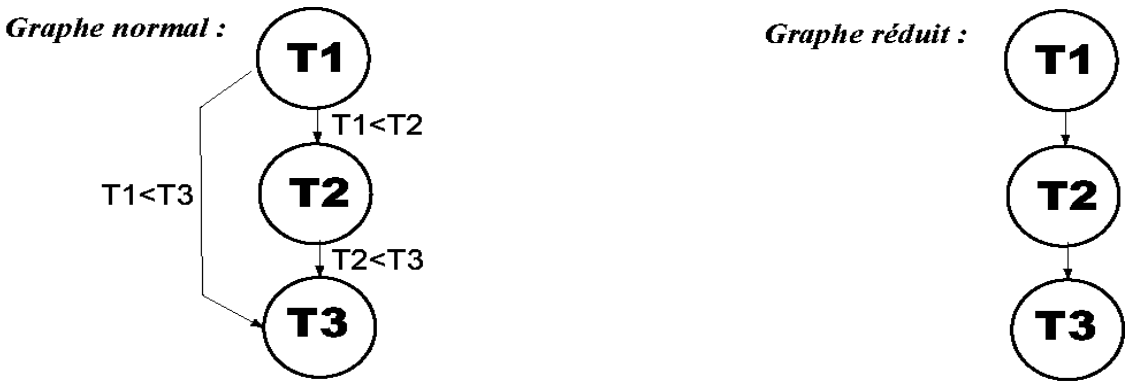
3.2.4 Graphe de précédence

Relation de précédence

Soit E un ensemble de tâches.
Dans E, on définit la relation "est exécuté avant", noté <, de la façon suivante :
- $\forall t \in E, non[t < t]$
- $\forall (t, t') \in E^2, non[(t < t')et(t' < t)]$
- $\forall (t, t', t") \in E^3, [(t < t')et(t' < t")] \Rightarrow (t < t")$

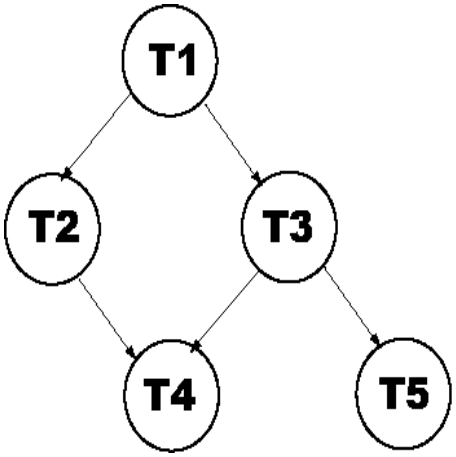
t et t' sont exécutables en parallèle (=concurrents) ssi $[non(t < t')etnon(t' < t)]$

Graphe de précédence

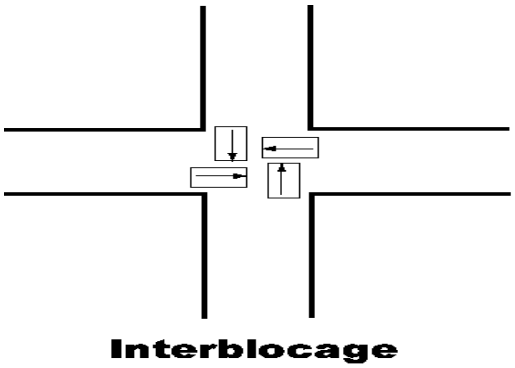


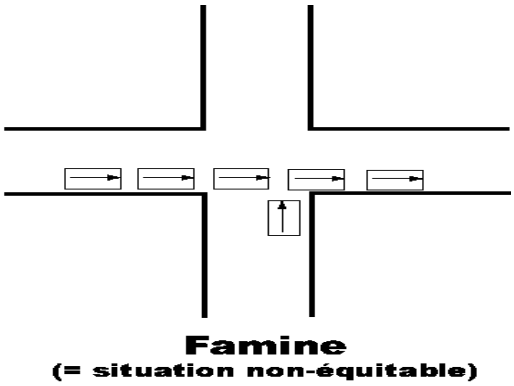
Exemple :
Soient 5 tâches T1 ...T5.
T1 s'exécute avant T2 et T3.

T4 s'exécute après T2 et T3.
T5 s'exécute après T3.

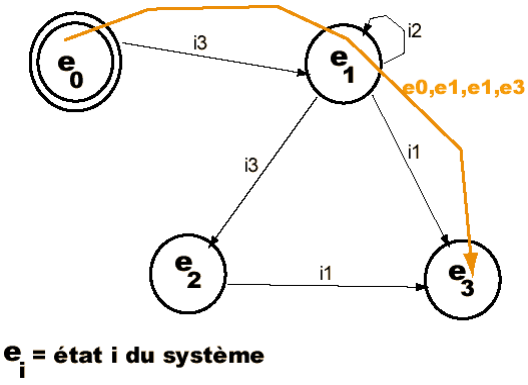


3.2.5 Interblocage (deadlock) et famine (starvation)





3.2.6 Systèmes de transitions étiquetés et automates



3.3 Exclusion mutuelle et sections critiques

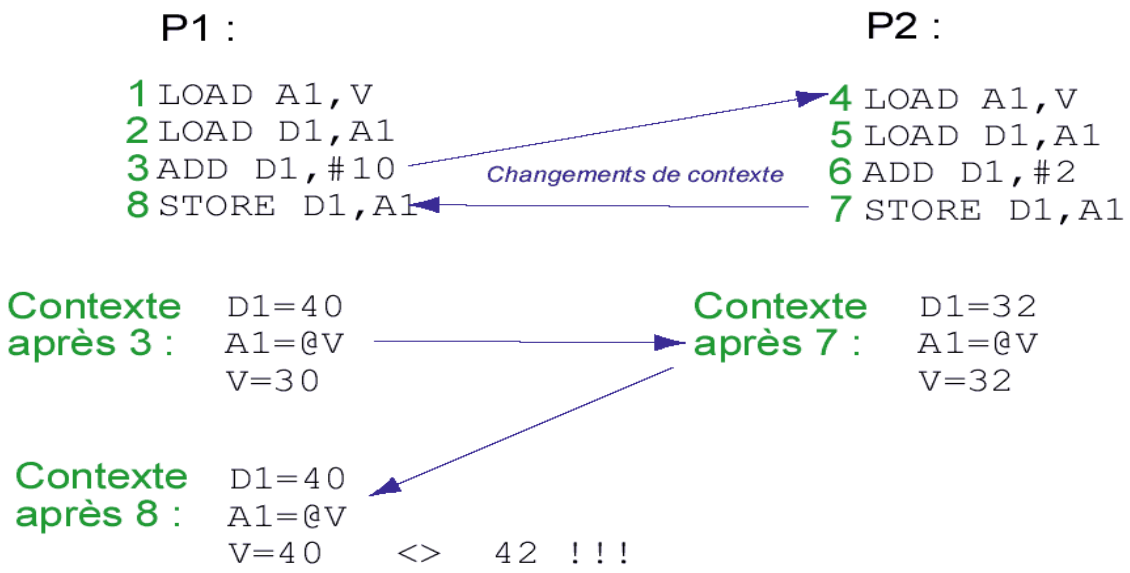
3.3.1 Introduction

Exemple préliminaire :

Soient 2 processus :

```
P1 :    actions A1              |       P2 :    actions A2
        V <- V+10               |               V <- V+2
        actions B1              |               actions B2
```

L'objectif est V reçoit V+12.
Au départ, pour cet exemple, soit V=30.



Section critique (S.C.)

C'est une partie de programme qui doit utiliser une ressource R de manière **exclusive**.

