université
PARIS-SACLAY

UFR Sciences et Technologies
Master 1 E3A

EC932 : Informatique Industrielle : Programmation Système (Unix)

Malik Mallem - Bâtiment Pelvoux

UFR S&T, ,40 Rue du Pelvoux F-91020 Evry Cedex Tél :+33-1-69 47 75 15
Fax :+33-1-69 47 75 99

Programmation système sous UNIX

Objectif : Approfondir la connaissance du système d'exploitation UNIX en étudiant son fonctionnement et en manipulant ses fonctions systèmes

Contenu :

1 Présentation d'UNIX 2
2. Noyau Unix 4
3. Fichiers 7
4 Processus 26
5. Communication inter processus 53

Bibliographie :

Maurice J. Bach "Conception du système UNIX" Masson paris 1989

Jean-Marie Rifflet "La programmation sous UNIX" Ediscience International 1998

1 Présentation d'UNIX

1.1 Historique

UNIX créé au Laboratoire BELL, USA, en 1969 .
Destiné à la gestion d'un mini-ordinateur pour une petite équipe de programmeurs.
Intéresse rapidement de nombreuse universités puis des constructeurs.
Deux principales familles de systèmes UNIX (1983):
Berkeley(BSD) et **System V** de Bell.
De nombreux efforts de normalisation : norme System V , **POSIX**(1988), OSF
De nombreuses versions d'UNIX sont donc apparues :
ULTRIX (BSD) puis OSF sur DIGITAL, IRIX(System V) sur Silicon Graphics,
LINUX(POSIX) sur PC, et bien d'autres.

1.2 Les Concepts Fondamentaux

Système multi-utilisateurs et multi-tâches, indépendant de toute architecture matérielle/constructeur.

Permet la répartition des ressources (mémoire, processeurs, espace disque, imprimantes, programmes et utilitaires) entre les utilisateurs et les tâches.

Chaque utilisateur peut exécuter plusieurs programmes simultanément.

Fournit des primitives pour construire des applications complexes à partir d'autres plus simples, avec une interface graphique puissante- X-Windows (MIT) , MOTIF, ...

Il est possible de rediriger les entrées et sorties des processus.

Un mécanisme de communication par tubes permet de synchroniser des processus et de leur faire échanger des informations.

Un système UNIX est administré par un super-utilisateur ("superuser").

2. Noyau Unix

Le système UNIX est avant tout une collection d'appels système sur les fichiers, processus, entrées/sorties, communications inter-processus, communications réseaux, etc.

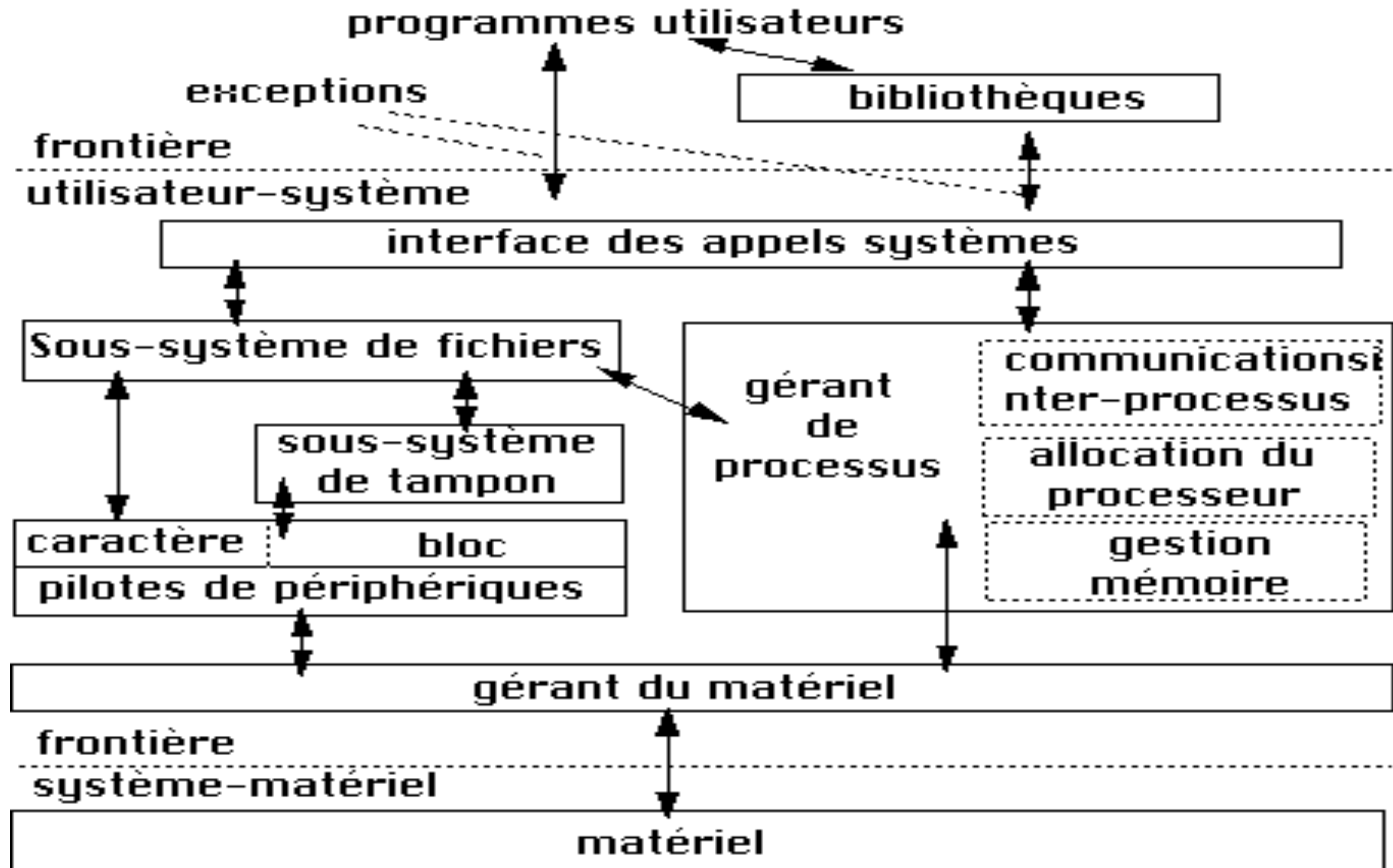
Chaque appel se présente comme un appel de fonction.

