

# **INTRODUCTION**

## **au**

## **ystème**

## **Unix**

Eric Gressier

**CNAM-CEDRIC**

# PLAN

1. Introduction
2. Processus
3. Gestion Mémoire
4. Fichiers et Tubes
5. IPC System V

## REFERENCES BIBLIOGRAPHIQUES

**Unix Network Programming.** W. Richard Stevens. Prentice Hall Software Series.1990.

**Advanced Programming in the Unix Environment.** W. Richard Stevens. Addison Wesley.1992.

**The Design and Implementation of the 4.3 BSD Unix Operating System.** S.J. Leffler. M. K. Mc Kusick. M.J. Karels. J.S. Quaterman. Prentice Hall.1989.

**The Magic Garden Explained. The Internals of UNIX SYSTEM V RELEASE 4. An open systems design.** B. Goodheart. J. Cox. Prentice Hall.1994.

*Eric Gressier*

# 1. INTRODUCTION

# Les Unix

Il existe plusieurs UNIX, mais les plus répandus sont :

- Unix System V Release 4
- Unix 4.4 BSD

Et une standardisation de l'interface des appels systèmes : POSIX.

Pour la partie communication, les travaux majeurs ont été développés à partir de la famille BSD. Mais, aujourd'hui, on peut quasiment dire que tout est dans tout et réciproquement.

Unix System V release 4 semble marquer l'avantage sur l'Unix BSD arrêté à la version 4.4.

Dans la présentation le noyau 4.3 BSD a été choisi, Ultrix (DEC) a été construit à partir de cette souche.

## 2. Processus

## Processus

Processus = instance d'un programme exécutable en mémoire

Programme source :

```
main (argc, argv)
```

```
int argc;          /* nombre d'arguments, nom du programme inclu */
```

```
int *argv[];       /*tableau de pointeurs sur la liste des arguments :
```

```
argv[0] -> "nom du programme\0"
```

```
argv[1] -> "1er argument\0"
```

```
...
```

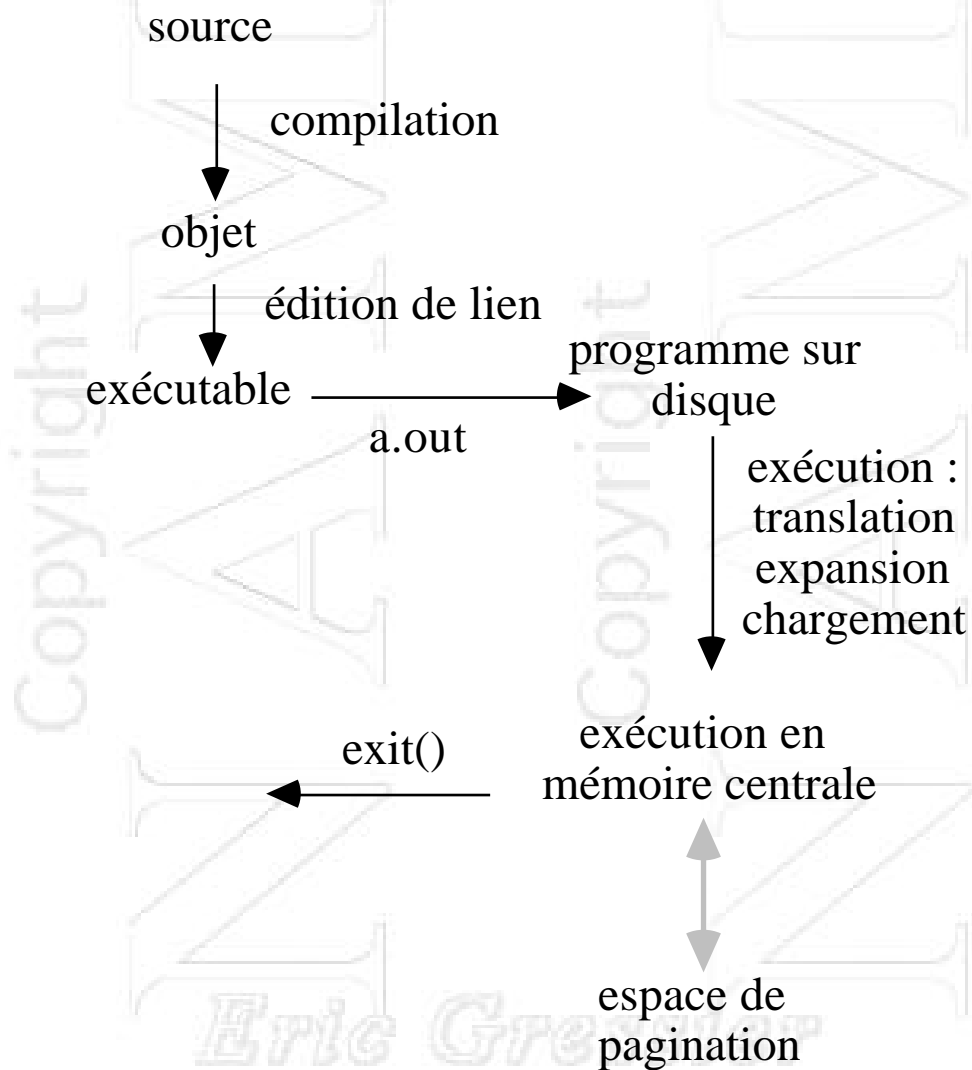
```
*/
```

```
{
```

```
corps du programme
```

```
}
```

# Génération d'un exécutable et utilisation d'un exécutable





## Image d'un processus sur disque

structure décrite par a.out.h :

Magic number	
entête	
Code (text)	
données initialisées readonly	zone données initialisées
données initialisées read-write	
table des symboles	
	liens utilisables pour une autre édition de liens

### Magic Number :

si #! dans a\_magic, le fichier est exécutable par un interpréteur :

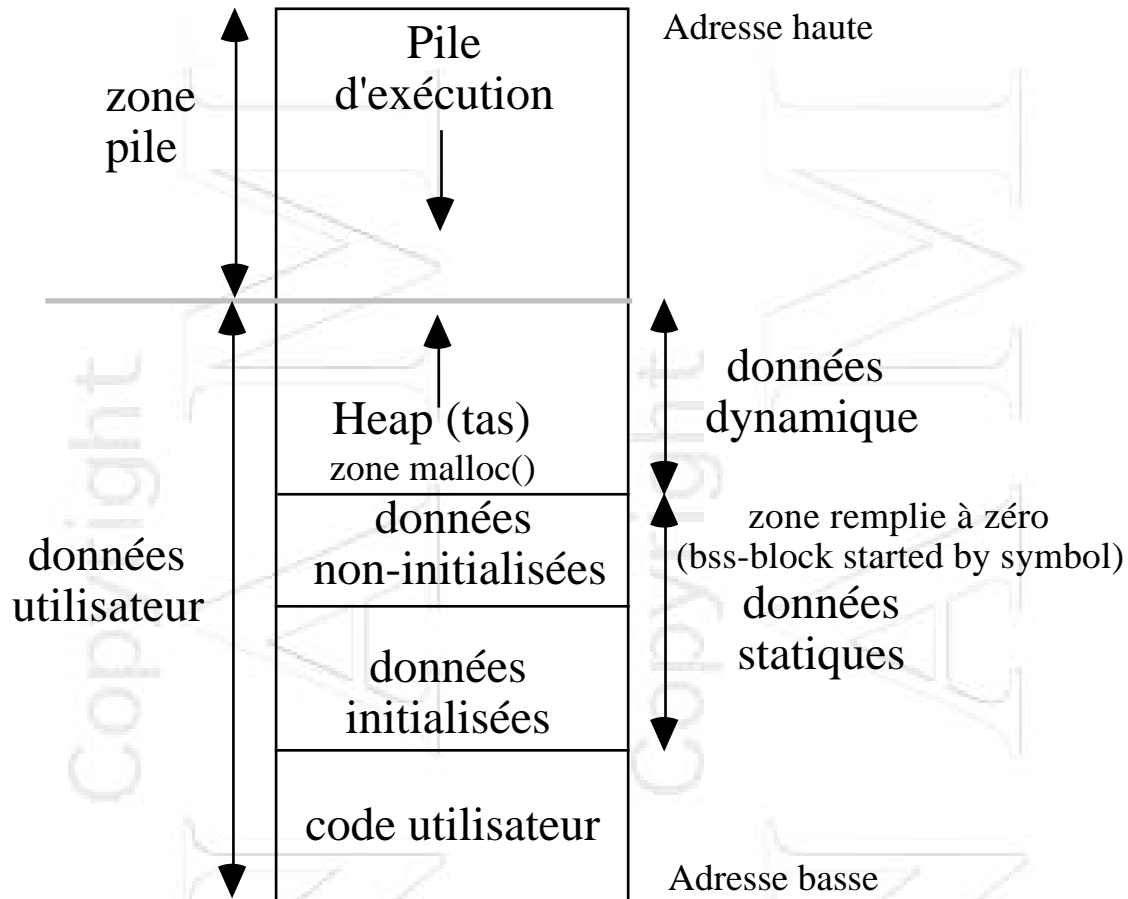
#! est suivi par le chemin d'accès à l'interpréteur :

#!/bin/sh pour le Bourne Shell

#!/bin/csh pour le C-Shell

sinon le fichier est directement exécutable, c'est le résultat d'une compilation et d'une édition de lien, ce champ indique si la pagination est autorisée (stickybit), si le code est partageable par plusieurs processus.

## Image d'un processus en mémoire



Organisation de l'image d'un processus en mémoire virtuelle<sup>1</sup>.

Le chargement de l'image du processus en mémoire se fait à la demande, au fur et à mesure des défauts de pages.

---

<sup>1</sup> Plusieurs Processus peuvent utiliser le même code, le code est alors partagé entre plusieurs processus, un seul exemplaire de la partie "text" réside en mémoire

# Génération de Processus

**1 utilisateur = plusieurs processus**

création de processus :

appels système

fork() crée un nouveau processus

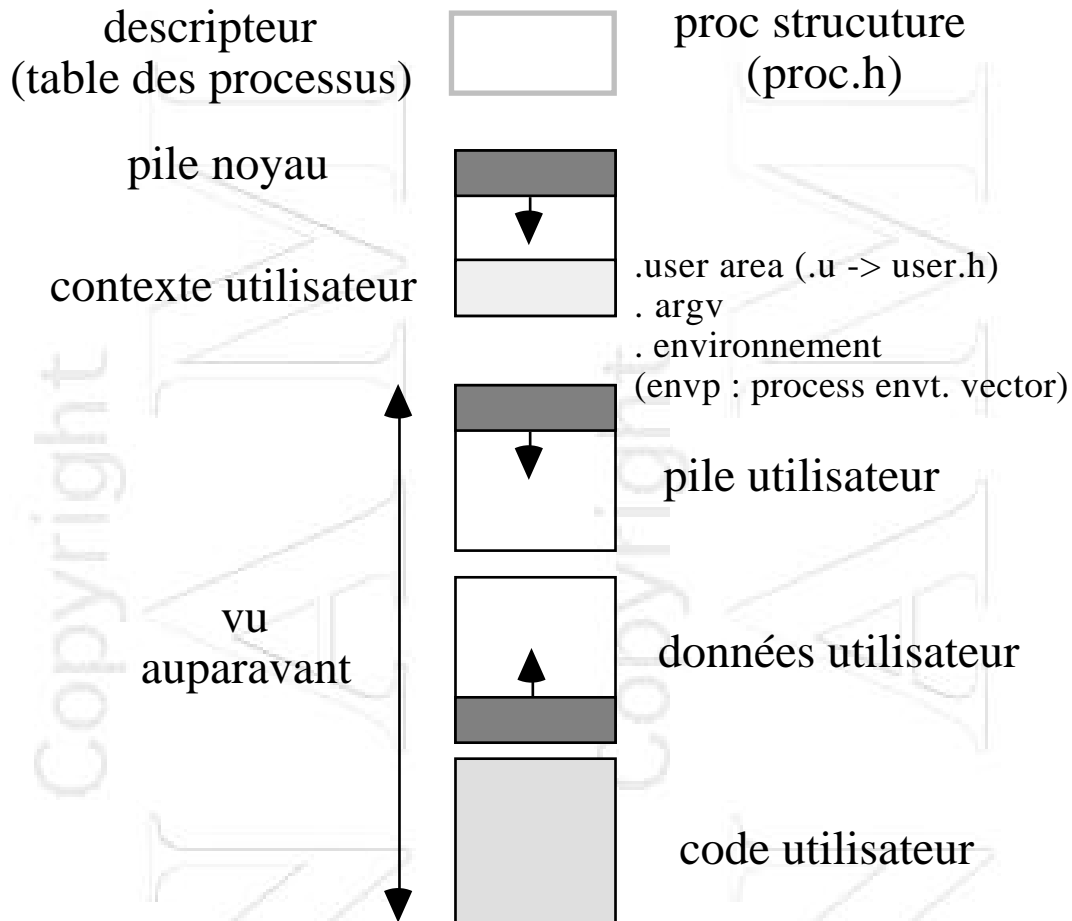
exec() charge un nouvel exécutable

**Processus :**

Unité d'exécution d'un programme  
Espace d'adressage & d'allocation de  
ressources