Parties de programmes en exclusion mutuelle

Parties qui sont des sections critiques par rapport à une ressource **commune**.

Spécifications de l'exclusion mutuelle

Processus P_i

- actions avant la S.C.
- demande d'entrée en S.C.
- S.C.
- sortie de S.C.
- actions après S.C.

Exclusion mutuelle

Progression = garantie d'entrée en S.C. (si aucun autre processus ne le demande)

Absence de blocage

Non-famine = attente bornée

3.3.2 Solutions matérielles pour l'exclusion mutuelle

- masquer les interruptions
- test and set (TAS) = lecture/modification indivisible exemple :

```
init :          move #1,cmd
attente :       move cmd,d0
```
empêcher la commutation de contexte ICI
```
                bne attente
```

3.3.3 Solutions logicielles

- attente active : variable(s) partagée(s) → voir en TD
- attente passive (voir section 3.3.4)

3.3.4 Solutions par attente passive : sémaphores

- Sémaphore = `S`
- Compteur du nombre de droits de passage en S.C. = `S.ctr`
- Demande d'entrée en S.C. = `P(S)`
- En sortie de S.C. = `V(S)`
- Valeur initiale de `S.ctr` = `S.ctr0`
- L'exécution de P(S) et de V(S) se fait de manière atomique, c'est-à-dire non-interruptible

```
P(S) :          S.ctr <- S.ctr - 1
                Si (S.ctr<0) Alors
                                * bloquer l'appelant
                                * le placer dans S.f (= file d'attente de S)
                FinSi

V(S) :          S.ctr <- S.ctr + 1
                Si (S.ctr<=0) Alors
                                * sortir un processus (q) de S.f
                                * reveiller (q)
                FinSi
```

Propriétés du compteur

Rappel : le compteur est `S.ctr`

- `S.ctr` = `S.ctr0` - n[`P(S)`] + n[`V(S)`] (où n[x]=nb d'appels)
- Si `S.ctr` ≥ 0 Alors `S.ctr` = nb de droits de passage
- Si `S.ctr` < 0 Alors `S.ctr` = nb de processus bloqués dans `S.f`

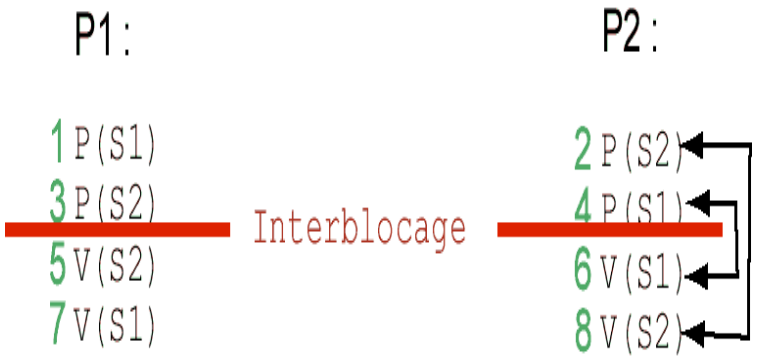
Utilisations pour l'exclusion mutuelle

`S.ctr0=1`

```
                P 1                             P 2
                 .                               .
                 .                               .
        P(S)----------                  P(S)----------
                 .                               .
                S.C.                            S.C.
                 .                               .
```

```
V(S)----------                    V(S)----------
      .                                 .
      .                                 .
```

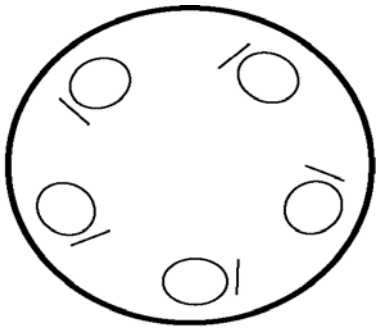
ATTENTION aux interblocages



3.4 Problèmes de synchronisation

3.4.1 Les philosophes

L'idée est la suivante : 5 philosophes chinois dînant autour d'une même table ont chacun besoin de 2 baguettes pour manger, mais n'en disposent que de 5 au total. Il faut donc trouver une solution équitable de synchronisation de leurs actions, afin de leur permettre de manger tour à tour (seuls 2 peuvent manger simultanément).



Penser

Manger ----------> acquérir les 2 baguettes
manger
rendre les 2 baguettes

3.4.2 Lecteurs / Rédacteurs

Voir TD + cours SYSB (2nd semestre)...

3.4.3 Producteurs / Consommateurs

Voir TD + cours SYSB (2nd semestre)...

3.4.4 Spool d'impression

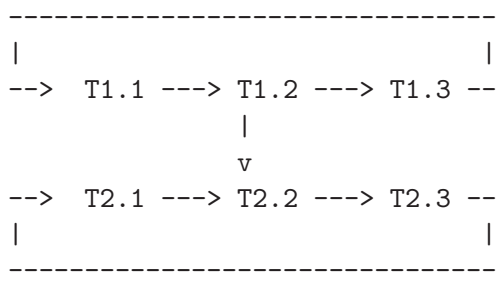
Voir TD + cours SYSB (2nd semestre)...

3.5 Outils de synchronisation

3.5.1 Sémaphores

- exclusion mutuelle
- synchronisation temporelle

Exemple :



où on veut que debut(T2.2) $\geq$ fin(T1.2)
Utilisons un sémaphore `S` tel que `S.ctr0` $= 0$.

3.5.2 Moniteurs

- Exclusion mutuelle pendant l'exécution des procédures et fonctions du moniteur.
- $\hookrightarrow$ voir TD

3.5.3 Compteurs d'évènements

Voir cours SYSB (2nd semestre)...

3.5.4 Transfert de messages

Voir cours SYSB (2nd semestre)...

Chapitre 4

Gestion des processus sous UNIX

4.1 Noyaux des Systèmes d'exploitation

4.1.1 Structuration du noyau

- En couches : verticalement de la couche n à la couche matérielle (0), chaque couche p ne peut être liée qu'à la couche p-1
- Micro-noyau : voir section 1.4.4
- Monolytique : par exemple UNIX (même s'il évolue actuellement vers une structure à micro-noyau)

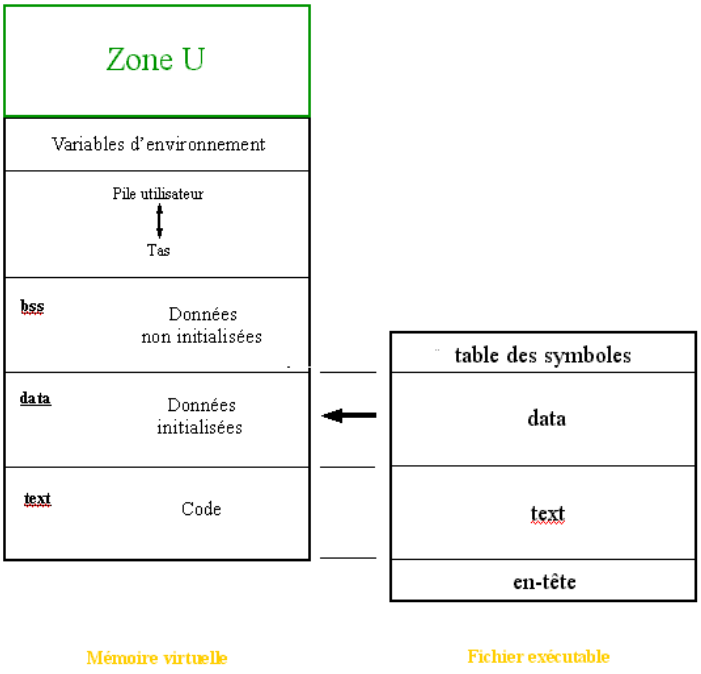
4.1.2 Le noyau d'UNIX

- tout processus possède 2 modes d'exécution : *user* (utilisateur) et *kernel* (noyau), d'où l'existence de 2 contextes pour chacun d'entre eux
- exécuter le noyau, c'est basculer du mode *user* vers le mode *kernel*
- le code et les données du noyau sont partagées
- chaque processus possède un identificateur unique (PID)
- le processus initial (au démarrage du système) est le *swapper* et son PID=0

4.2 Processus UNIX

4.2.1 Mémoire virtuelle

- C'est l'espace d'adresses du processus (en général d'une taille très supérieure à celle de la mémoire physique disponible)
- Cartographie = correspondance fichier exécutable $\longleftrightarrow$ mémoire virtuelle
- Zones dans la mémoire : text (code) + data (données initialisées) + bss (données non-initialisées) = régions (qui peuvent être partagées). Une région est une zone contigue d'adresses virtuelles.