Le traitement d'un appel systèmes provoque un passage en mode système (ou noyau).

Lorsque le traitement de l'exception se termine, le processeur repasse en mode utilisateur et on revient exécuter l'instruction se trouvant immédiatement derrière l'appel système.

La plupart des appels systèmes retournent une valeur qui indique le succès ou l'échec de l'opération demandée. En cas d'échec la cause exacte de l'échec peut être précisée par le contenu de la variable globale *errno* (langage C)disponible en incluant le fichier <errno.h>. Celui-ci contient également la liste des codes d'erreur.

Noyau UNIX (suite) - Diagramme bloc



Noyau UNIX (suite)

Le traitement demandé est réalisé par le processus lui-même ce qui évite de faire une commutation et simplifie beaucoup le passage de paramètres et de valeur de retour puisqu'on reste dans le même espace d'adresses. (Les fonctions du noyau sont adressées dans l’espace d’adresses du processus)

* les appels systèmes sont exécutés en utilisant une pile d'exécution différente (Evite le blocage du système) ;

* la priorité d'exécution est plus grande ;

* certaines interruptions peuvent être masquées.

3. Fichiers

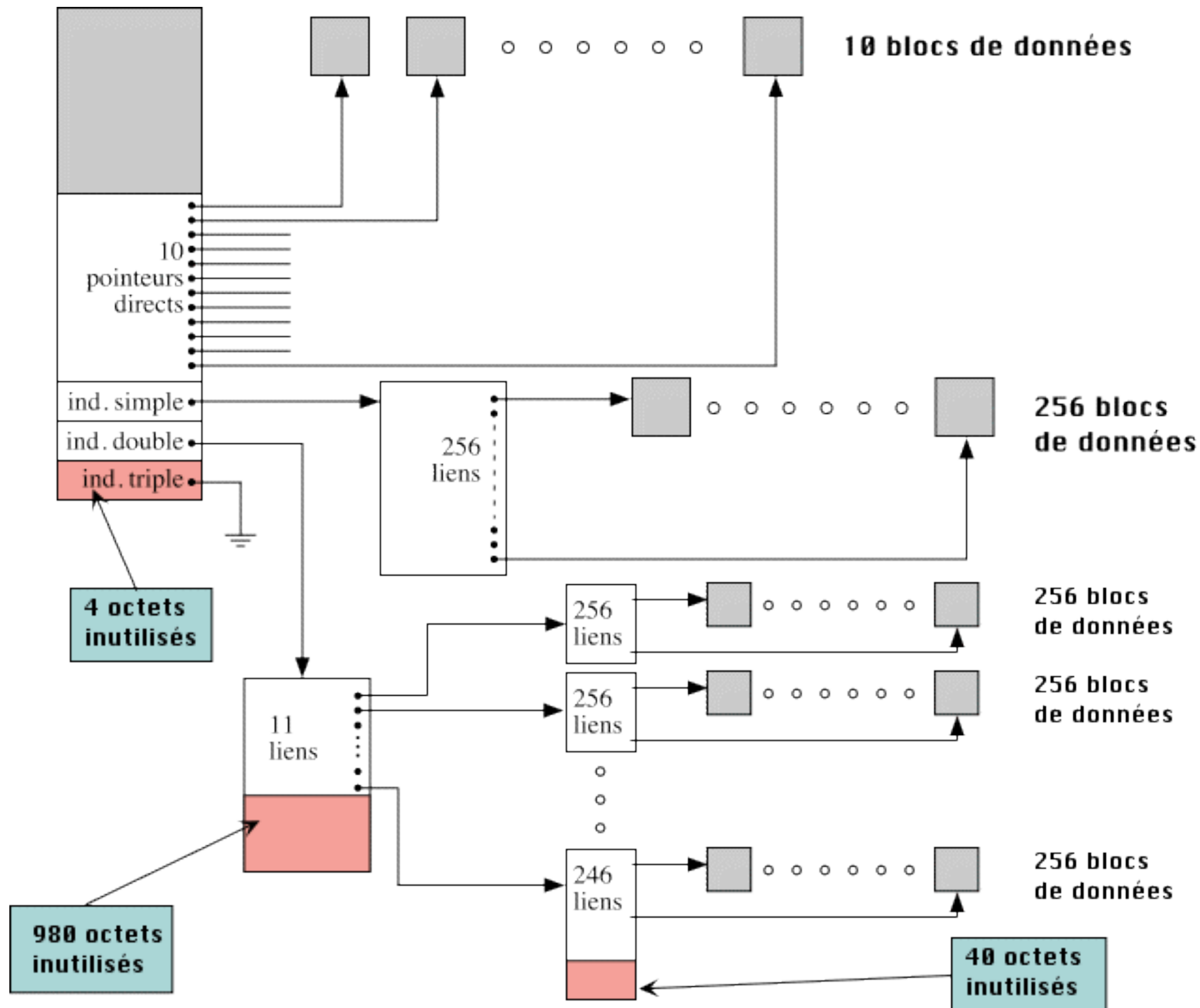
3.1 Structure d'un disque logique(partition) :

Bloc 0 "Bootstap"	Super Bloc	Table des i-noeuds	Bloc des données

Taille d'une partition : #cylindres x #faces x #pistes x #secteurs x taille d'un secteur