(désignation locale à une machine des objets systèmes, non unicité des noms sur le réseau)

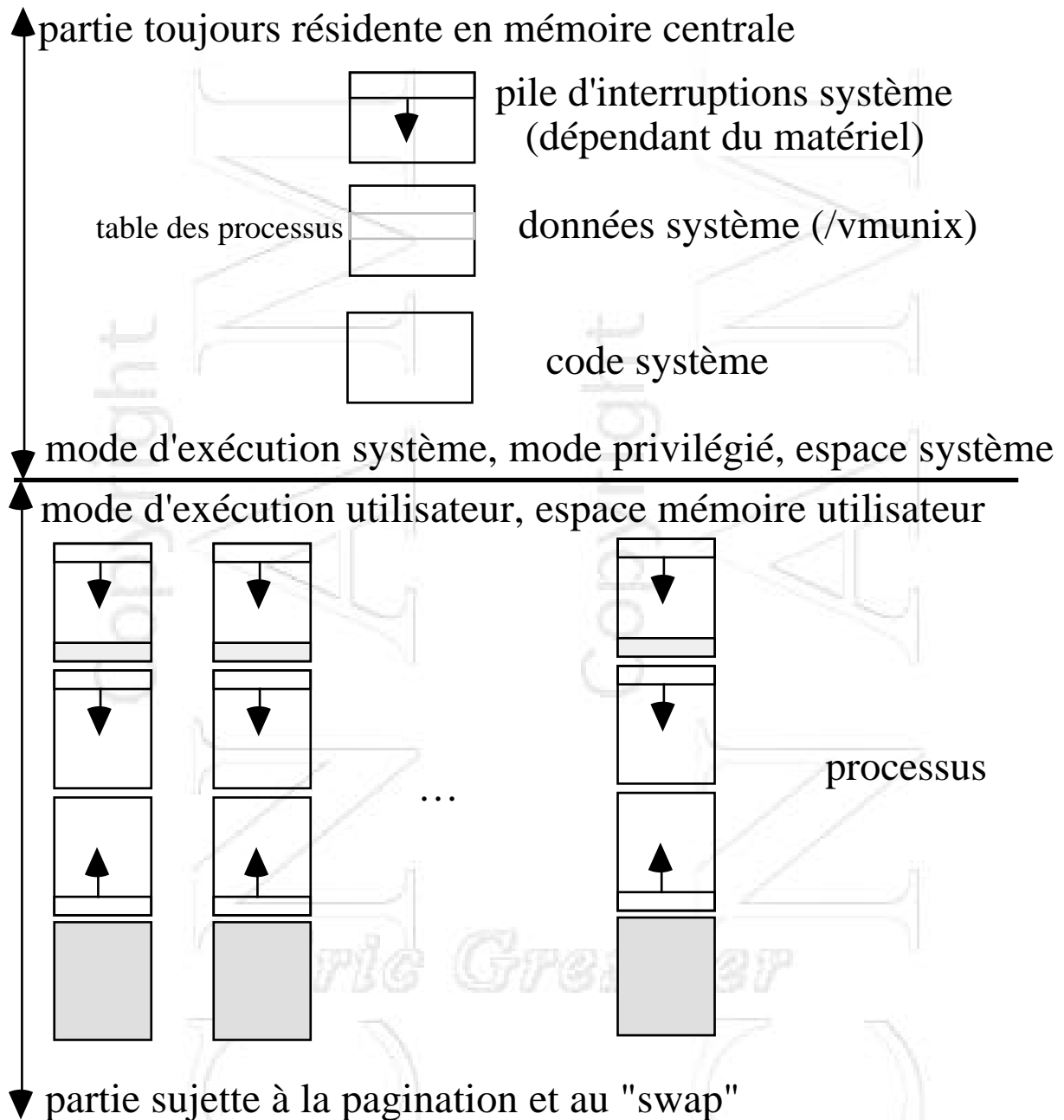
## Processus du point de vue du Système



La pile noyau est utilisée quand le processus s'exécute en mode système, un appel système provoque le passage du processus du mode d'exécution utilisateur au mode système, le processus exécute alors du code système pour son propre compte.

"user area" : ensemble d'informations sur le processus, informations nécessaires mais non vitales quand le processus n'est plus en mémoire (sur disque de pagination)

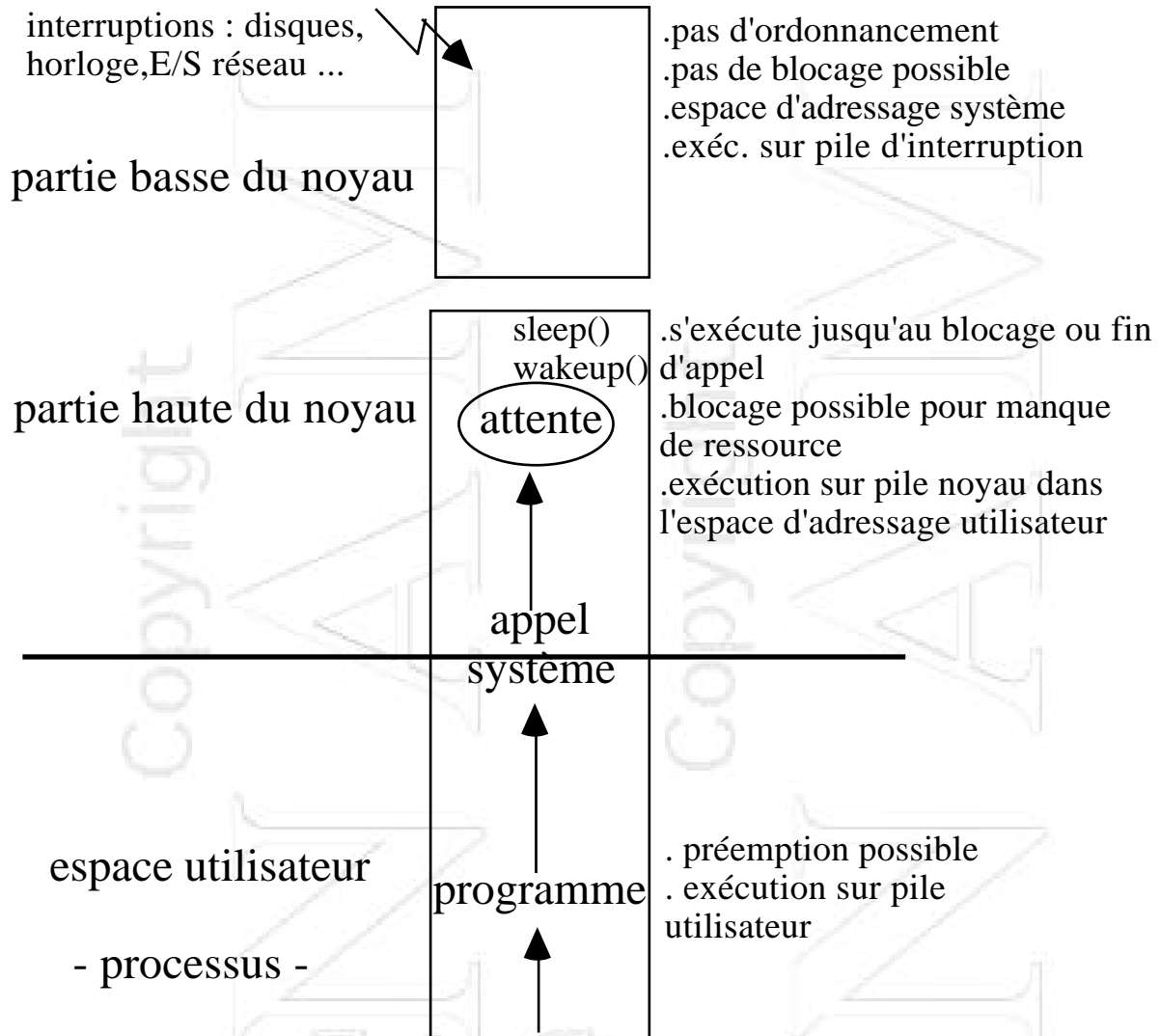
# Image de la mémoire vue par le système



pagination -> lié à la gestion de la mémoire virtuelle

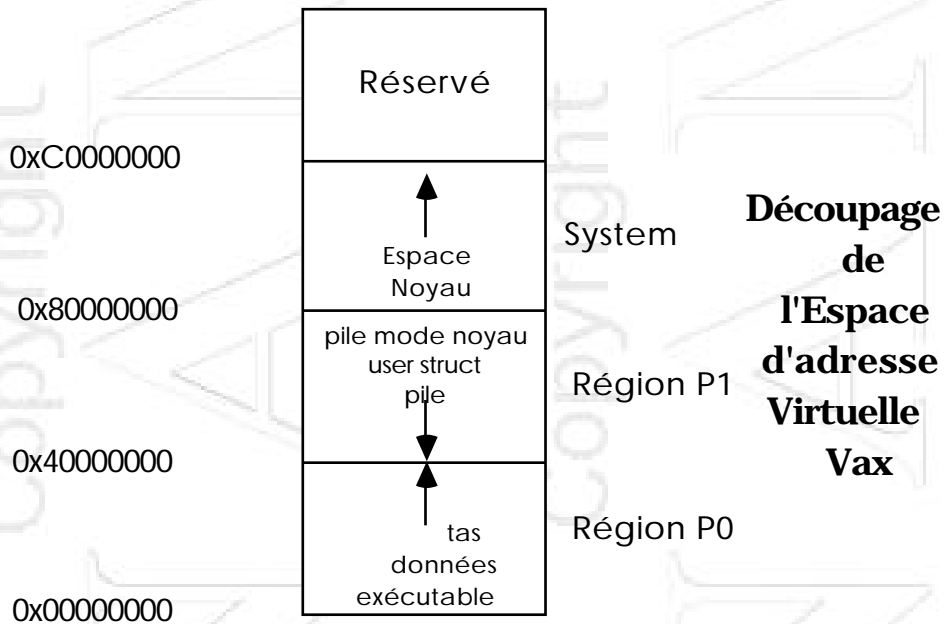
"swap" -> recopie de processus entier sur disque, concerne la gestion de ressource mémoire et processeur, et le taux d'utilisation de la machine

# Modes d'exécution



# Adressage en mémoire virtuelle

exemple Vax :



## Lecture "système" de la mémoire

**/dev/kmem** mémoire virtuelle vue comme un périphérique

**/dev/mem** mémoire physique vue comme un périphérique

- les données noyaux ont une adresse (Vax) :

0x80 000 000 + déplacement

- les données utilisateur ont une adresse (Vax):

0x00 000 000 + déplacement

**/dev/drum** mémoire de pagination/swap vue comme un périphérique

Les périphériques sont vus comme des fichiers, on peut y accéder de la même façon en respectant les règles d'accès.