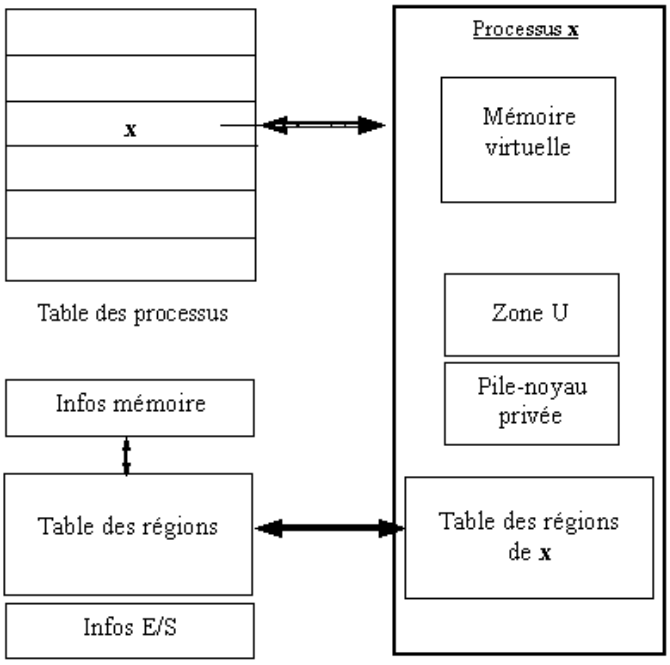


4.2.2 Contextes d'un processus

Rappel de la définition d'un contexte

C'est l'ensemble des états de la mémoire virtuelle, des mots d'états CPU et des registres, ces 2 derniers éléments étant composés de :

- Une entre dans la table des processus :
  - ordonnancement
  - identité : pid, ppid, gid et uid
  - mémoire : zone u, table des régions
  - signaux
- Une table des régions des processus



Contextes d'un processus

- contexte utilisateur
- contexte système

4.2.3 Naissance, vie et mort d'un processus

Mécanisme fork

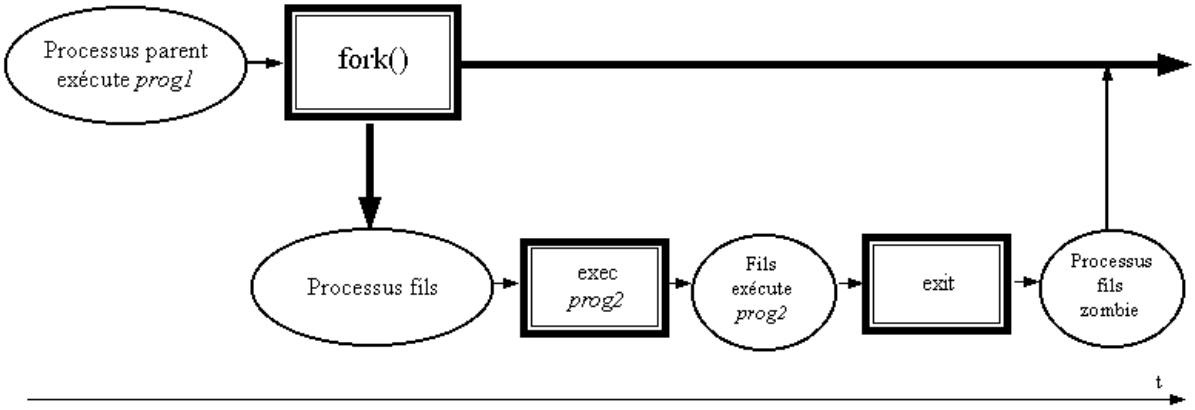
1 processus est TOUJOURS créé par un autre processus (sauf swapper)
⇒ présence de processus pères/fils

Mécanisme exec

Ce mécanisme entraîne un changement de code exécutable et de contexte utilisateur (voir en TD).

Mécanisme exit

Fin d'exécution.



Voir aussi figure 6.1 distribuée en cours . . .

4.2.4 Initialisation - Mécanisme du Shell

Boot

- initialisation des E/S
- chargement du swapper(0)
- init(1) → root
- getty

Mécanisme du Shell

sh : commande → fils

4.3 Création des processus par l'utilisateur

4.3.1 Exemple

1

```
main()
{
int a,n;
a=1;
printf("Avant fork, pid=%d\n",getpid());
n=fork();
```

C'est ICI que le fils est créé. Si c'est le père qui est exécuté, n=pid(fils), sinon n=0 (cas du fils).

2 (père)

```
if (n!=0)
        {
        printf("Pere - pid de mon fils = %d\n",n);
        fflush(stdout);
        a=2;
        printf("Pere - a=%d\n",a);
        fflush(stdout);
        }
```

3 (fils)

```
else
        {
        printf("Fils - mon pid = %d\n",getpid());
        fflush(stdout);
        a=3;
        printf("Fils - a=%d\n",a);
        fflush(stdout);
        }
```

4 (commun)

```
printf("Code commun - pid=%d, a=%d\n",getpid(),a);
}
```

4.3.2 Style d'écriture du code

Sans limitation, mais nécessitant parfois de nombreux if...else imbriqués :

```
if ((n=fork())==0)
        fils();
else
        {
```

```
        /* code du pere */
        /*     ....     */
        }
```

Plus lisible en général, mais ATTENTION ! ici, le fils ne doit JAMAIS revenir (exit à la fin obligatoire) :

```
if ((n=fork())==0)
        fils();
/* code du pere */
/*     ....     */
```

où l'on a obligatoirement `fils()` de la forme :

```
/* code du fils */
/*     ....     */
/*     ....     */
/*     ....     */
exit(_);
```

4.4 Changement d'environnement d'exécution

4.4.1 Principe

Voir section 4.2.3
Le moment où le mécanisme `exec` est appelé est nommé *recouvrement*.

4.4.2 Exemple type

```
execl("cp","nouveaucp","fic1.c","fic2.c",NULL);
```

où :
- le 1er paramètre (`"cp"`) est le nom du programme exécutable à exécuter
- le 2nd paramètre (`"nouveaucp"`) est le nom du programme tel qu'il sera vu par la commande `ps`
- les paramètres suivants sont les paramètres du programme à exécuter, et la liste (de taille variable) est terminée par NULL.

4.4.3 Différentes versions de exec

Pour connaître les différences entre `execl`, `execle`, `execlp`, `execv`, `execve`, `execvp`, consulter la commande `man exec` sous Unix !