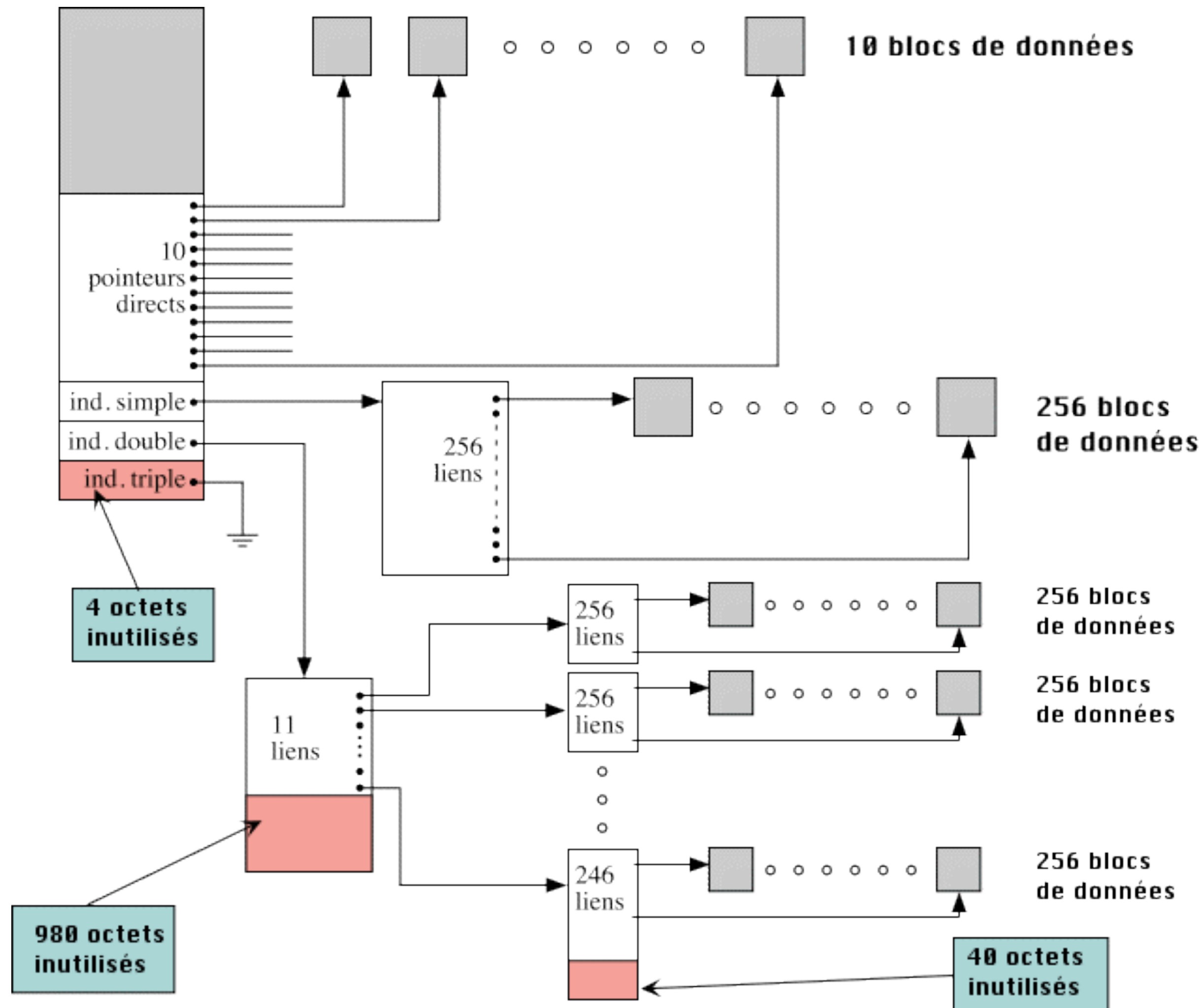
Taille d'un secteur (bloc) = ½ Koctet ou 1 Koctet
1 mot d'un bloc = 4 octets

I-nœud et blocs données d'un fichier :



1. Calculer la taille *T* en octets occupée par le fichier exemple ci-dessus.
2. Calculer la taille maximale *Tmax* d'un fichier selon les hypothèses ci_dessus.

I-nœud et blocs données d'un fichier :



3. Calculer la taille *T* en octets occupée par le fichier exemple ci-dessus. **$T = 1Ko(10 + 2^8 + 10*2^8 + 246) \approx 3072$ Kilo Octets**
4. Calculer la taille maximale *Tmax* d'un fichier selon les hypothèses ci_dessus. **$Tmax = 1Ko(10 + 2^8 + 2^{16} + 2^{24}) \approx 2^{10} \times 2^{24} = 2^{34}$ = 16 Giga Octets**

3.2 Caractéristiques des Fichiers

Tout fichier est défini par un descripteur de fichier unique appelé i-noeud. Il contient les informations concernant le fichier :

- taille ;
- adresse des blocs utilisés sur le disque ;
- identification du propriétaire ;
- permissions d'accès ;
- type de fichier (fichier ordinaire, catalogue, ...) ;
- date de dernière modification ;
- compteur de références à ce fichier dans un répertoire.

L'i-noeud ne contient aucun nom pour le fichier. Un fichier n'est effectivement détruit (espace disque et i-noeud récupérés par le système) que si le compteur de références du i-noeud du fichier devient nul.

Fichiers (suite)

En plus des permissions d'accès et du type du fichier, 3 autres bits ayant un rôle spécial sont utilisés pour chaque fichier :

* set-uid permet d'exécuter un programme avec les privilèges de son propriétaire et non pas ceux de l'utilisateur qui lance l'exécution.
  Ce mécanisme est utilisé en particulier pour changer le mot de passe. Le bit set-uid se traduit lorsqu'il est positionné par une lettre *s* dans les permissions d'accès (affichées lors d'un *ls -l*) ;
* set-gid : même chose avec le groupe ;
* bit de collage (sticky bit) assure le maintien du programme en mémoire même lorsque aucun processus actif ne correspond à une exécution du programme. Un fichier régulier UNIX se comportecomme un tableau de caractères et les opérations de lecture/écriture se font à partir d'un "pointeur de position" qui pointe sur le premier caractère lors de l'ouverture et est ensuite avancé au fur et à mesure des lectures et des écritures.

Fichiers (suite)

- ***Nouvelles_protections*** peut être exprimé de deux manières :
  - 750 correspond en octal à rwxr-x--- (1 = autorisé, 0 = interdit)
  - ug+w rajoute les autorisations d'écriture au propriétaire et au groupe

*X X X r w **x** r w x r w x*

y y y yyy_ yyy r w x r w x r w x

Type Usage User(Owner) Group Other

- ***Type : Type de fichiers :*** Il existe 4 autres bits qui représentent le type de fichier

- : ordinaire; d : directory; p : pipe; s : socket; c : caractere; b : bloc; l : lien (symbolique/physique)

Usage : Il existe 3 autres autres combinaisons qui positionnent des bits spéciaux :

04000 : positionne le **suid bit** (le fichier s'exécute avec les droits du propriétaire)

*XXX r w **s** r w x r w x (le 'x' du propriétaire est transformé en 's')*

02000 : positionne le **sgid bit** (le fichier s'exécute avec les droits du groupe)

*XXX r w x r w **s** r w x (le 'x' du groupe est transformé en 's')*

01000 : positionne le **sticky bit** (le fichier est maintenu en mémoire après son exécution, ex : compilateur …)