exemple : accès en lecture à la mémoire virtuelle

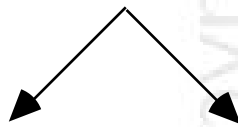
```
fd_kmem = open ("/dev/kmem",0)
```



Pour savoir si une variable est dans l'espace noyau :

son adresse est sur 32 bits (long int), si loc est l'adresse de la variable examinée :

loc "ET" 0x 8000 0000

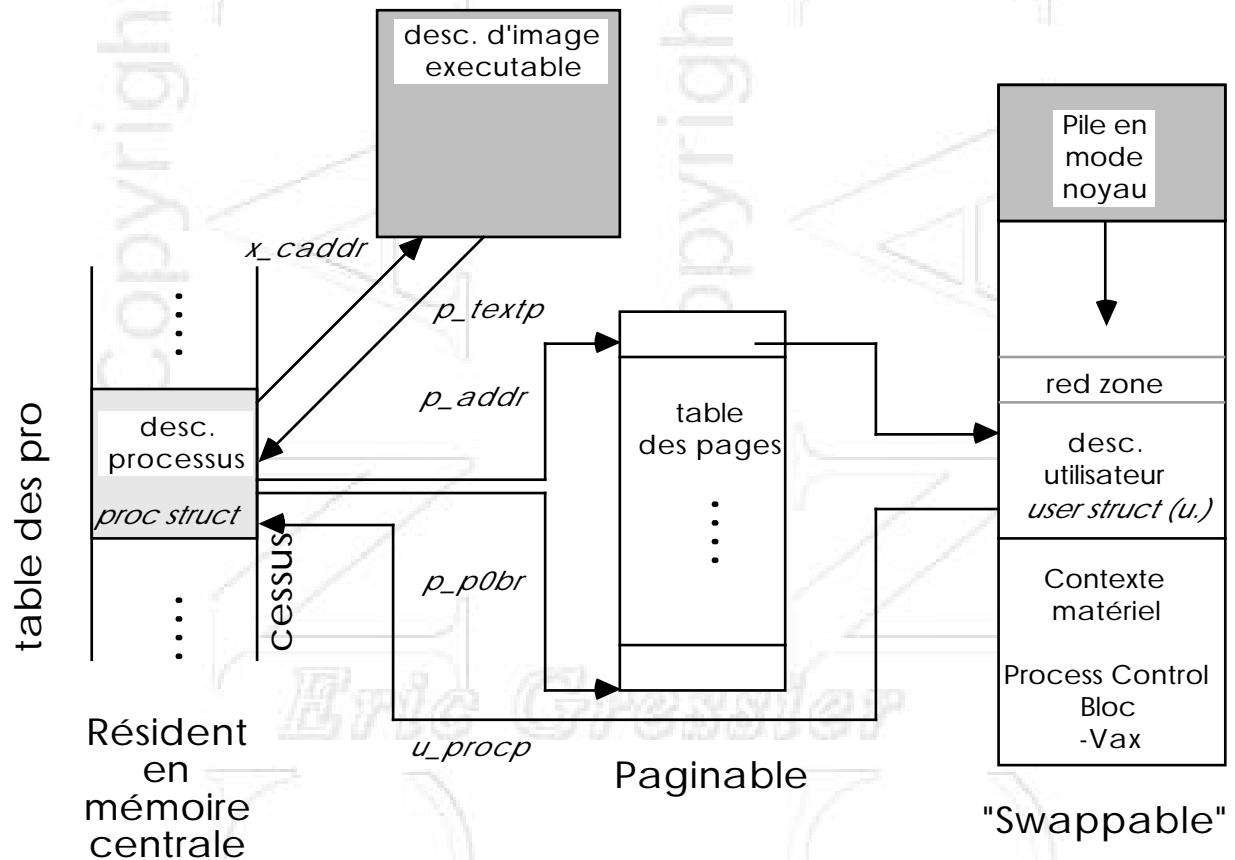


vrai : variable dans  
l'espace noyau

faux : variable dans  
l'espace utilisateur

# Processus en mémoire vive

**processus en mémoire** : un ensemble de structures de données



# descripteur de processus - proc structure proc.h -

## **. infos d'ordonnancement**

- > p\_pri           => priorité courante du processus (petit nombre = prioritaire)
- > p\_usrpri       => priorité du processus en mode utilisateur, calculé à partir  
                  de p\_nice et p\_cpu
- > p\_nice          => priorité proposée par l'utilisateur pour le processus
- > p\_cpu           => priorité qui tient compte de l'utilisation réelle du  
                  processeur par le processus
- > p\_slptime       => temps passé par le processus dans l'état endormi

## **. identificateurs**

- > p\_pid           => identificateur du processus (unique sur la machine), obtenu  
                  par getpid()
- > p\_ppid          => identificateur du père créateur du processus, obtenu par  
                  getppid()
- > p\_uid           => identificateur de l'utilisateur (user identifier) du processus,  
                  obtenu par getuid()
- > p\_pgid          => identificateur du groupe dans lequel est le processus, obtenu  
                  par getgid()
- > p\_euid<sup>2</sup>       => identificateur effectif du processus, identique à p\_uid  
                  souvent sauf quand le programme dispose du bit suid (set user  
                  id) qui lui permet de prendre l'uid d'un autre utilisateur, root  
                  par exemple, peut être obtenu avec geteuid())
- > p\_egid          => idem juste avant pour le groupe, obtenu avec getegid())

## **. gestion mémoire**

- > p\_textp       => pointeur vers un descripteur de fichier exécutable
- > p\_p0pbr       => adresse de la table des pages du processus
- > p\_szpt        => taille de la table des pages
- > p\_addr        => localisation de la zone u. (user area) qd le processus est en  
                  mémoire
- > p\_swaddr      => localisation de la zone u. qd le processus est "swappé"

## **. gestion d'évènements et signaux**

- > p\_wchan       => signaux attendus par le processus : fin d'E/S, fin de défaut  
                  de page, ...
- > p\_sig
- > p\_sigignore
- > p\_sigcatch

---

<sup>2</sup> champ dépendant du système (Unix BSD, Ultrix ...)

## Accès à la table des processus à l'intérieur du noyau BSD

1. Explorer l'image disque du système pour obtenir l'adresse en mémoire virtuelle (MV) des variables cibles.

Pour la table des processus, on cherche :

`_proc` : la table elle-même

`_nproc` : le nombre maximum d'entrées dans la table

Utiliser la fonction `nlist()`

```
struct {
char                *name; /* nom de la variable */
unsigned charn_type;
char                n_other;
short               n_desc;
unsigned longn_value; /*déplact dans la MV*/
};
```

utilisation :

```
struct nlist nlst[] = {
    {"_nproc"},
    {"_proc"},
    {0}, /* obligatoire */
}
```

**`nlist("/vmunix",nlst);`**

## 2. Accéder à la mémoire virtuelle pour retirer le contenu de ces variables

```
kmem = open("/dev/kmem",0); /*au début*/
```

```
lseek(kmem,offset_var,0);
```

```
read(kmem,ptr_buff_stockage,taille);
```

**pour \_proc :**

**au préalable :**

```
struct proc    *bproc
```

```
long int      nproc
```

```
lseek(kmem,nlst[1].nvalue,0);
```

```
read(kmem,&nproc,sizeof(longint3));
```

**Accès pour obtenir le premier mot machine contenu dans la table des processus :**

```
lseek(kmem,nlst[2].nvalue,0);
```

```
read(kmem,&bproc,sizeof(longint));
```

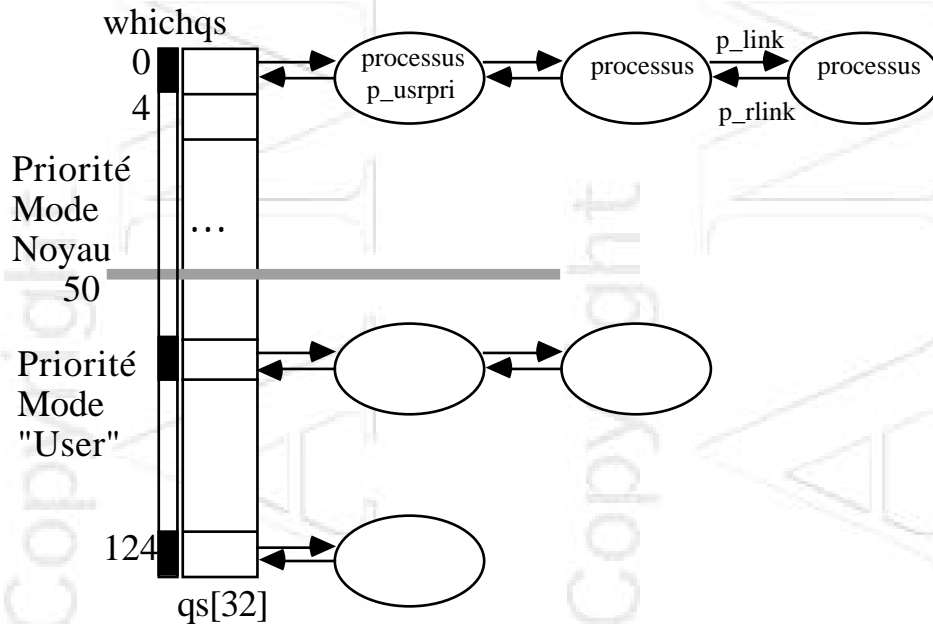
bproc contient ce premier mot !!!

---

<sup>3</sup> Un mot sur un vax fait 32 bits et est décrit par un long int

# Processus prêts

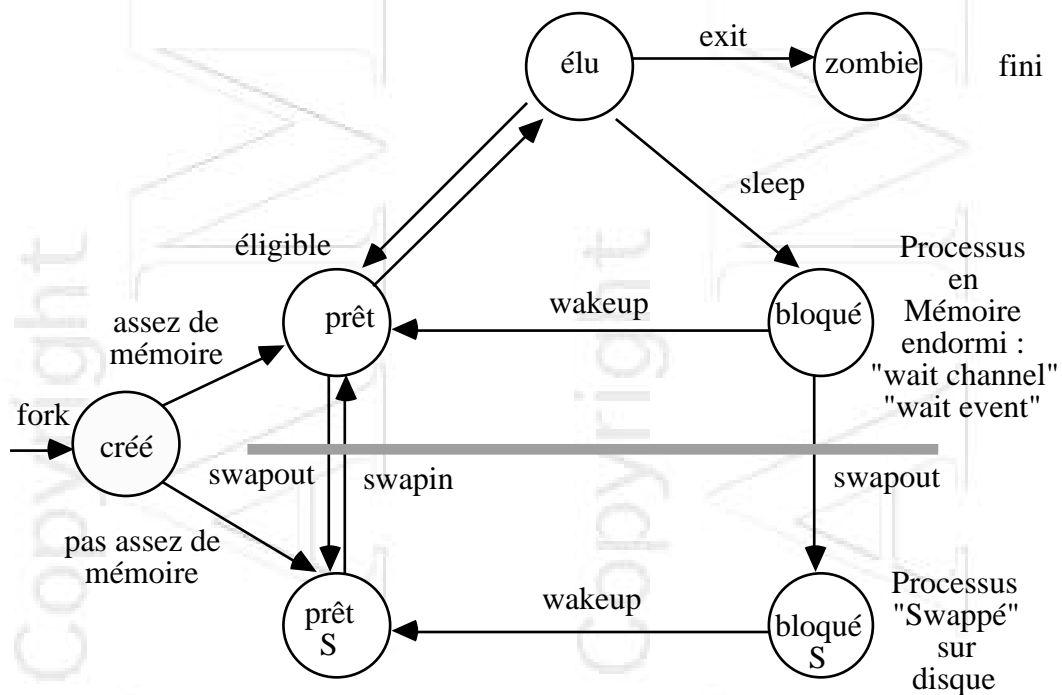
File des Processus Prêts -> Tableau de files, sauf le processus élu



La priorité va de 0 à 127 : 0 la plus forte, 127, la plus faible. Le tableau des files s'appelle `qs[ ]`. Un vecteur de bits `whichqs` (32 bits, soit un part entrée de `qs` !!!) est associé à `qs` et indique si la file correspondante est vide ou non.

`qs` est géré suivant une méthode de hachage, pour obtenir la bonne file on fait `p_pri/4`.

# Etats d'un processus



voir la commande `pstat(8)` qui permet d'imprimer une photographie des informations contenues dans les tables du système, `pstat -p` liste le contenu de la table des processus ... le résultat est assez cryptique !!!

## Ordonnancement - aspects généraux

top horloge noyau : 1 tick (10 ms souvent)

L'ordonnancement s'effectue par priorité : la file de plus forte priorité est examinée en premier, et pour une même file en tourniquet, 1 quantum de temps correspond à 10 ticks.

Quand un processus se bloque pour une ressource manquante, il ne rejoint pas la file des processus prêts.

Quand un processus épuise son quantum, il est mis en fin de la file dont il vient.

Les processus sont bougés de file en file en fonction de la valeur de leur priorité courante.

Un processus en exécution ne peut être préempté par un processus plus prioritaire qui vient de s'éveiller que s'il effectue un appel système. Sinon, le processus plus prioritaire doit attendre l'épuisement de son quantum.