4.5 Synchronisation

4.5.1 Relations père/fils

On place le père dans l'état "endormi" jusqu'à ce qu'un de ses fils ne meure, par l'appel de :
`n=wait(&cr);`
qui retourne n=-1 si pas de fils, et n=pid(fils mort) dans le cas contraire.
`cr` est alors une variable de 16 bits, dont les 8 de poids fort contiennent le code d'exit du fils, et les 8 de poids faible contiennent 0 en cas de fin normale, et le numéro du signal ayant tué

le fils dans le cas contraire.

4.5.2 Assassinat d'un processus connu

`kill(p,SIGKILL)`
où p=pid du processus à tuer, qui doit être de même uid que le tueur.
Sous Shell, on utilise : `kill -9 p`

4.5.3 Attente endormie d'événement

`sleep(n)` : attends pendant `n` secondes
`pause()` : attente endormie "infinie"
Dans les 2 cas, le reveil peut tout de même être provoqué par un autre processus qui enverrait un signal.

4.6 Ordonnancement

4.6.1 Introduction

Le but de l'ordonnancement (scheduling) est de "choisir" le processus à exécuter, et éventuellement sur quel CPU.
Dans un système "temps réel", l'utilisateur choisit complètement l'ordonnancement. Dans notre cas, c'est le système d'exploitation qui réalise ces choix, via l'ordonnanceur (scheduler).

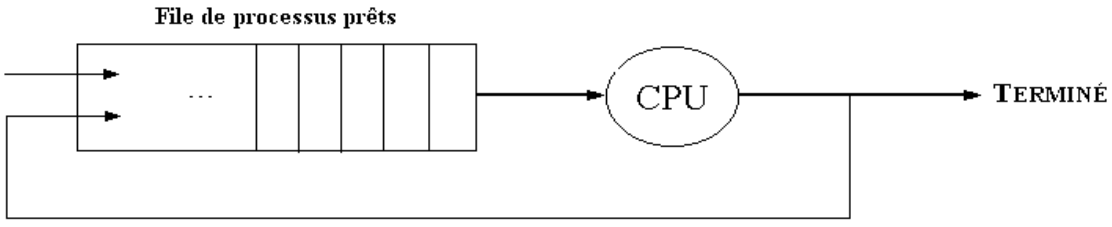
4.6.2 Méthodes d'ordonnancement

Critères

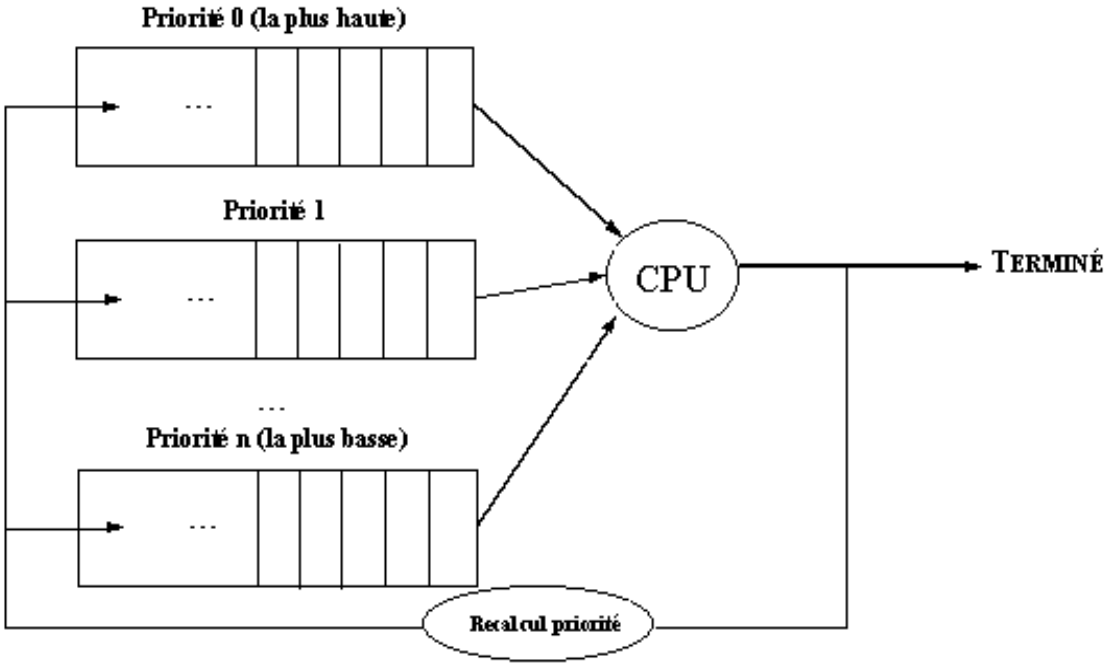
- priorités
- ressources nécessaires et disponibles
- valeur du quantum
- durée d'exécution déjà réalisée

3 exemples

- plus haute priorité
- tourniquet simple (= round robbin)



- tourniquet à files multiples



En pratique, ce sont les processus de temps d'exécution court qui sont privilégiés.

Sous UNIX

Sous un système de type UNIX, un utilisateur (même le superutilisateur root !) ne peut pas augmenter la priorité de l'un de ses processus. Par contre, il peut la diminuer grâce à la commande `nice (n)` qui demande de diminuer la priorité de ses processus de `n` niveaux.

Chapitre 5

Fichiers sous UNIX

5.1 Introduction

5.1.1 Fichier, système de fichiers (SF) et système de gestion de fichiers (SGF)

Fichier

- c'est un ensemble structuré d'informations dont la durée de vie est supérieure à la durée d'exécution du programme
- il est placé (en général) sur un support permanent de stockage
- il est composé de :
  - un nom interne unique (en général un nombre entier)
  - un état : données dans le fichier, informations sur le fichier (droits, ...), etc...
  - des fonctions d'accès (l'accès proprement dit est réalisé par le SGF, voir ci-dessous)

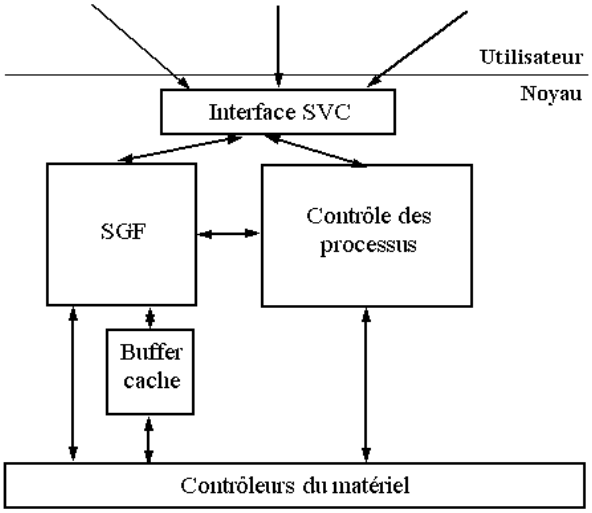
SGF

- c'est une partie du système d'exploitation, chargée d'assurer la maintenance des fichiers, et qui permet de réaliser les fonctions d'accès
- 2 vues sont disponibles :
  - organisation logique (vue de l'utilisateur)
  - organisation physique (vue de la machine)

SF

- c'est la correspondance logique/physique
- exemples sous UNIX : ufs (Unix File System), proc, boot, S5, ...

5.1.2 Le SGF de UNIX



5.2 Structure du SF sur disque

5.2.1 Structure générale

Un disque physique est partagé en volumes et partitions. Une partition est une suite de blocs (tous de même taille), structurée de la manière suivante :

boot block (0)	super block (1)	liste des inodes	blocs de données

2 commandes essentielles : `mkfs` (make filesystem) et `fsck` (filesystem check).