*XXX r w **t** r w x r w x (le 'x' du propriétaire est transformé en 't')*

Exemple : $chmod u+s a.out

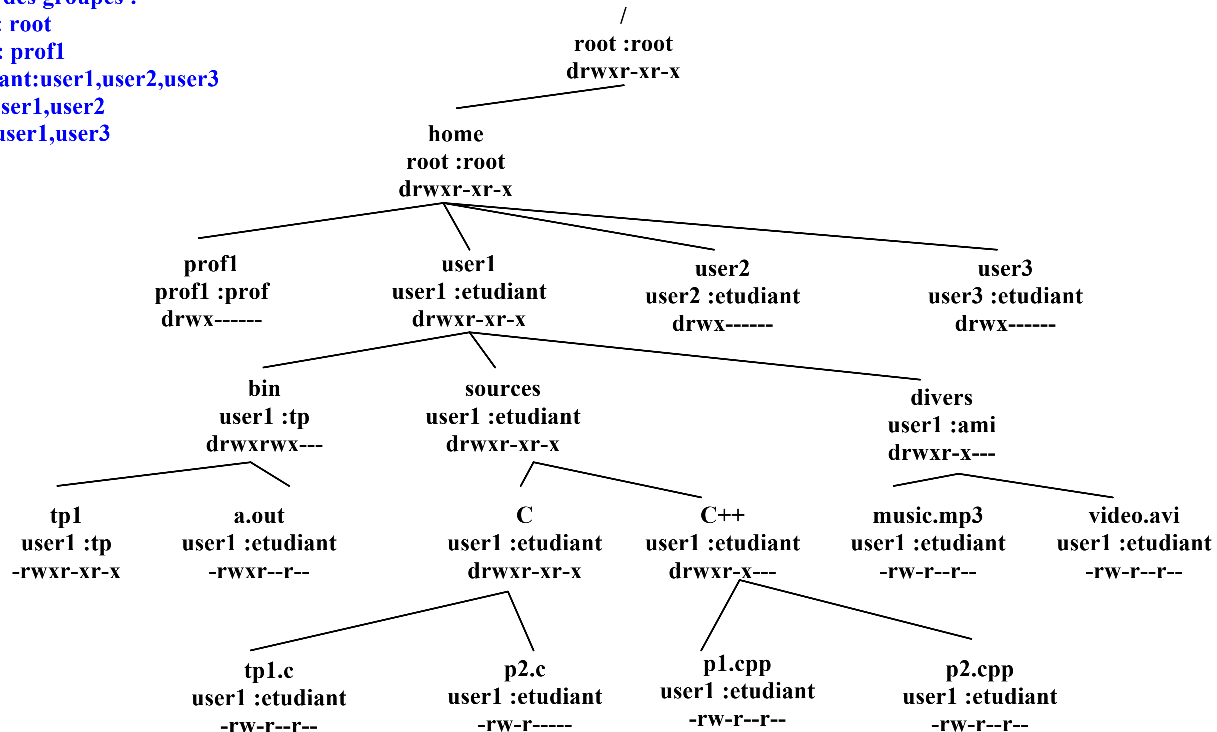
$ ls –l a.out

-rwsr-xr-x

```
-bash-3.00$ ls -l /etc/passwd
-rw-r--r--  1 root root 87900 oct  3 06:00 /etc/passwd

-bash-3.00$ ls -l /usr/bin/passwd
-r-s--x--x  1 root root 15540 jun 20  2004 /usr/bin/passwd*
```

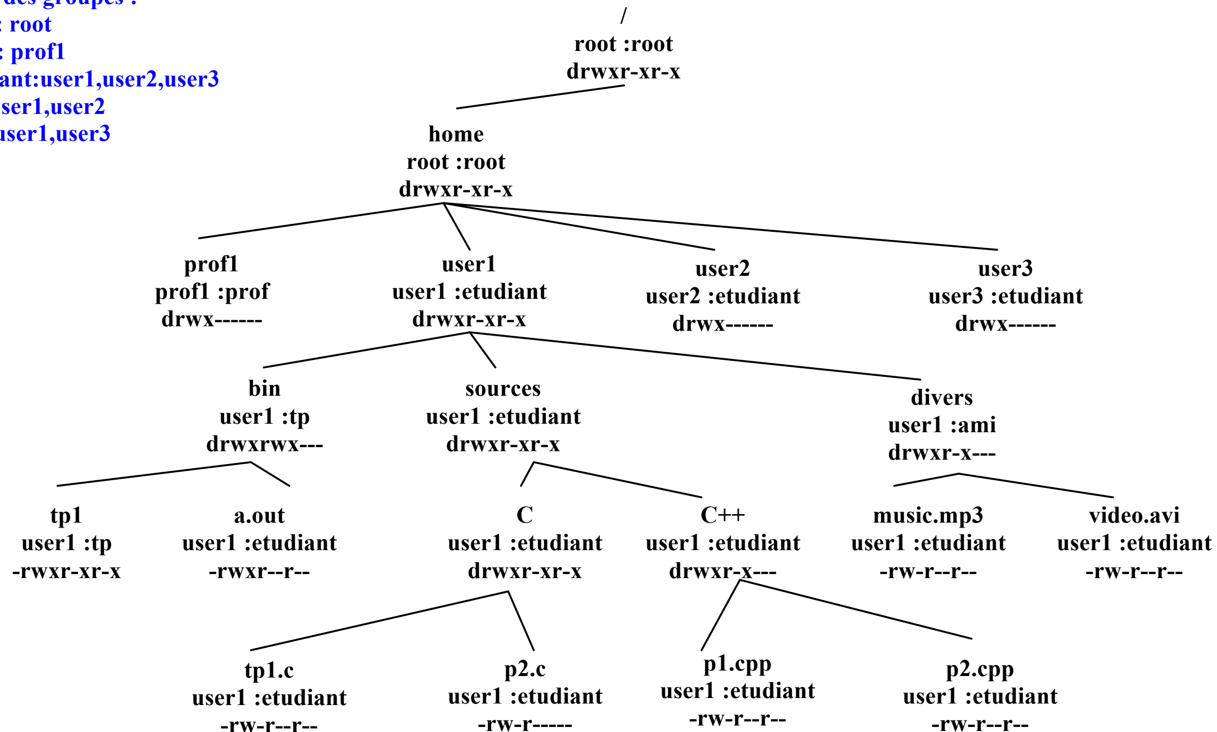
Exemple (cf. TD1) : Soit l'arborescence de fichiers suivante :



Les lignes ci-dessous indiquent "oui" si la commande est possible pour l'utilisateur indiqué et par "non" sinon.