## Calcul de priorité (1)

Calcul de la priorité fondé sur :

$p\_nice$  priorité demandée par l'utilisateur<sup>4</sup>

$p\_cpu$  indicateur d'utilisation du processeur par le processus

évaluée tous les 4 ticks en cours d'exécution (processus élu) par :

$$p\_usrpri = PUSER^5 + \text{Entier}(p\_cpu/4) + 2 * p\_nice \quad (1)$$

L'utilisation du processeur par le processus élu est calculée à chaque tick :

$$p\_cpu = p\_cpu + 1$$

Chaque seconde  $p\_cpu$  est ajusté par le calcul:

$$p\_cpu = (2*n)/(2*n + 1) + p\_nice$$

où  $n$  est la longueur moyenne de la file des processus prêts échantillonnée sur 1 minute.

La priorité d'un processus décroît avec le temps passé en exécution.

---

<sup>4</sup> (-20, processus "booster" par l'administrateur, 0 normal, +20 ralenti par l'utilisateur)

<sup>5</sup> PUSER vaut 50 en général

## Calcul de priorité (2)

Quand le processus est resté endormi plus d'1 s, sa priorité est recalculée à son réveil par :

$$p_{cpu} = [\text{Entier}((2*n)/(2*n+1))] p_{slptime} \quad p_{cpu}$$

$p_{slptime}$  est la durée d'endormissement, mis à zéro au moment de la mise en sommeil, puis incrémenté de un à chaque seconde écoulée.

Puis il applique la formule (1) pour calculer la priorité d'ordonnancement.

## Calcul de priorité (3)

### Priorités en mode noyau :

Non-interruptibles : 0 à  $PZERO_6$

- swapper (priorité 0)
- attente E/S disque
- attente infos sur fichier (infos de contrôle)

Interruptibles :  $PZERO+1$  à  $PUSER_7-1$

- attente de ressources
- attente sur verrou de ressources
- attente d'évènement

### Priorités en mode utilisateur :

$PUSER$  à 127

$p\_usrpri$  est toujours ramené à une valeur 127

---

<sup>6</sup> 25 habituellement

<sup>7</sup> 50 habituellement

## **Ce qu'il faut retenir !!!**

Plus on s'exécute, plus on consomme de temps CPU, et plus la priorité du processus baisse.

**qqch à faire ... lancez un processus qui  
dure peu de temps**

# Commande ps

USER	PID	%CPU	%MEM	SZ	RSS	TT	STAT	TIME	COMMAND
gressier	5015	7.5	0.8	649	530	p3	R	0:00	ps -aux
gressier	5000	0.6	0.1	247	86	p3	S	0:00	-csh (csh)
root	4748	0.3	0.1	102	44	p5	S	0:00	telnetd
root	3592	0.2	0.1	84	37	q8	S	0:05	rlogind
root	298	0.1	0.1	145	83	?	I	23:50	/etc/inetd
root	303	0.1	0.1	244	29	?	S	29:05	/etc/rwhod
root	1	0.0	0.3	254	212	?	S	3:03	init
root	4355	0.0	0.1	102	43	qb	I	0:00	telnetd
root	348	0.0	0.1	134	47	?	S	0:37	- std.9600 tty32 (getty)
root	208	0.0	0.1	163	50	?	I	1:29	/usr/local/cap/atis
root	291	0.0	0.1	48	33	?	S	0:19	/etc/cron
root	2	0.0	0.1	4096	0	?	D	4:46	pagedaemon
root	14693	0.0	0.0	0	0	?	Z	0:00	<exiting>
root	18500	0.0	0.0	147	0	?	IW	0:00	telnet
root	22368	0.0	0.0	0	0	?	Z	0:00	<exiting>
root	7600	0.0	0.0	13	0	?	IW	0:00	- std.9600 tty23 (lattelnet)
root	12550	0.0	0.0	340	12	?	IW	0:00	-ml0.iiijnet.or.jp: anonymous/
root	28562	0.0	0.0	340	12	?	IW	0:00	-crl.dec.com: anonymous/put_y
amaral	2686	0.0	0.8	864	555	03	T	0:11	emacs
root	4655	0.0	0.0	126	17	?	IW	0:00	- std.9600 tty04 (getty)
root	1215	0.0	0.2	368	148	?	I	0:00	rshd
root	2284	0.0	0.0	320	7	?	IW	0:00	rshd
root	8223	0.0	0.0	376	12	?	IW	0:00	-indy207.cs.york.ac.uk: anony
donzeau	29702	0.0	1.1	3763	724	01	T	0:00	latex exam_juin94.tex

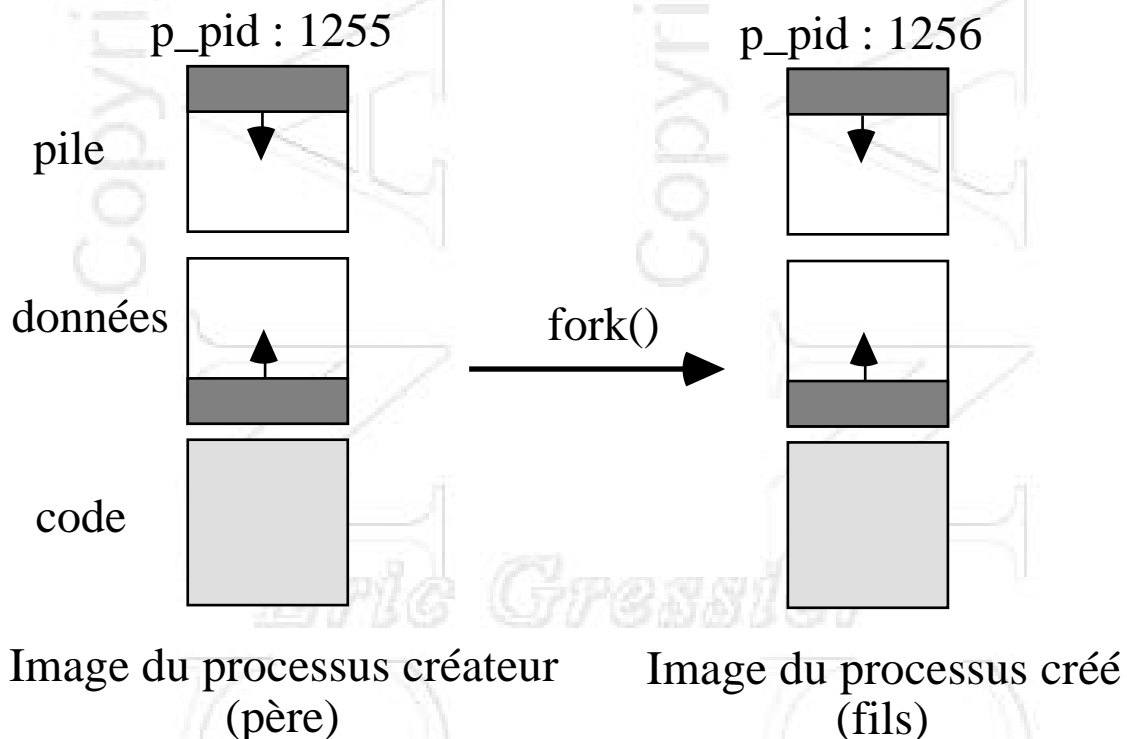
## état d'un processus sur une séquence de 5 lettres RWNVA:

- La première lettre donne des indications sur le processus en cours d'exécution :  
 "R" pour "running", "T" pour "stopped", "P" pour en attente de page, "D" pour attente d'E/S disque, "S" pour "sleeping" (moins de 20 secondes), "I" pour "ic" (endormi plus de 20 secondes).
- La seconde lettre indique si le processus est mis sur disque (swapped out) :  
 "W" pour swappé, "Z" pour tué(zombie, pas encore détruit), "<espace>" le processus en mémoire, ">" le processus est résident en mémoire et a dépassé son quota mémc centrale.
- La troisième lettre indique si le processus s'exécute avec une priorité modifiée la commande nice :  
 "N" sa priorité a été réduite, "<" sa priorité a été augmentée, "<espace>" cas norm
- La quatrième lettre indique si le processus est sujet à un traitement spécial mémoire virtuelle, le plus courant : "<espace>" représente le cas normal VA\_NORM.
- La cinquième dépend des capacités matérielle du processeur et de la façon dont processus les utilise.

## Primitive fork

```
int fork( )
```

crée un nouveau processus qui est un **clone du père au moment de la création**, le fils hérite de l'environnement du père : même vue du code, même vue des données, mêmes fichiers ouverts, même environnement (variables PATH, TERM, ...)



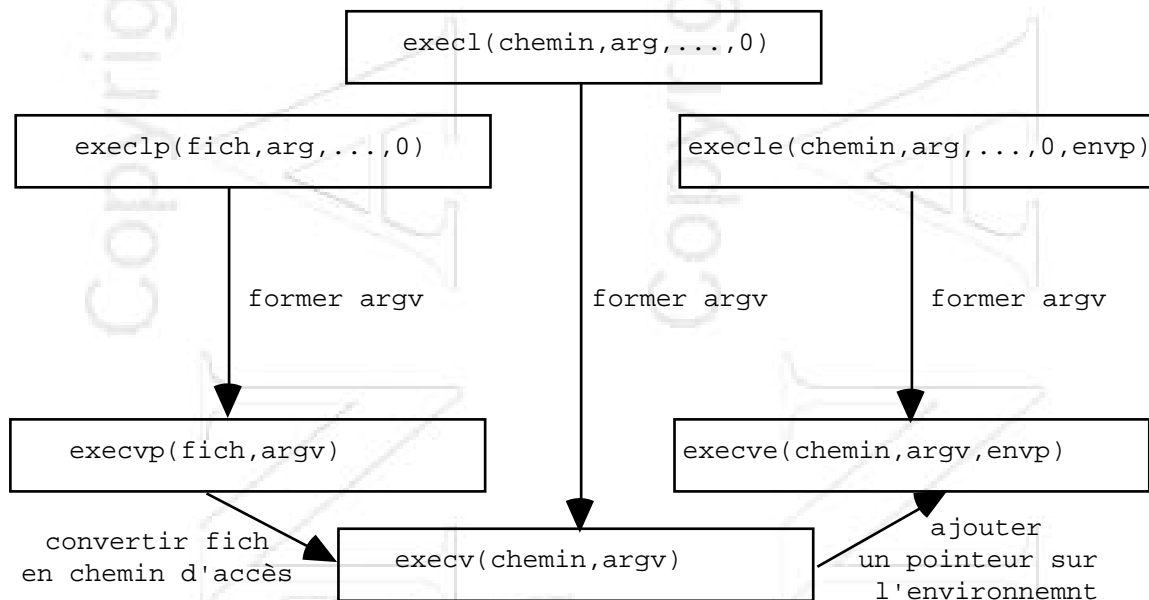
En réalité le code n'est pas dupliqué et les données le sont le plus tard possible, technique du **copy-on-write** ... parfois on fait un fork pour changer son image exécutable immédiatement après, dans ce cas, on utilise un **vfork()**.

```
main ()  
  
{  
    id = fork();  
    if ( id == 0 ) {  
        printf("fils : %d\n", getpid());  
        /*processus fils*/  
        exit(0);  
    } else {  
        printf("père : %d\n", getpid());  
        /*processus père*/  
        printf("fils : %d\n", id);  
        exit(0);  
    }  
}
```

## Primitives exec

L'utilisation d'une primitive de type exec permet de changer l'image d'un processus en mémoire. On peut ainsi lancer un nouveau programme.

Après un exec, le contrôle de l'exécution est donné au programme lancé.



Le programme hérite d'un certain nombre de paramètres du processus qui l'héberge : n° de processus, n° de processus père, l'uid utilisateur réel... le numéro de processus effectif peut changer si le programme lancé a un bit suid.



## Primitive `exit`

Quand il se termine un processus fait `exit(status)`.

Cet appel système a pour effet de passer une information d'état, "status", au système.

Cet état de sortie est récupérable par le processus père à l'aide de l'appel `wait`.

**`exit()`** provoque un vidage des E/S en cours non effectuées (action flush) avant de terminer le processus, c'est un appel standard de la librairie C.