5.2.2 Le superblock

Il décrit l'état complet du SF. Il contient les informations suivantes :

- taille du SF
- nombre de blocs libres
- liste des n° de blocs libres
- index du premier bloc libre de cette liste
- nombre d'inodes dans le SF
- nombre d'inodes libres
- liste des n° d'inodes libres

5.2.3 Fichiers et inodes

Un fichier est un couple (inode, liste de blocs de données).

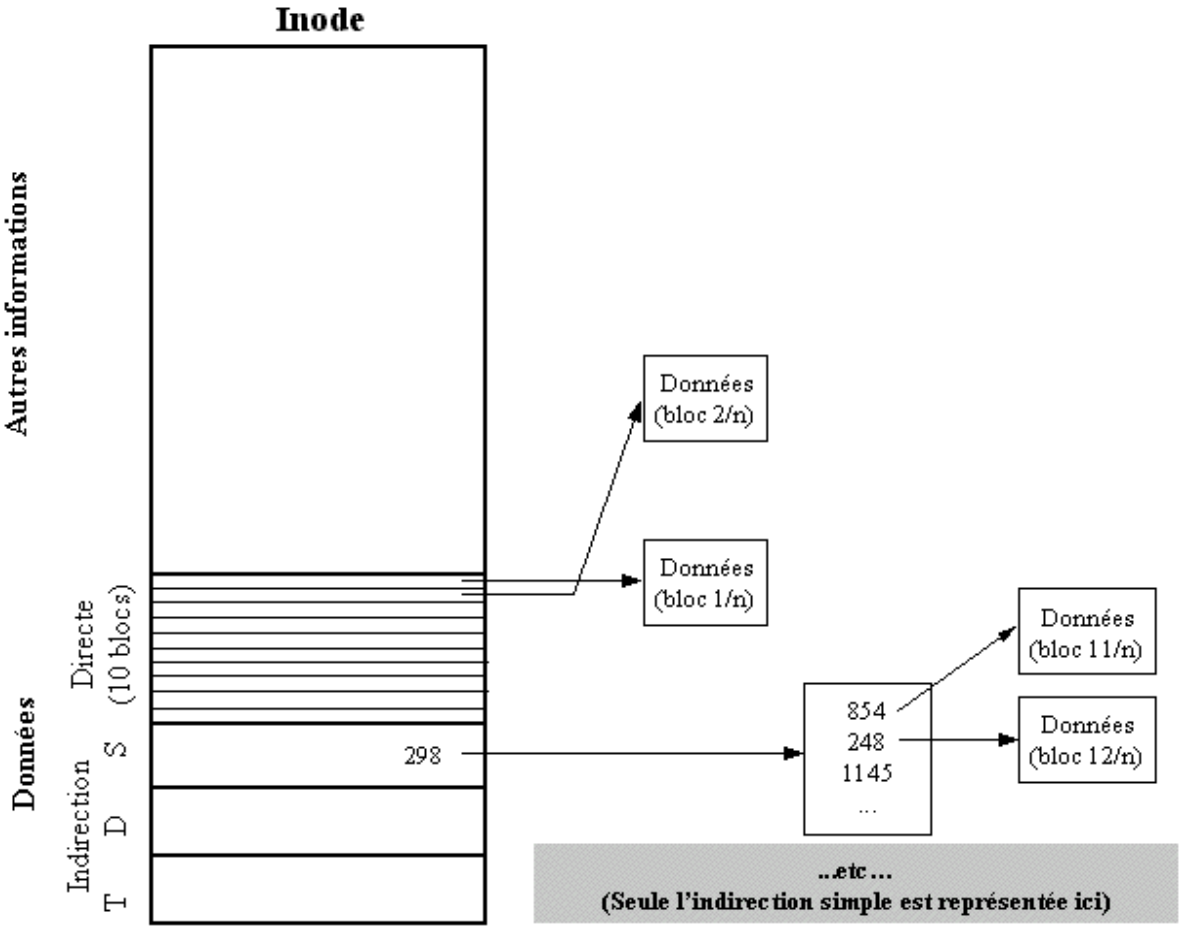
Types de fichiers

- ordinaires
- répertoires
- périphériques (`/dev`)

– tubes nommés (pipes)

L'inode contient :

– propriétaire individuel
– groupe propriétaire
– type du fichier
– droits d'accès
– date de dernière modification ou de dernier accès
– date de dernière modification de l'inode
– nombre de liens au fichier
– taille du fichier (en octets)
– données, c'est-à-dire n° des blocs contenant les données. 4 zones :
  – partie directe : 10 adresses de blocs
  – indirection simple : 1 bloc d'adresses
  – indirection double : 1 bloc d'adresses de blocs d'adresses
  – indirection triple : 1 bloc d'adresses de blocs d'adresses de blocs d'adresses



5.2.4 Répertoires

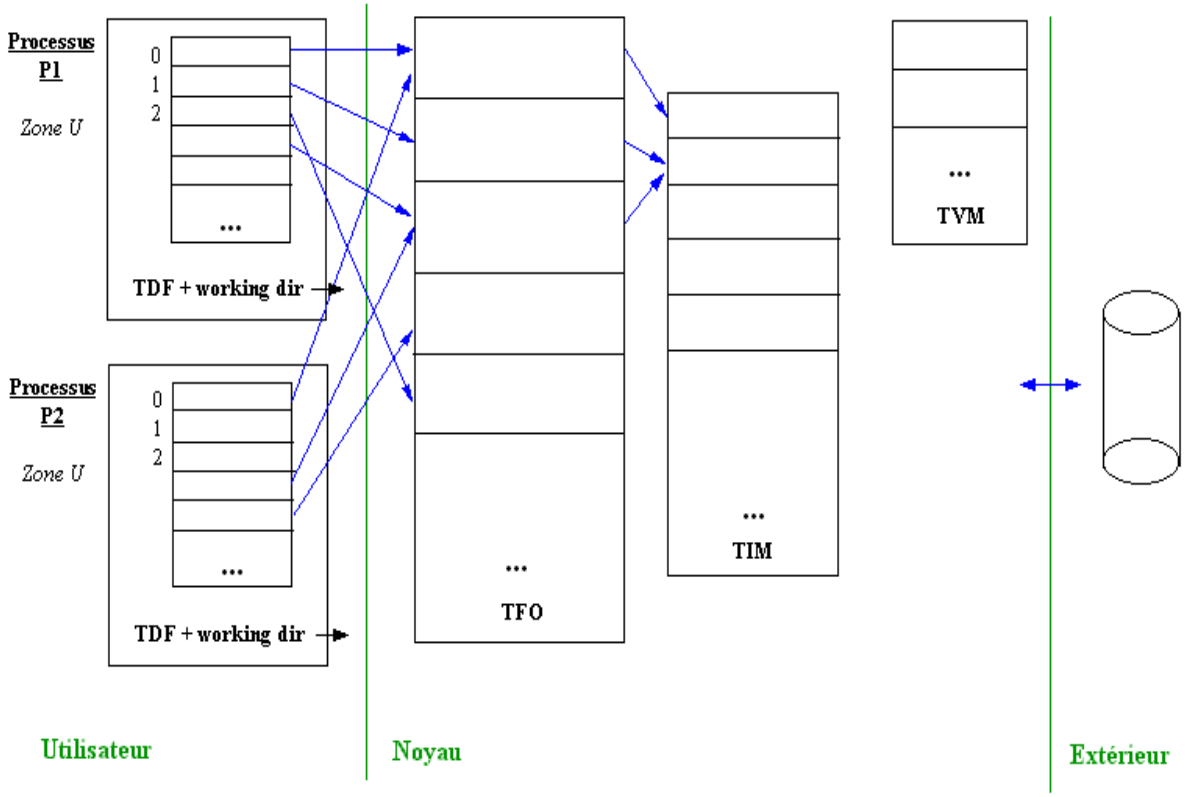
C'est un fichier qui contient les associations (n° d'inode, nom en texte).
L'inode du `/` (`root`) du SF = n° 2.
Si n° d'inode = 0, c'est qu'il n'y a plus d'association.
Liaisons :