Code	Commande shell	prof1	User2	user3
C1	ls /home/user1/bin	N	O	N
C2	/home/user1/bin/tp1	N	O	N
C3	ls /home/user1/sources/C	O	O	O
C4	ls /home/user1/sources/C/tp1.c	O	O	O
C5	ls /home/user1/divers	N	N	O

Exemple (cf. TD1) : Soit l'arborescence de fichiers suivante :



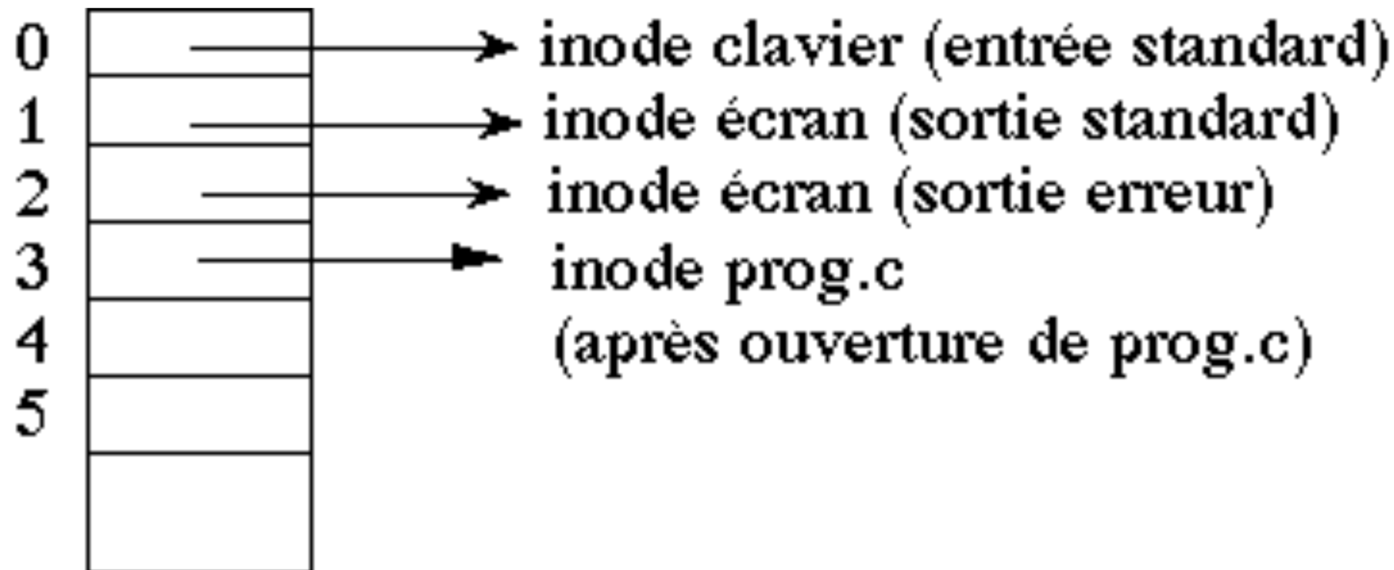
Les lignes ci-dessous indiquent "oui" si la commande est possible pour l'utilisateur indiqué et par "non" sinon.

Code	Commande shell	prof1	User2	user3
C1	ls /home/user1/bin	N	O	N
C2	/home/user1/bin/tp1	N	O	N
C3	ls /home/user1/sources/C	O	O	O
C4	ls /home/user1/sources/C/tp1.c	O	O	O
C5	ls /home/user1/divers	N	N	O

Fichiers (suite) **Tables des fichiers ouverts**

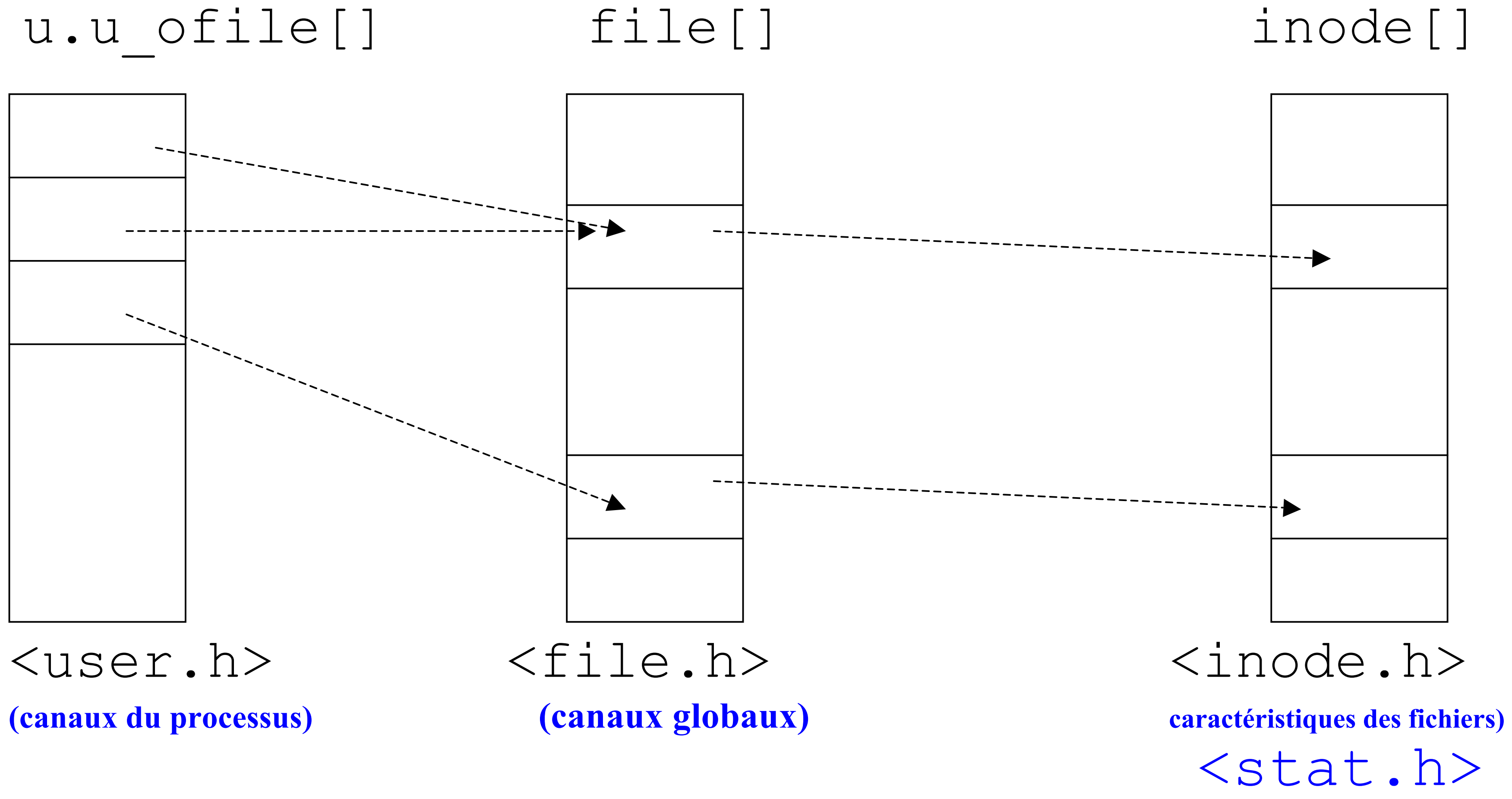
Toute utilisation d'un fichier commence par une ouverture qui renvoie un "file descriptor" traduit ici par "numéro de fichier ouvert" qui n'est que le numéro de l'entrée qui contient un pointeur vers l'inoeud du fichier, dans la table des fichiers ouverts pour un processus.