**`_exit()`** effectue la terminaison directement, c'est un appel système.

## Primitive wait

```
int wait (int *status);
```

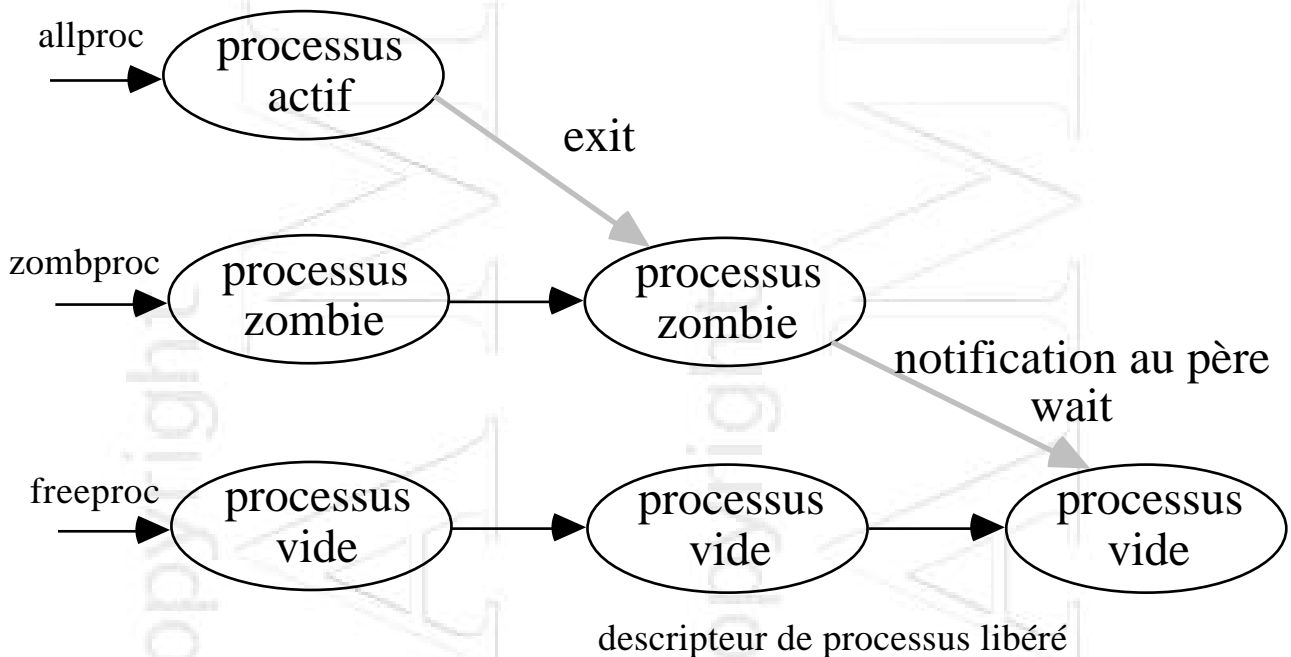
Un processus peut récupérer des informations sur un fils qui vient de se terminer grâce à `wait`. Il récupère la variable "status" de terminaison, et/ou d'autres données sur le mode de terminaison du processus.

autres variantes de `wait` :

- . `wait3` qui n'existe que pour Unix 4.3 BSD, ne bloque pas le processus qui effectue `wait3` alors que `wait` bloque "éternellement" le processus demandeur.

- . `waitpid` qui n'existe que pour Unix 4.3 BSD, permet d'attendre la fin d'un processus fils particulier

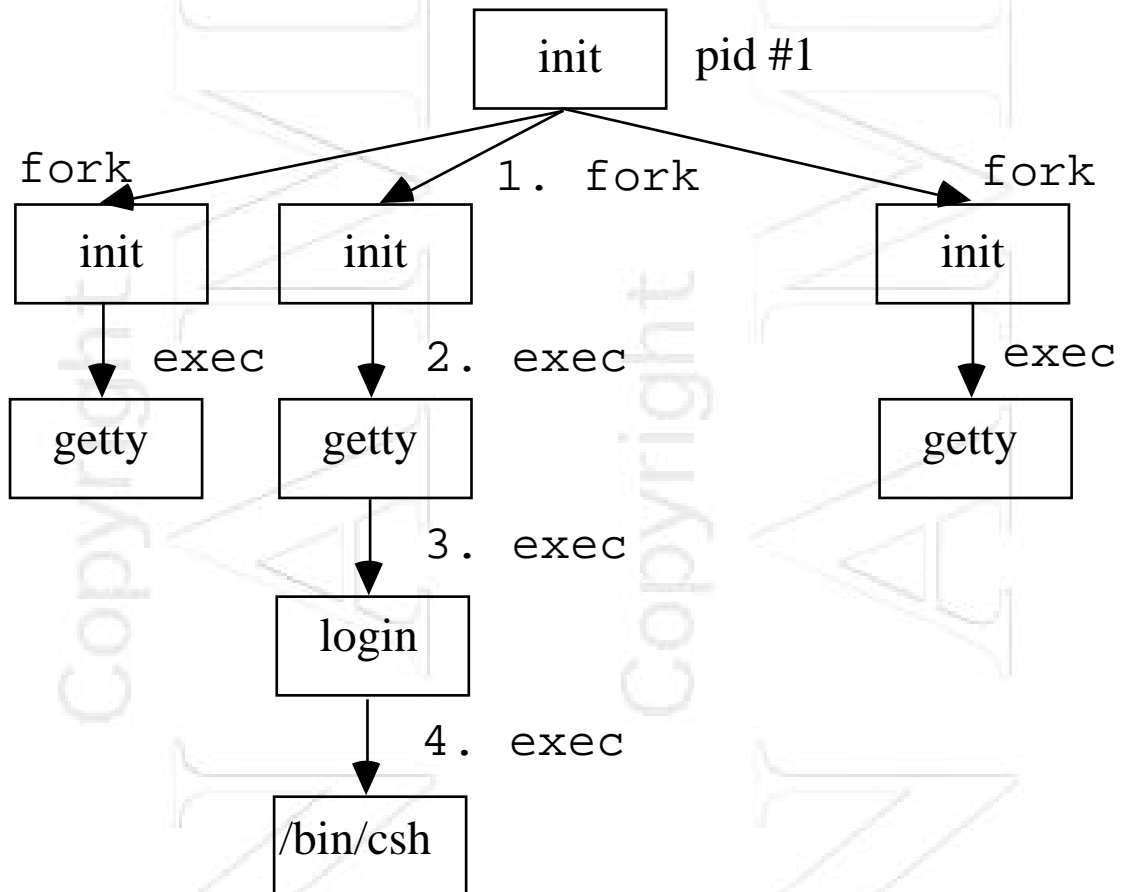
## Terminaison d'un processus



Si le père est terminé avant le fils, l'id du processus père n'a plus de sens ... Unix trouve tous les processus orphelins, leur père devient "init" qui ne se termine jamais.

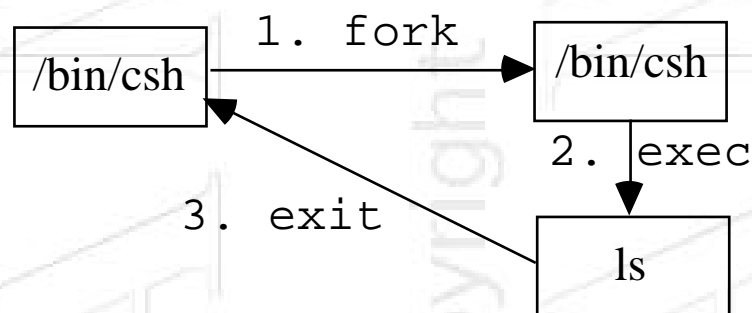
Un processus ne sait jamais quand son père a pu se terminer ... seulement à sa propre terminaison.

# Lancement d'un shell utilisateur

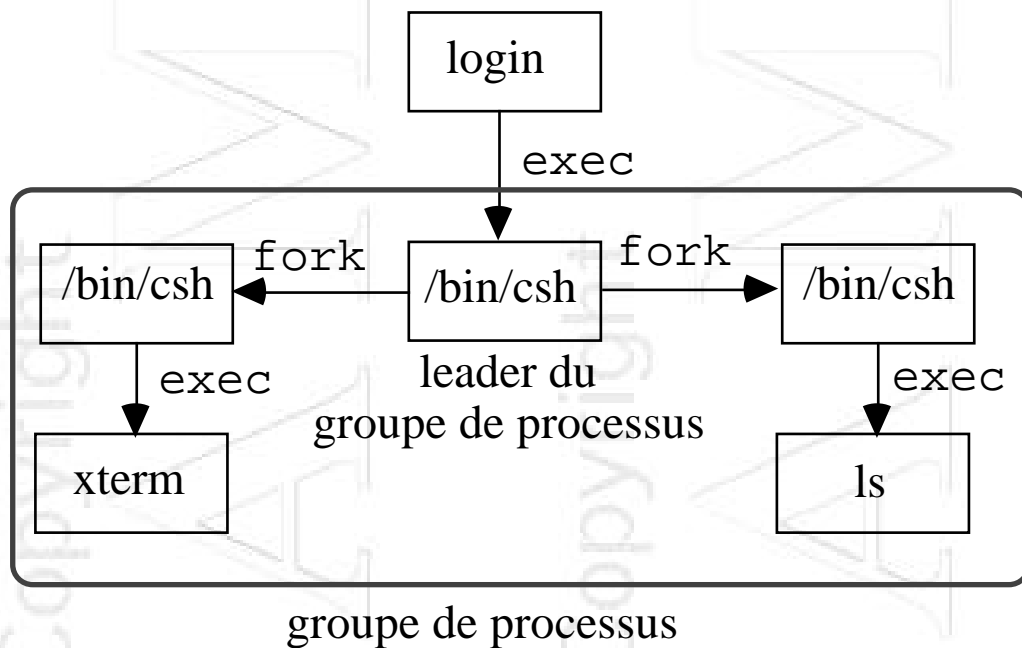


## Lancement d'une commande

Résultat du point de vue processus de la commande ls :



## Groupe de processus



Le leader du groupe de processus a son `pid` égal à son `pgid`. Un processus fils crée un nouveau groupe en se retirant du groupe auquel il appartient par la primitive `setpgrp()`.

En Bourne shell, quand vous lancez une commande, un fils est créé, il appartient au même groupe de processus que son père. Dès que vous quittez votre session on a à propos du processus qui se termine :

id de processus = id de groupe de processus = id de terminal attaché au groupe<sup>8</sup>

Le signal SIGHUP est envoyé à tous les processus qui ont le même id de groupe de processus. Tous les processus fils, petits fils, ... dans le groupe sont tués automatiquement.

En Cshell, chaque commande lancée en tâche de fond (avec & au bout) crée son propre groupe, et devient donc son propre "group leader". Par conséquent, la fin de la session utilisateur ne peut tuer les commandes lancées en tâches de fond.

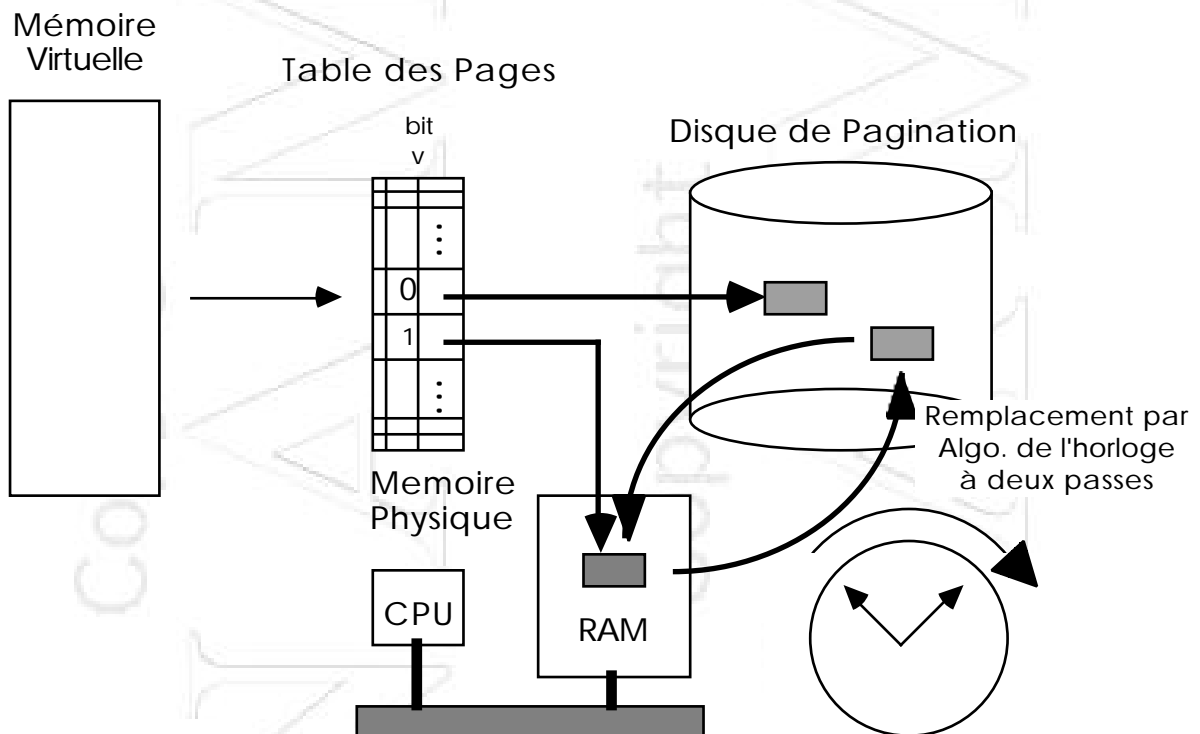
---

<sup>8</sup> Pour certains shells, l'identificateur de terminal attaché à un groupe de processus est initialisé avec l'identificateur de groupe de processus du processus shell activé dès la fin de la phase de login. En général, cette situation n'est vraie que pour les processus actifs en session (foreground), pas pour les autres, ceux lancés en arrière plan (background).