– directe : `link fexistant nouveauf`

– symbolique : `link -s fexistant nouveaufsymbolique`

5.3 Structures de données du SGF en mémoire

5.3.1 Tables du SGF



– 1 fichier = 1 entrée dans TDF (équivalent au n° de l'entrée, càd le descripteur de fichier, qui est un entier).
– TDF = Table des Descripteurs de Fichiers (1 par processus !!). Sa configuration minimale par défaut (à la création d'un processus) est la suivante :

  – 0 : `stdin`
  – 1 : `stdout`
  – 2 : `stderr`
– TFO = Table des Fichiers Ouverts (1 seule !!)
– TIM = Table des Inodes en Mémoire. Remarque : à un instant donné, 1 inode en mémoire peut être dans un état différent de celui sur le disque.
– TVM = Table des Montages Virtuels

5.3.2 Inode en mémoire

Il a la même structure qu'un inode sur le disque, à laquelle s'ajoutent les informations suivantes :

– état de l'inode (pour gérer les accès concurrents)
– n° de périphérique
– n° d'inode
– chaînage des inodes en mémoire
– compteur de références

5.3.3 Structures de gestion des ressources physiques

Il est réservé en mémoire centrale une petite liste de n° libres d'inodes et de blocs de données.

5.3.4 Bibliothèque standard - Descripteurs

Par la bibliothèque standard

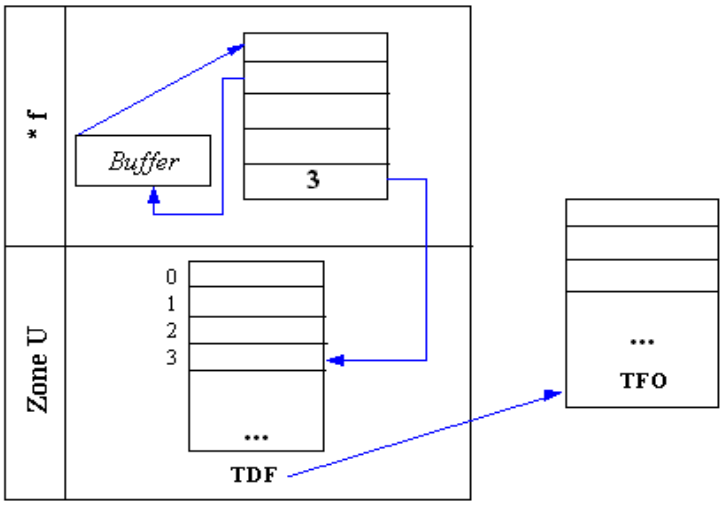
Dans le code :

```
FILE *f;
f=fopen(char *nomfic, char *mode);
```

est fait un appel sous-jacent au S.E. : `f` est un objet de type `struct _iobuf` définie avec les champs suivants :

- `char * _base;` (adresse du tampon)
- `char * _ptr;` (pointeur sur la position courante dans le tampon)
- `int _cnt;` (nombre de caractères dans le tampon APRES la position courante. S'il devient strictement négatif, alors la suite du fichier est chargée dans le tampon)
- `char flag;` (mode de traitement du fichier)
- `char _file;`

Au niveau système, pour un processus donné



5.4 Fonctions utilisateur de gestion des fichiers UNIX

5.4.1 Open - Close

Fonction d'ouverture :
`int open(char *nomfic, int mode, int droits);`
Cette fonction (de bas niveau) n'est pas *portable*, car elle travaille **directement** sur le SF du SE!!!
Elle renvoit le n° de descripteur créé (-1 en cas d'erreur), qui est **toujours** le plus petit n° libre.
Ses arguments sont :

- `nomfic` = nom du fichier à ouvrir
- `mode` = l'une des macros suivantes de `fcntl.h` :
  - `O_RDONLY` (lecture seulement)
  - `O_WONLY` (écriture seulement)
  - `O_NDELAY` (rend les E/S non bloquantes)
  - `O_APPEND` (écriture en mode ajout)
  - `O_CREAT` (crée le fichier)
  - `O_TRUNC` (vide le fichier si déjà existant)
  - `O_EXCL` (mode exclusif - renvoie une erreur si le fichier existe déjà)
  - exemple de combinaison : en lecture ET sans délai `O_RDONLY | O_NDELAY` (attention ! du point de vue interne, c'est un OU logique !)
- `droits` = idem à ceux de `chmod`

Fonction de fermeture :
`int close(int fd);` où `fd` = descripteur de fichier.

5.4.2 Read - Write

Fonction de lecture :
```
int read(int d, char *tamp, int nb);
```
Elle renvoit le nombre d'octets effectivement lus (si -1 :erreur, si 0 :fin de fichier).
Ses arguments sont :

- `d` = n° de descripteur de fichier
- `tamp` = tampon à utiliser
- `nb` = nombre d'octets à lire

Fonction d'écriture :
```
int write (int d, char *tamp, int nb);
```
qui fonctionne exactement sur le même principe que `read` !

5.4.3 Atomicité

Pendant l'exécution d'un tel appel système, aucun autre processus ne peut accéder au même fichier, du point de vue logique.

5.4.4 Liaison haut/bas niveau

On peut passer du bas au haut niveau (utile pour retrouver les fonctions de formatage, notamment), par la fonction :
```
FILE * fdopen(int d, int mode)
```

5.5 Fonctions utilisateur avancées

5.5.1 Héritage des descripteurs

Un processus fils hérite des descripteurs de fichiers ouverts par son père. Attention ! la position courante dans le fichier est stockée dans la TFO, qui est commune !!!

5.5.2 Redirection des E/S standards

Exemple préliminaire

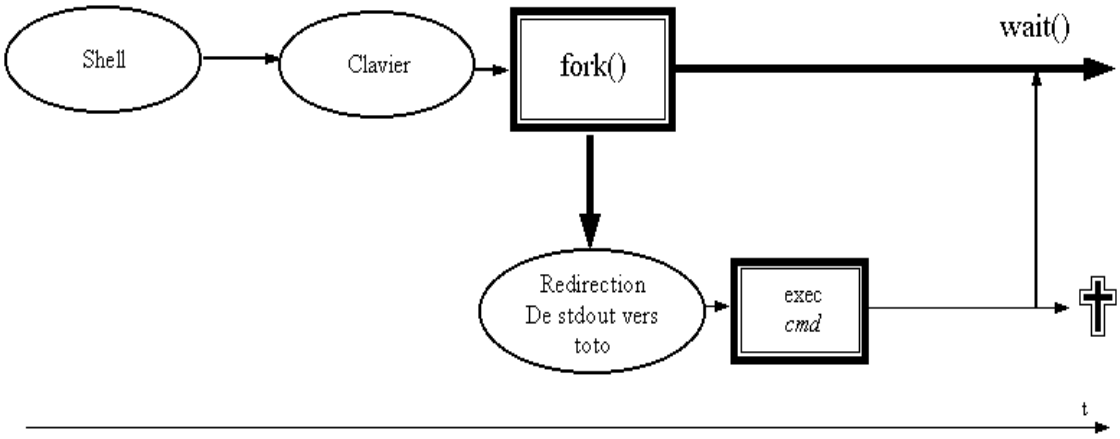
```
d=open("erreurs",O_CREAT,0755);
close(2);
dup(d);
```

Ce programme crée un fichier `erreurs` de descripteur `d`, puis ferme le fichier `stderr` (n° 2), puis duplique `d` dans le plus petit descripteur libre (forcément 2 ici). Donc, en fin de compte, ce programme redirige `stderr` vers le fichier `erreurs` !