Après ouverture, un programme désigne un fichier non plus par une référence relative ou absolue mais par le numéro de l'entrée dans la table.
Avantage : inutile de retraduire le nom a chaque opération sur le fichier.



Fichiers (suite) Tables de gestion de fichiers

(Ces structures se trouvent dans /usr/include/sys)



<stat.h>

3.3 Appels Systèmes Opérant sur les Fichiers

```
#include <fcntl.h>
int open(char *référence ,int mode [,int permission])
```

Si le fichier désigné par la référence relative ou absolue `référence` existe alors il est ouvert et `permission` peut être omis.

`mode` indique le type d'accès : `O_RDONLY`, `O_WRONLY` , `O_RDWR` et O_CREAT pour respectivement : lecture seulement, écriture seulement , lecture/écriture et création.

La fonction `open` retourne le numéro de fichier(non négatif) ouvert en cas de succès et -1 sinon. Le pointeur de position du fichier pointe sur le premier octet du fichier.

Exemple :

```
int nf;
nf = open("projet.c", O_RDWR);
```

- Ouverture d'un repertoire

```
#include <dirent.h>
int *opendir(char *référence)
```

Appels Systèmes Opérant sur les Fichiers (suite)

* `int close (int no_fichier);`

  Ferme le fichier de numéro *no_fichier*, ouvert par open()

  L'entrée dans la table des fichiers ouverts est libérée et peut être réutilisée ultérieurement lors de l'ouverture d'un autre fichier.

  La fonction `close` retourne 0 si l'opération se termine normalement et -1 sinon.

Exemple :

```
int nf,ff;
nf = open("projet.c", O_RDWR);
ff = close(nf);
```

• fermeture d’un repertoire

```
#include <dirent.h>
int closedir(DIR *dfp)
```

Appels Systèmes Opérant sur les Fichiers (suite)

* `int read(int no_fichier, char *adr_zone, int nb);`

Essaye de lire *nb* octets à partir de la position indiquée par le pointeur de position du fichier désigné par *no_fichier* et les copie à l'emplacement désigné par *adr_zone*.

La valeur retournée est le minimum de *nb* et de la quantité effectivement disponible et est égale à zéro si la fin de fichier(EOF) est atteinte, ou -1 en cas d'erreur. Le pointeur de position du fichier est augmenté de la quantité lue.

```
int nb;
  char c,tab[100];
  nb=read(0,&c,1); nb=read(nf,tab,sizeof tab);
```

- Ouverture d’un repertoire

```
#include <dirent.h>
struct dirent *p;
p=readdir(DIR *dfp);
```

Appels Systèmes Opérant sur les Fichiers (suite)

* `int write(int no_fichier, char *adr_zone, int nb);`

  Transfère *nb* caractères depuis l'adresse désignée par *adr_zone* à l'endroit pointé par le pointeur de position du fichier *no_fichier*.

  Retourne le nombre de caractères transférés dans le fichier ou -1 en cas de problème.

Exemple :

```
int nb;
char c,tab[8]="bonjour";
nb = write(1,&c,1);
nb = write(nf,tab,sizeof tab);
```

Appels Systèmes Opérant sur les Fichiers (suite)

* `long lseek(int no_fichier, long déplacement, int apartir);`

Déplace le pointeur de position du fichier désigné par *no_fichier* de *déplacement* octets par rapport à la position courante, par rapport au début du fichier, ou par rapport à la fin du fichier suivant que *apartir* est égal respectivement à 0, 1, 2.

Retourne la valeur du pointeur de position après l'opération et -1 en cas d'erreur.

Le début du fichier correspond à la valeur 0.

Appels Systèmes Opérant sur les Fichiers (suite)

```
#include <sys/types.h>
#include <sys/stat.h>
int stat(char *nom , struct stat *buf);
```

Lit l'i-nœud du fichier `nom` et renvoie la partie système de celui-ci sous forme d'une structure pointée par `buf`. Celle-ci contient le numéro d'i-nœud , la taille, le propriétaire, les protections du fichier …

`stat` retourne 0 en cas d'accés réussi et -1 en cas d'erreur.

Appels Systèmes Opérant sur les Fichiers (suite)

Exemple 1:

```
#include <fcntl.h>
void main()
{int no_fichier;
char tab[8]="bonjour";
no_fichier=open("/dev/lpr" , O_WRONLY);
if(no_fichier>0)write(no_fichier, tab,sizeof
tab);
}
```

Exemple 2 : affichage des permissions d'un fichier

Exemple 3 : affichage du contenu d'un répertoire

Autres fonctions systèmes utilisées pour les manipulations sur les fichiers :

`Creat` : création d'un fichier.

`Link`: création d'un lien physique sur un fichier (**ln**).

`unlink`: suppression d'un lien physique sur un fichier et suppression du fichier si c'était le dernier lien existant.

`Dup` : duplication d'une entrée dans la table des fichiers ouverts dans la première entrée disponible.