## 3. Gestion Mémoire



# relation mémoire centrale et disque de pagination



-> mécanisme de “va et vient”, superposition de l’espace “swap” avec l’espace de pagination

## commande vmstat

Statistiques sur la mémoire virtuelle, les processus, les disques, l'activité du processeur...

option -f pour avoir le nombre de fork et de vfork, option -v pour avoir des infos plus détaillées sur la mémoire, -S pour avoir des infos sur les pages soumises au va et vient (swap)

Sans options, la commande vmstat donne un résumé :

procs			faults			cpu			memory			page			disk				
r	b	w	in	sy	cs	us	sy	id	avm	fre	re	at	pi	po	fr	de	sr	s0	s1
0	0	0	98	571	75	16	12	72	21k	41k	2	4	4	1	1	0	0	0	0

procs : information sur les processus dans différents états.

- r (run queue) processus prêts
- b bloqués en attente de ressources (E/S, demandes de pages)
- w prêts ou endormis (< 20 seconds) mais swappés

faults : taux d'interruptions/d'appels système (trap) par seconde, la moyenne est calculée sur les 5 dernières secondes.

- in interruption due à un contrôleur (sauf horloge) par seconde
- sy appels systèmes par seconde
- cs taux de changement de contexte processeur ( nb chgt par seconde)

cpu : répartition de l'utilisation du processeur en pourcentage

- us temps utilisateur pour les processus de priorité normale et basse
- sy proportion utilisée par le système
- id cpu inutilisé - libre

La somme des 3 fait 100% !?!

memory: informations sur l'utilisation de la mémoire virtuelle et de la mémoire réelle, les pages virtuelles sont considérées actives si elles appartiennent à des processus qui sont exécutables ou qui se sont exécutés depuis moins de 20 secondes. Les pages sont indiquées en unité de 1 Ko, le suffixe k précise qu'il faut multiplier le chiffre affiché par 1000, et le suffixe m par  $10^6$ .

avm pages virtuelles actives  
fre taille de la liste des pages libres

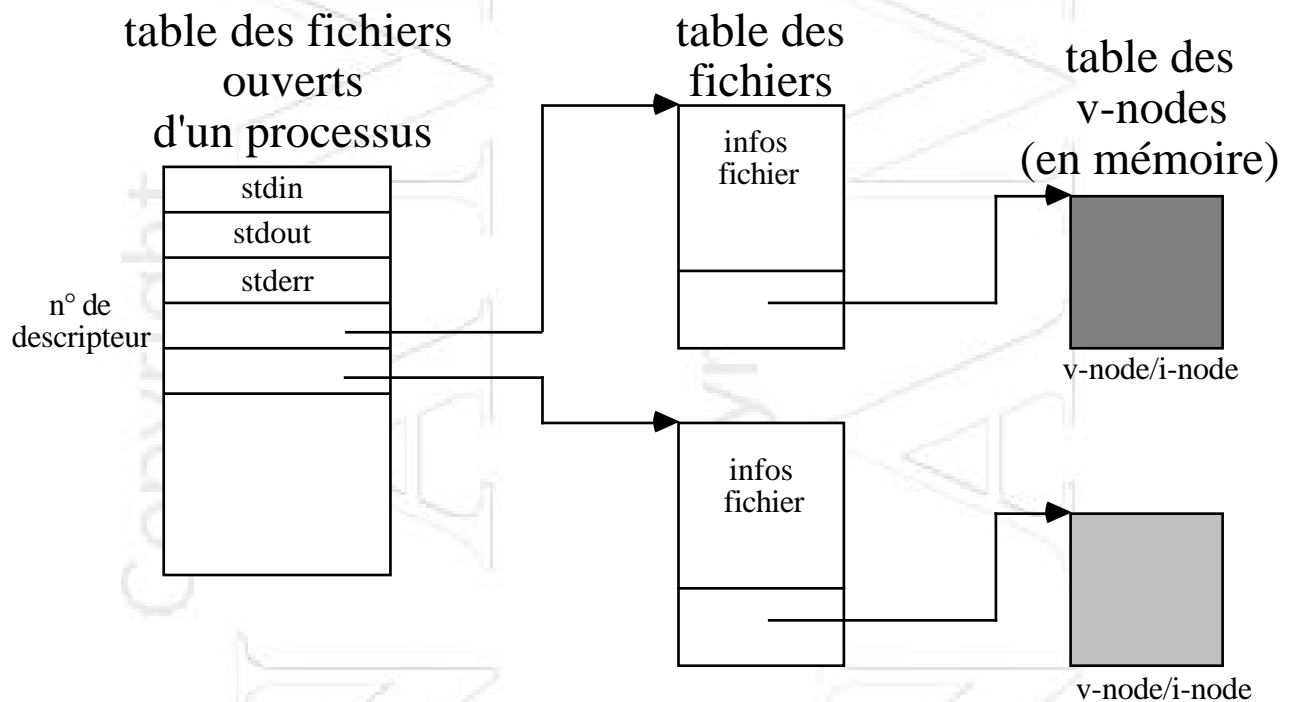
page: information les défauts de page et l'activité de pagination. La moyenne est calculée toutes les 5 secondes. L'unité est toujours 1Ko et est indépendante de la taille réel des pages de la machine.

re page "reclaims", pages référencées  
at pages "attached" ???  
pi pages "paged in"  
po pages "paged out"  
fr pages "freed", libérées par seconde  
de anticipation du manque de mémoire  
sr pages examinées par l'algorithme horloge à deux phases

disk: s0, s1 ...sn: activité de pagination ou de va et vient en secteurs transférés par seconde, ce champ dépend de la configuration du système. Habituellement l'espace swap est réparti sur plusieurs disques, on trouve la liste des périphériques qui supportent cette fonction et qui est configurée dans le système.

## 4. Fichiers et Tubes

# gestion des E/S : disque, réseau, terminal



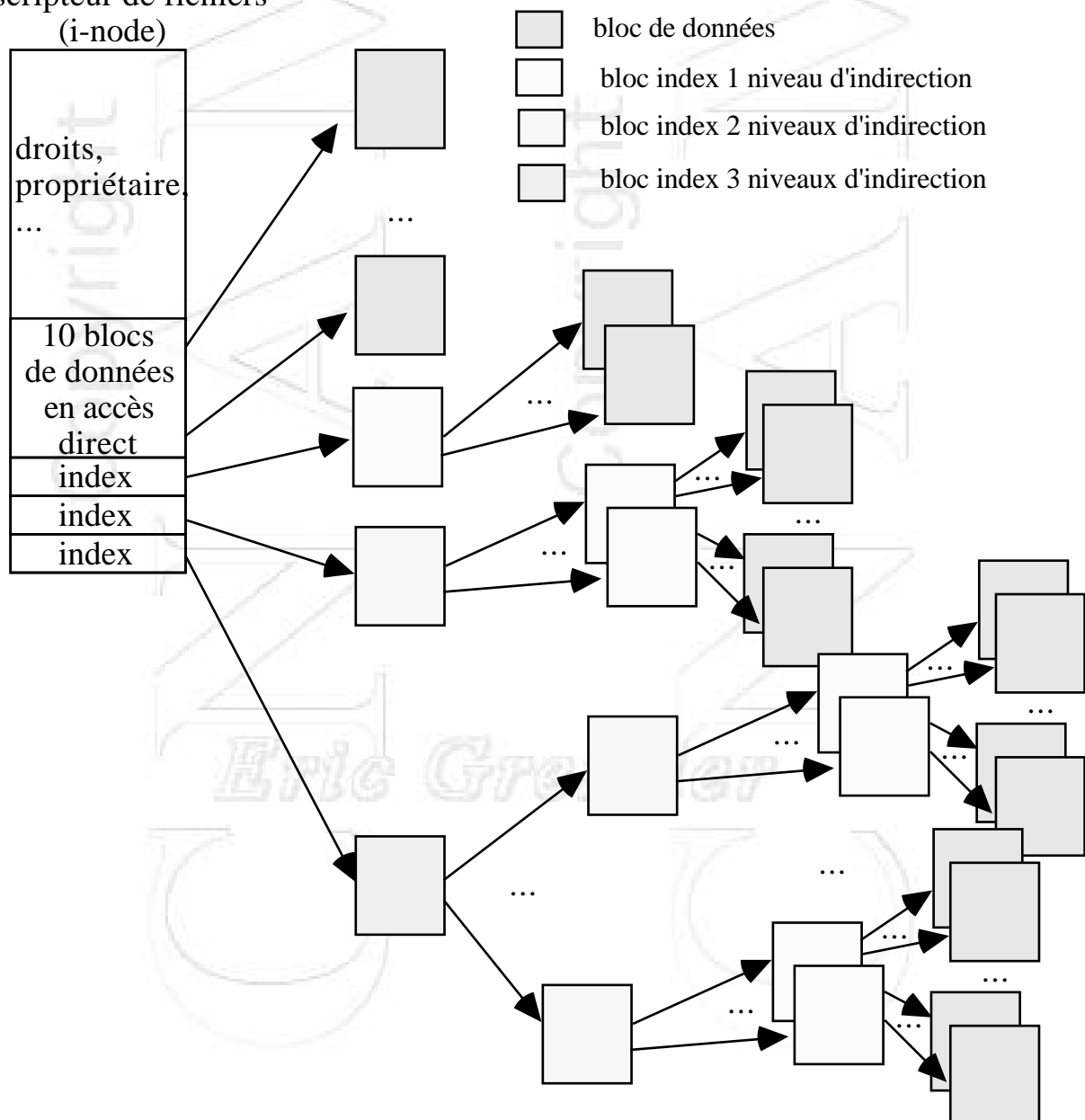
notion d'objet décrit par un descripteur et un ensemble d'opérations plutôt que par extension de l'espace d'adressage (Multics)

# Structure d'un i-node

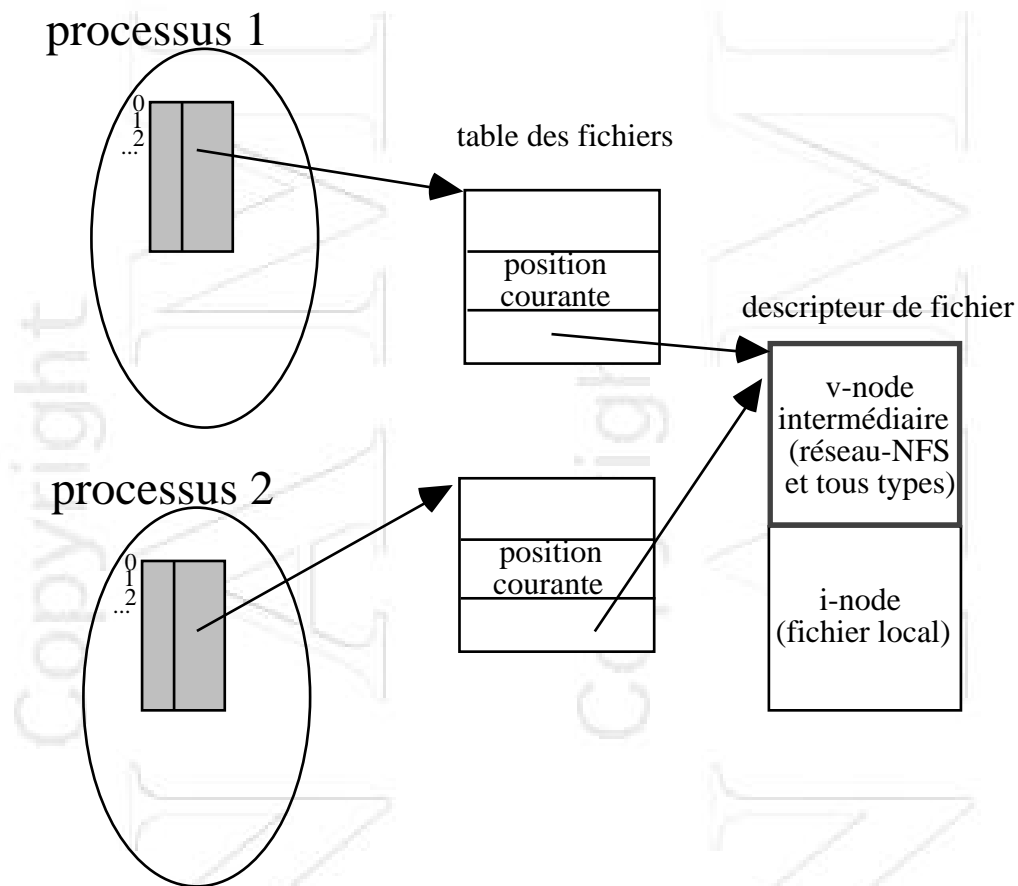
Vue utilisateur : Fichier non structuré



Vue Système :  
Descripteur de fichiers  
(i-node)