Redirection du Shell

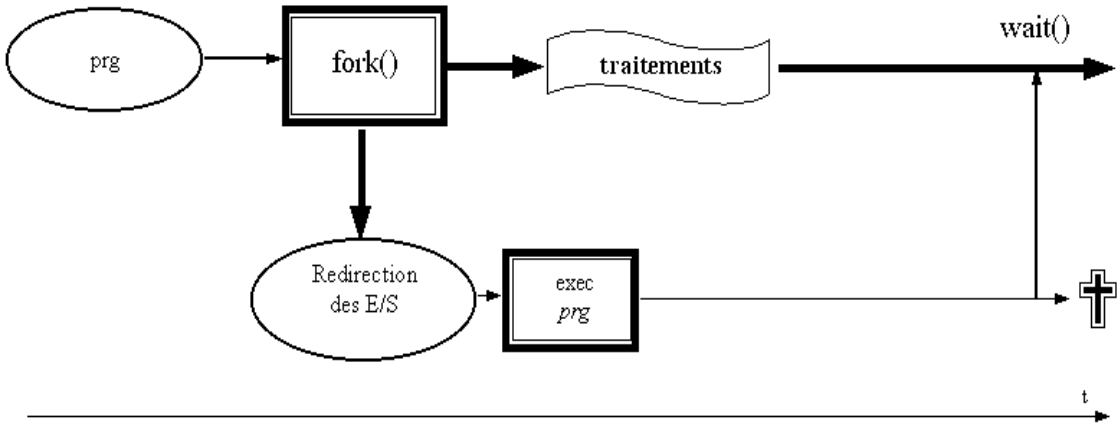
`$ cmd > toto` effectue les actions suivantes :

1. `fork`
2. redirige `stdout` dans le fils
3. `exec` dans le fils



Utilisation d'un binaire exécutable comme fonction dans un programme prg

1. créer un fils
2. redirige les E/S du fils dans ce programme prg
3. `exec(prg)` dans le fils



5.5.3 Positionnement dans le fichier

```
long lseek (int d,long dpl,int origine);
```

où `origine` = `SEEK_SET` (depuis le début du fichier), `SEEK_CUR` (depuis la position courante) ou bien `SEEK_END` (depuis la fin du fichier, et dans ce cas `dpl` est tout de même un nombre positif).

5.5.4 Changement de mode d'un fichier

```
int fcntl (int d,int cmd,int options)
```

Par exemple, pour passer un fichier en mode non-bloquant, on utiliserait :

```
fcntl(descr,F_SETFL,fcntl(descr,F_GETFL,0)|O_NDELAY)
```

où `fcntl(descr,F_GETFL,0)` renvoit les bits de mode courant du fichier de descripteur `descr`.

5.5.5 Verrouillage

Il s'agit ici des moyens pour bloquer l'accès aux fichiers, ou à certaines parties d'un fichier. Pour le détail : voir cours de SYSB (2nd semestre).

5.6 Fonctions utilisateur de gestion du SF

5.6.1 Gestion des propriétés des fichiers

- `int stat(char *nomfic,struct stat *p);`
  ou `int fstat(int desc,struct stat *p);`
  permet de récupérer dans `p` les infos contenues dans l'inode
- `int chmod(char *nomfic,int droits);`
  ou `int fchmod(int desc,int droits);`permet de changer les droits sur un fichier
- `chown` ou `fchown` permet de changer le propriétaire d'un fichier
- `umask` définit les droits par défaut (masque) des futurs fichiers
- `link`, `symlink` (BSD), `unlink` et `readlink` sont disponibles
- `mkdir` / `rmdir` / `chdir-fchdir` / `chroot`

5.7 Gestion du SF dans son ensemble

5.7.1 Montage et démontage des SF

- mount
- unmount

5.7.2 Quotas

5.7.3 Fonctions internes du SGF

Chapitre 6

Tubes (pipes)

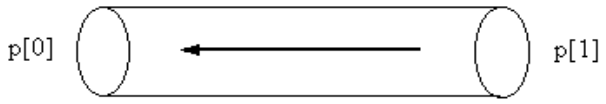
6.1 Tubes anonymes

6.1.1 Définition

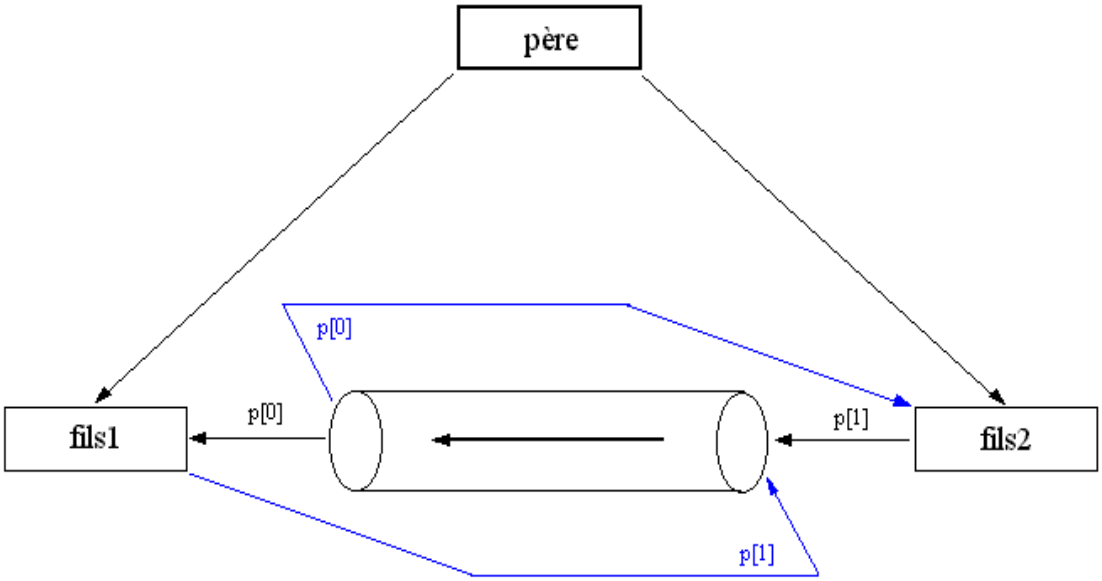
C'est un canal de communication unidirectionnel, de type FIFO, et constitué d'un flot d'octets. Sa durée de vie est limitée à celle du processus qui l'a créé. Un tube anonyme n'a pas d'existence dans le système de fichiers, mais constitue un canal de communication fiable. Un tube anonyme ne peut être manipulé que par le processus qui l'a créé ou ses descendants.