`Chdir` : changement de répertoire de travail (**cd**).

`Chown` : changement de propriétaire de fichier.

`Chmod` : changement de permission d'accès.

`Mknod` : création d'un répertoire.

`Mount` : montage d'une arborescence d'un volume amovible.

`Tar` : création, modification et lecture d'archives.

4 Processus

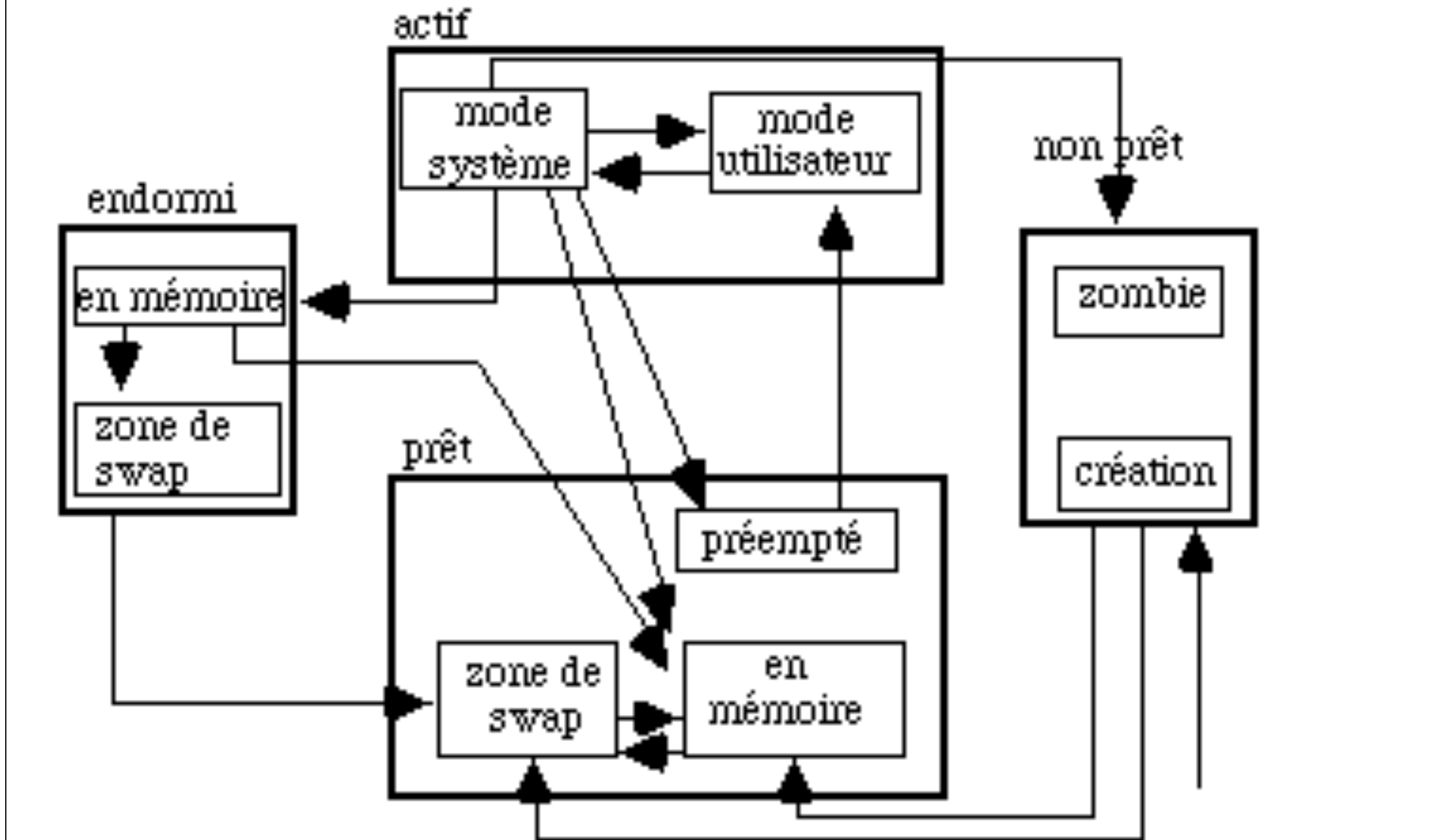
4.1 Caractéristiques des Processus

Un processus est une exécution d'un programme à un instant donné. Le programme lui-même est un objet inerte rangé sur disque sous forme d'un fichier ordinaire exécutable. Plusieurs caractéristiques sont associées à un processus, dont :

- une identification ou Process IDentifier ou PID (entier) ;
- un propriétaire réel ou Real User IDentifier (entier) ;
- un propriétaire effectif ou Effective User IDentifier (entier) ;
- un groupe propriétaire réel ou Real Group IDentfifier (entier) ;
- un groupe propriétaire effectif ou Effective Group ID (entier)
- un terminal d'attachement (le terminal du login+Fenêtre).

Lors du lancement d'un processus, le programme qu'il doit exécuter est chargé en mémoire centrale en vue de son exécution.

Différents états d'un processus :



Différents types de processus :

Un processus système est un processus créé lors du lancement du système pour exécuter des tâches à l'intérieur du noyau :

- Gestion des pages de mémoire,
- Transferts des processus suspendus sur disque ...
- créé lors du démarrage .

Un démon est un processus chargé d'une tâche d'intérêt général :

- Gestion d'impression, gestion réseau, gestion du courrier ...
- Il n'est pas attaché à un terminal.
- créés au démarrage ou par l'administrateur
- interrompus que par l'administrateur ou lors de l'arrêt du système.

Un processus utilisateur est lancé par un utilisateur particulier depuis un terminal donné. Lors du login sur un terminal, un processus est lancé (shell en général , mais ce peut être un autre programme).