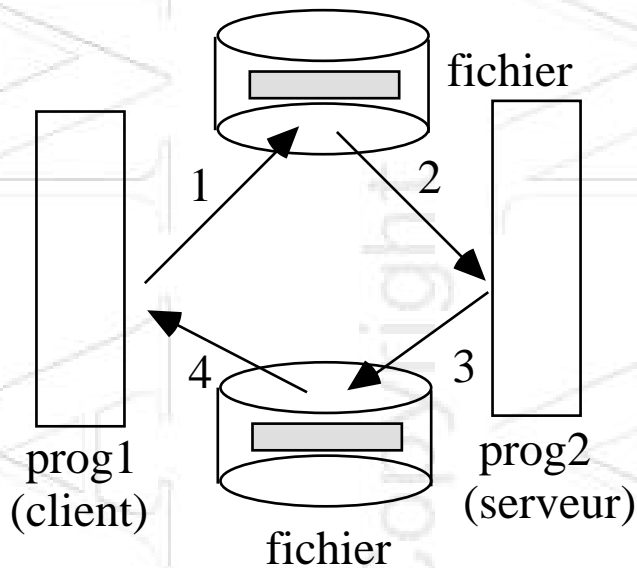


# Partage de fichier



## Communication par fichier

Les programmes peuvent communiquer par fichier :



Il peut être nécessaire de procéder à un verrouillage de la ressource fichier :

`flock()`, `lockf()`, `fcntl()`

**ATTENTION, ces mécanismes ne fonctionnent pas nécessairement à travers NFS !!!**



## Verrouillage de fichier (1)

verrouillage de tout le fichier (Unix souche BSD) :

```
flock( )
```

Le type de verrouillage doit être précisé :

LOCK_SH	verrouillage en mode partagé (opération bloquante)
LOCK_EX	verrouillage en mode exclusif (opération bloquante)
LOCK_UN	déverrouillage
LOCK_NB	demande d'opération non bloquante

Une opération est dite **bloquante** si le **demandeur est bloqué jusqu'à ce que l'appel système correspondant soit terminé** ... ce qui peut arriver ... l'appellant restant dans ce cas indéfiniment en attente... d'où l'utilisation par exemple de la combinaison LOCK\_SH | LOCK\_NB.

Un fichier peut être verrouillé "LOCK\_SH" par plusieurs processus en même temps, mais ne peut être verrouillé "LOCK\_EX" que par un seul à la fois.

## Verrouillage de fichier(2)

verrouiller une partie du fichier (Unix souche system V) :

`lockf ( )`

dans ce cas, il faut utiliser `lseek ( )` pour se positionner au début de la zone du fichier ciblée, puis spécifier dans l'appel de `lockf ( )` la longueur de la zone à verrouiller.

Le verrouillage peut s'effectuer de différentes façons, qu'il faut spécifier :

<code>F_TEST</code>	test si une zone est déjà verrouillée
<code>F_LOCK</code>	verrouiller une zone (opération bloquante)
<code>F_TLOCK</code>	test si la zone est verrouillée, verrouille sinon
<code>F_ULOCK</code>	déverrouille une zone déjà verrouillée

*Eric Gressier*

## **Primitive `mmap( )` (1)**

`mmap( )`

Permet à un processus de projeter le contenu d'un fichier déjà ouvert dans son espace d'adressage. La zone mappée s'appelle une région. Au lieu de faire des lectures et des écritures sur le fichier, le processus y accède comme si les variables qu'il contient étaient en mémoire.

Plusieurs processus peuvent "mapper" le contenu d'un même fichier et ainsi le partager de façon efficace.

Un fichier peut être soit un fichier sur disque, soit un périphérique.

`mmap( )` s'utilise pour un fichier déjà créé, il est impossible d'étendre un fichier "mappé"

suivant les implantations, il semble qu'un `munmap( )` soit nécessaire pour "démapper" une région avant de fermer le fichier par `close( )`.

**pas de client/serveur !:-)**

## Primitive `mmap( )` (2)

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

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

C'est une mise en correspondance à partir de l'adresse "pa" de l'espace d'adressage d'un processus, d'une zone de "len" octets prise à l'intérieur d'un objet de descripteur "fd" à partir de "off". La valeur de "pa" dépend du paramètre "addr", de la machine, et de la valeur de "flags".

Si l'appel système réussit, `mmap()` retourne la valeur pa en résultat. Il y a correspondance entre les régions [pa, pa+len] de l'espace d'adressage du processus et [off, off+len] de l'objet. Une mise en correspondance remplace toute mise en correspondance précédente sur la zone [pa, pa+len] .

`prot` détermine le mode d'accès de la région : "read", "write", "execute", "none" (aucun accès) ou une combinaison de ces modes

`flags` donne des informations sur la gestion des pages mappées :

<code>MAP_SHARED</code>	page partagée, les modifications seront visibles par les autres processus quand ils accèderont au fichier
<code>MAP_PRIVATE</code>	les modifications ne seront pas visibles
<code>MAP_FIXED</code>	l'adresse <code>addr</code> donnée par l'utilisateur est prise exactement, habituellement le système effectue un arrondi sur une frontière de page

## Exemple mmap( )

```
int fd, *p, lg;
fd = open("fichier",2);
p =
    mmap((caddr_t) 0,lg,
        PROT_READ|PROT_WRITE,
        MAP_SHARED,fd,0);
*p = *p +1;
close(fd);
```

## **Tube ou "pipe" - "|" du niveau shell -(1)**

`pipe()`

Un tube est un **canal unidirectionnel** qui fonctionne en mode flot d'octets. Mécanisme d'échange bien adapté à l'envoi de caractères.

La suite d'octets postée dans un tube n'est pas obligatoirement retirée en une seule fois par l'entité à l'autre bout du tube. Lors de la lecture, il peut n'y avoir qu'une partie des octets retirés, le reste est laissé pour une lecture ultérieure. Ceci amène les utilisateurs à délimiter les messages envoyés en les entrecoupant de `"\n"`.

Sous certaines conditions, une écriture ou une lecture peut provoquer la terminaison du processus qui effectue l'appel système.

Les écritures dans un tube sont atomiques normalement. Deux processus qui écrivent en même temps ne peuvent mélanger leurs données. Mais si on écrit plus d'un certain volume max (4096 octets souvent), l'écriture dans le tube n'est pas atomique, dans ce cas les données écrites par deux processus concurrents peuvent s'entrelacer.

## **Tube ou "pipe" - "|" du niveau shell -(2)**

L'utilisation d'un tube se fait entre processus de même "famille"/"descendance". On peut dire que le tube est privé.

L'usage courant est :

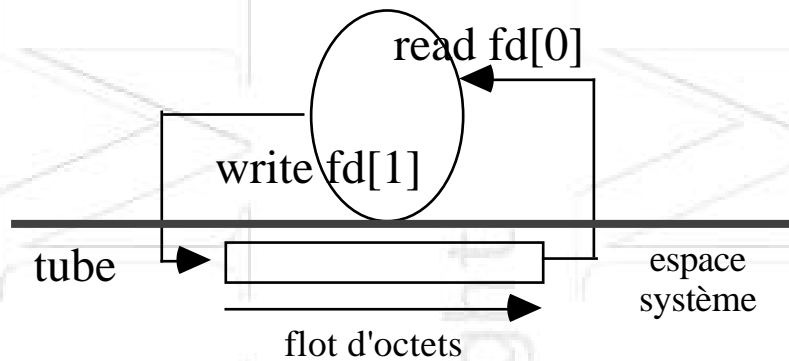
```
int pipefd[2];  
pipe(pipefd);  
write (pipefd[1], ...);  
read (pipefd[0], ...);  
...
```

se souvenir de `stdin` (0), on lit ce qui vient du clavier, et, `stdout`(1), on écrit sur l'écran.

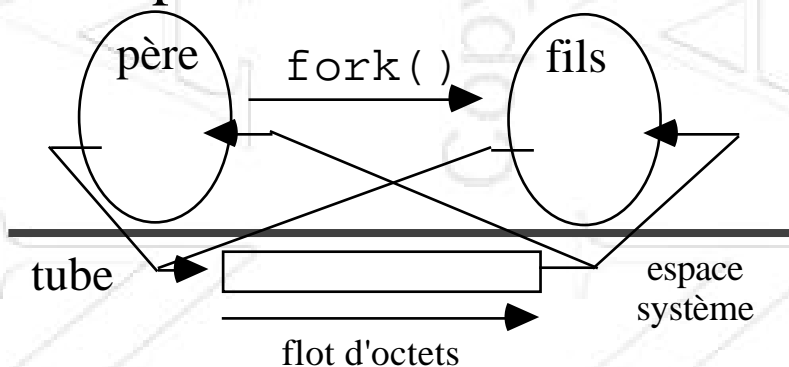
Un tube existe tant qu'un processus le référence. Quand le dernier processus référençant un tube est terminé, le tube est détruit.

# Etapes d'un échange client/serveur avec tube

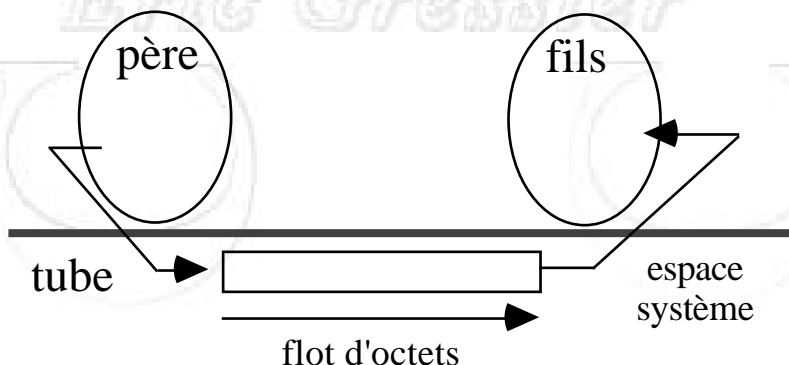
## 1. création du tube :



## 2. fork ( ) du père :



## 3. fermeture des extrémités non utilisées



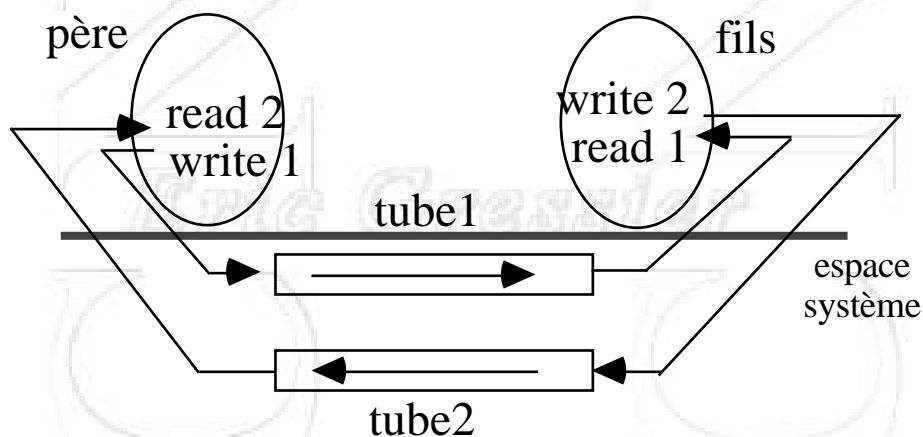
situation équivalente à celle de "ls -l | more"



## Schéma Client/Serveur avec tubes

### Etapes :

1. création de tube1 et de tube2
2. `fork()`
3. le père ferme l'entrée 0 (in) de tube1 et l'entrée 1 (out) de tube2
4. le fils ferme l'entrée 1 (out) de tube1 et l'entrée 0 (in) de tube2



## Tubes Nommés ou FIFOs

Les tubes sont privés et connus de leur seule descendance par le mécanisme d'héritage. Les tubes nommés sont publics au contraire.