6.1.2 Création

```
int p[2];
pipe(p);
```



6.1.3 Utilisation



père :

```
int p[2];
pipe(p);
....
...=fork();
...=fork();
....
```

fils1 :

```
close(p[1]);
....
read(p[0],&rcv,longueur);
....
```

fils2 :

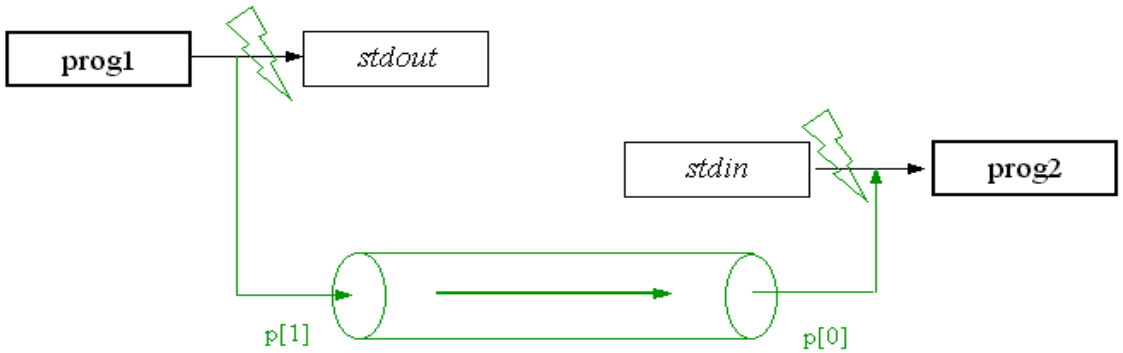
```
close(p[0]);
....
write(p[1],&msg,longueur);
....
```

6.1.4 Particularités

- à chaque lecture, les données lues sont supprimées du tube
- `read` retourne 0 s'il n'y a plus d'écrivain ni de données pour ce tube
- lorsqu'il n'y a plus de lecteur, chaque processus écrivain reçoit le signal `SIGPIPE`
- taille du tampon de tube : `PIPE_BUF` (POSIX)

6.2 Redirection des E/S standards

6.2.1 Enchaînement de 2 programmes



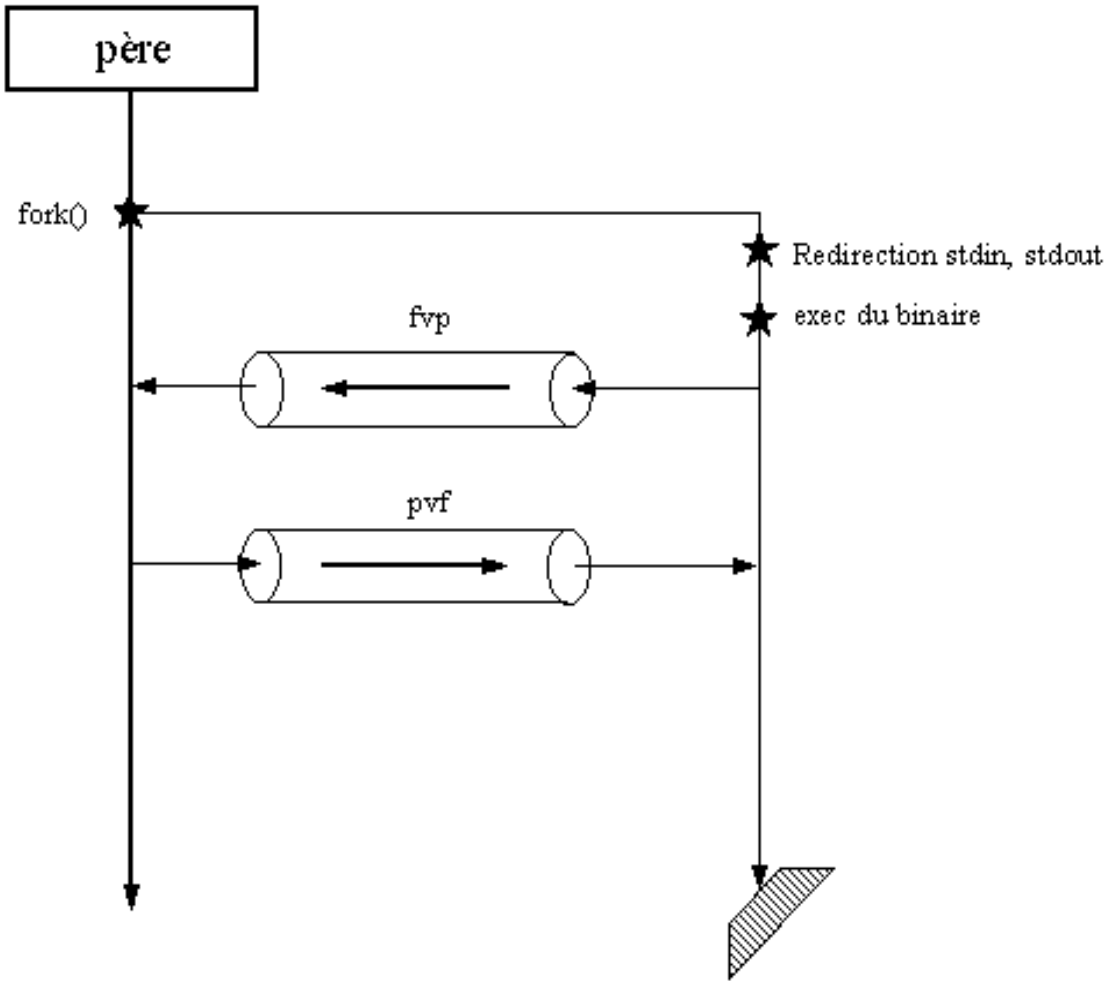
Exemple du shell qui exécute `cat toto | wc -l` :

```
shell()
{
        pipe(p);
        if ((n1=fork())==0) fils1;
        if ((n2=fork())==0) fils2;
        close(p[0]);
        close(p[1]);
        wait(&cr); wait(&cr);
}


void fils1()
{
        close(p[0]);
        close(stdout);
        dup(p[1]);
        execl("cat","cat","toto",NULL);
}


void fils2()
{
        close(p[1]);
        close(stdin);
        dup(p[0]);
        execl("wc","wc","-l",NULL);
}
```

6.2.2 Utilisation d'un binaire




```
void pere()
{
        int fvp[2],pvf[2];
        ....
        pipe(fvp);
        pipe(pvf);
        if ((n=fork())==0) fils;
        close(fvp[1]);
        close(pvf[0]);
        ....
        write(pvf[1],&msg,....);
        ....
        read(fvp[0],&rcv,....);
        ....
        if ((r=read(...)==0)
                {
                /* le fils a ferme stdout */
                ....
                }
}
```

```
void fils()
{
        close(fvp[0]);
        close(pvf[1]);
        close(stdin); dup(pvf[0]);
        close(stdout); dup(fvp[1]);
        exec(....);
}
```

6.3 Tubes nommés

C'est une extension des tubes ordinaires (anonymes). Ici, des processus indépendants peuvent avoir accès à un même tube.

6.3.1 Définition d'un named pipe ou FIFO

C'est un tube ayant une existence permanente dans le SF. S'il n'est pas utilisé, sa taille est rigoureusement nulle.

6.3.2 Création

`int mknod(nomFIFO, S_IFIFO|permissions, 0);` (UNIX standard), où `nomFIFO` est de type `char *`
`int mkfifo(nomFIFO, mode)` (POSIX)
`mknod nomFIFO` (sous Shell)

6.3.3 Propriétés et particularités

Les processus doivent se "mettre d'accord"

Qui crée le tube ?

Utilisation (ouverture + écriture OU lecture)

Ouverture en lecture : `p=open(nomFIFO,O_RDONLY,0);`
Ouverture en écriture : `p=open(nomFIFO,O_WRONLY,0);`
ATTENTION !!! L'ouverture en lecture (respectivement en écriture) est BLOQUANTE s'il n'y a aucun écrivain (respectivement lecteur) sur ce tube, à moins d'ajouter le paramètre `O_NDELAY`.