Le tube nommé est créé par :

```
mknod("chemindaccès", mode d'accès)
```

ou par

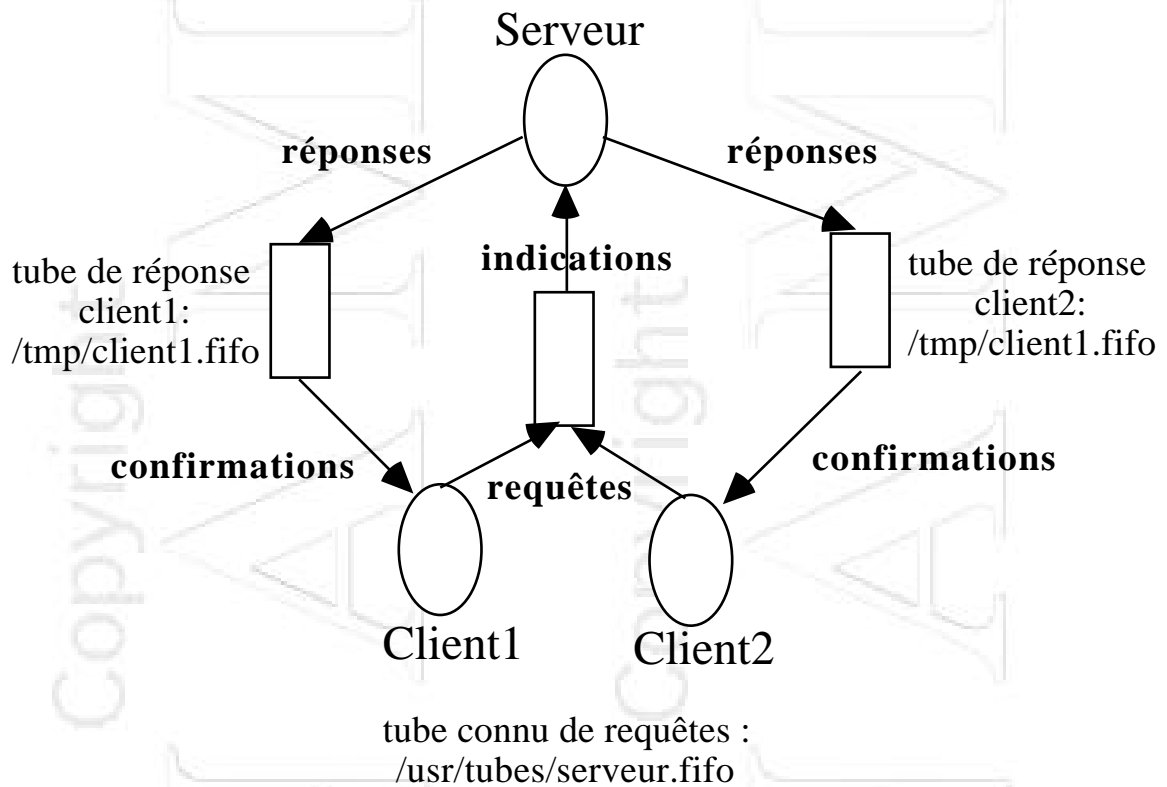
```
/etc/mknod chemindaccès p
```

C'est presque l'équivalent d'un fichier pour le système.

Le tube est connu par son chemin d'accès et accessible comme un fichier par `open()`, `read()`, `write()`, `close()`.

Les tubes nommés fonctionnent comme les tubes.

## Client/Serveur avec des tubes nommés



Ne pas oublier de détruire les tubes nommés après utilisation !!!

# Mécanismes de communication inter-processus

## IPC distants :

- sockets (Unix BSD) ou streams (Unix System V)

## IPC locaux :

- fichiers avec mécanisme de verrouillage (lock)
- tubes (pipe), tubes nommés,
- interruptions logicielles (signaux)
- sockets,
- sémaphores,
- files de messages,
- mémoire partagée,

## 5. IPC System V

## Identification des objets IPC system V

Les objets IPC system V :

Files de messages  
Sémaphores,  
Segments de mémoire partagée

Ils sont désignés par des identificateurs uniques. La première étape consiste donc à créer un identificateur.

1. On peut fabriquer un identificateur grâce à la fonction suivante :

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

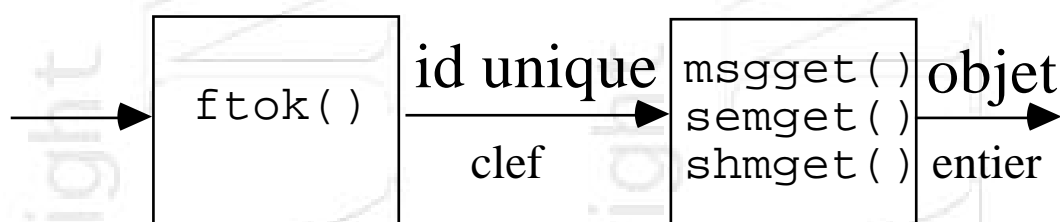
```
key_t ftok(char *chemindaccès, char proj)
```

L'identificateur rendu en résultat est garanti unique. "chemindaccès" correspond à un fichier, attention à ce qu'il ne soit pas détruit de façon inopportune... sinon, la fonction `ftok()` ne pourrait retourner l'identificateur qui sert à tous les processus pour repérer l'objet IPC.

2. On se le fixe soi même

## Création d'objets IPC System V

Il est attaché à un objet IPC un certain nombre d'informations : l'id utilisateur du propriétaire, l'id du groupe du propriétaire, l'id utilisateur du créateur, l'id du groupe du créateur, le mode d'accès, son identificateur.



Mode de création :

IPC\_PRIVATE

création de l'objet demandé et  
association de celui-ci à  
l'identificateur fourni

IPC\_CREAT

création si l'objet n'existe pas déjà,  
sinon aucune erreur

IPC\_CREAT | IPC\_EXCL

création si l'objet n'existe pas déjà,  
sinon erreur

aucun mode précisé erreur si l'objet n'existe pas

Pas comparable à un fichier dans son mode de gestion et d'héritage. L'identificateur d'objet se passe

## Files de messages

L'objet "Message Queue" fonctionne suivant le modèle d'une file donc en FIFO, la politique FIFO peut être appliquée à l'ensemble des messages dans la file ou par seulement en fonction du type de message.

création : `msgget ( )`

Lors de la création, on spécifie les droits d'accès : read/write pour propriétaire/groupe/autres en combinant avec `IPC_XXX`.

envoi de message : `msgsnd ( )`

réception de message : `msgrcv ( )`

Les messages peuvent être retirés en fonction de leur type.

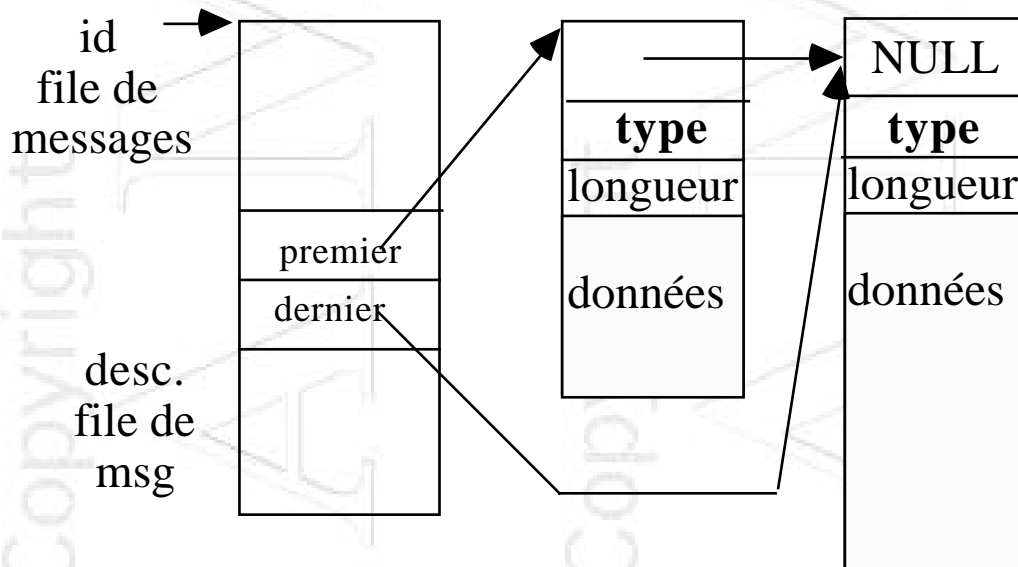
opérations de contrôle : `msgctl ( )`

Permet en particulier de détruire une file de messages, sinon il faut utiliser la commande `ipcrm`.

La commande `ipcrm` est applicable à tout objet IPC system V.



# Objet File de Messages



# Sémaphores

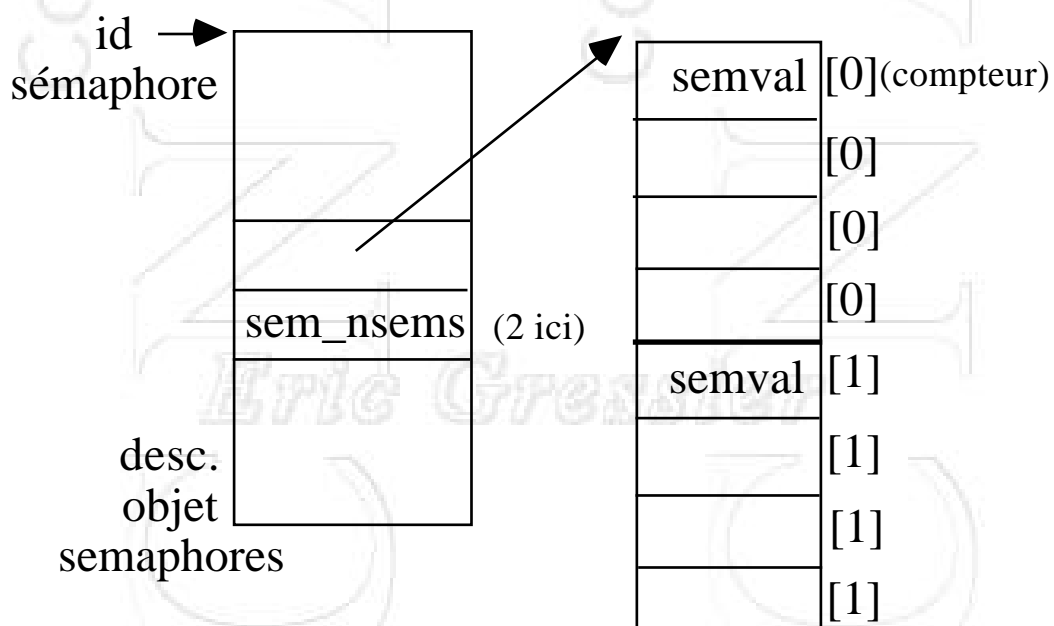
Les objets sémaphores permettent de gérer un groupe de sémaphores par objet.

création : `semget ( )`

manipulation du sémaphore : `semop ( )`

gestion du sémaphore : `semctl ( )`

Objet sémaphore :



Quand plusieurs processus attendent le relachement d'une ressource, on a aucun moyen de déterminer à l'avance celui qui l'obtiendra. En particulier, pas d'ordre FIFO d'attente.

## Segments de Mémoire partagée

Communication par variable partagée à travers l'espace d'adressage des processus.

création : `shmget ( )`

Le segment de mémoire partagée est créé mais non accessible. Là encore, il faut spécifier le mode d'accès.

attachement à l'espace d'adressage d'un processus : `shmat ( )`

Le segment est inclus dans l'espace d'adressage du processus demandeur à l'adresse indiquée suivant le cas par `*shsmaddr`.

détachement du segment de mémoire partagée de l'espace d'adressage d'un processus : `shmdt ( )`

opérations de gestion du segment de mémoire partagée : `shmctl ( )`

## Utilisation de segments de mémoire partagée

En général on fait du client/serveur à travers un segment de mémoire partagée avec une signalisation par sémaphore.