4.2 Gestion de Processus par le Système Unix

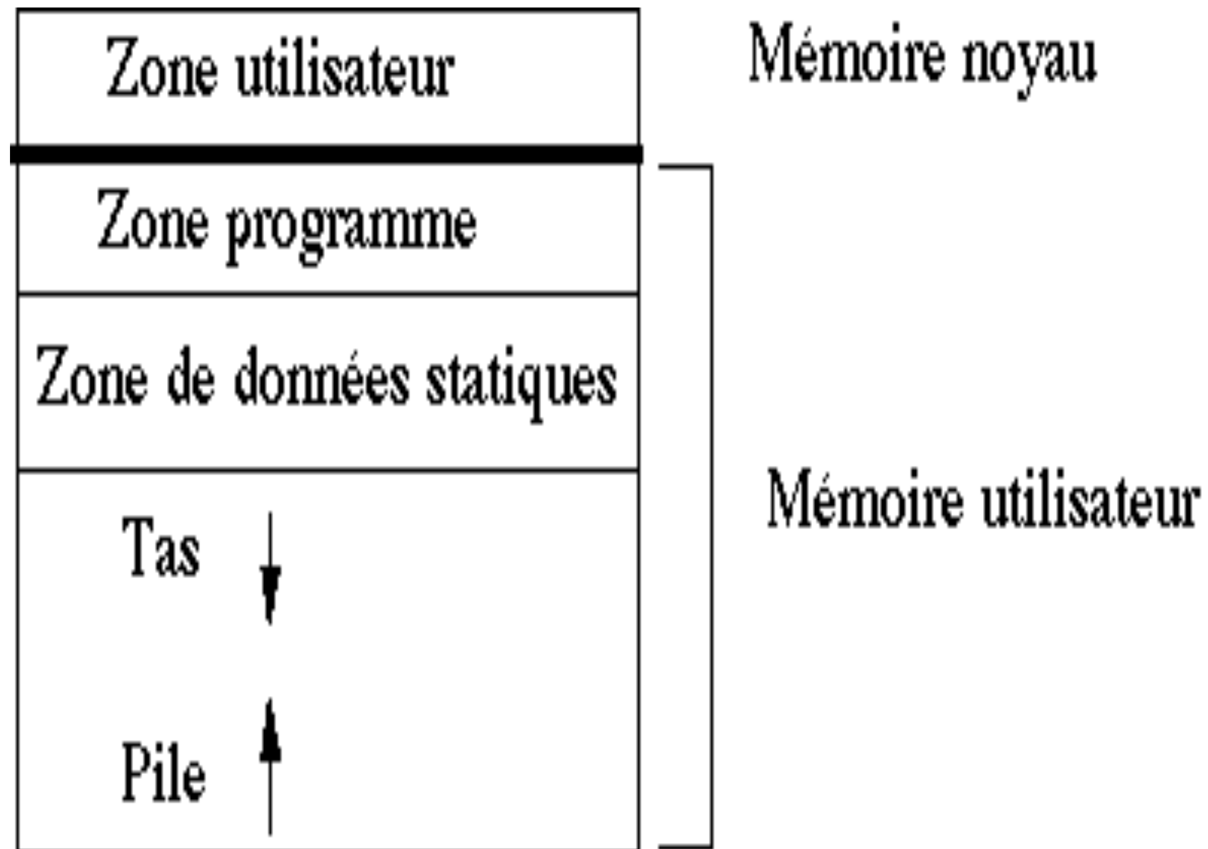
La mémoire allouée à un processus (ou image mémoire) est constitué à partir du fichier qui contient le programme.

C'est un espace d'adresses virtuelles (0 à 2^{32}) propre à chaque processus

Seules les parties utilisées correspondent a des zones de la mémoire physique (cf. gestion de la mémoire page 24).

Des zones physiques peuvent être partagées.

- la zone utilisateur : en zone système, contient les données de gestion du processus : tables des fichiers ouverts, paramètres des E/S ...
- le segment de texte contient la séquence d'instructions exécutée;
- la zone des données statiques (variables globales du programme, ...);
- la zone de données dynamique pour les objets non permanents.
  - un tas pour les variables accédées par pointeurs.
  - une pile pour les variables locales, les paramètres de fonctions, ...);



Zones mémoires d'un processus

Gestion de Processus par le Système Unix (suite)

Tables de gestion des processus

Table des processus

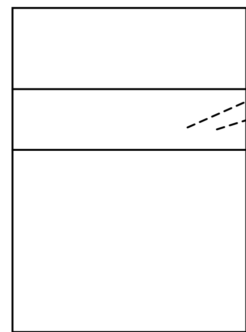
Table des régions par processus

Table globale des régions

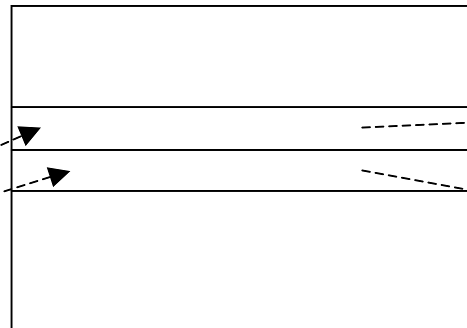
u:

<user.h>

Proc:

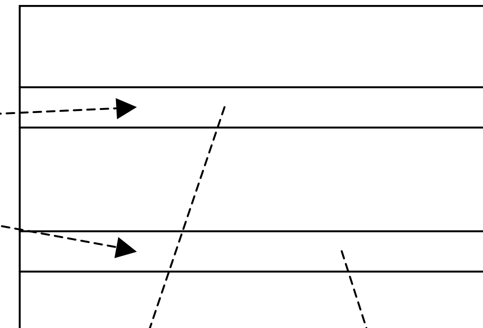


<proc.h>



Text

Data



Mémoire centrale

Gestion de processus (suite)

Remarques :

Programmes généralement partageables :

- si plusieurs utilisateurs demandent l'exécution d'un même programme, une seule copie du segment de texte du programme est placée en mémoire centrale, puis elle est partagée par les différents processus correspondants.
- programmes partageable généralement réentrants :
  - le système affecte des zones mémoire propres pour les données de chacun d'eux => pas d'interférences entre les différentes exécutions d'un même programme.

Table des processus avec une entrée par processus

- contient toutes les informations nécessaires pour que le système puisse gérer le processus lorsqu'il n'est pas actif.
- allouée lors de la création du proc. et libérée lorsqu'il se termine.

Création de processus

- Création par clonage d'un processus père qui utilise l'appel système fork().
- Le processus fils est une copie exacte du processus père avec :
  - le même programme mais
  - des copies des données du père.

Mécanisme de substitution :

- on remplace l'image mémoire d'un processus par une nouvelle image construite à partir d'un fichier exécutable :
  - nouveau programme et
  - nouvelles données
- L'exécution du processus se poursuit au début du nouveau programme.

Mécanisme de suspension : attente la terminaison d'un fils.

4.3 Appels Systèmes Opérant sur les Processus

4.3.1 Création de Processus

```
int fork();
```

Crée un processus, le "fils", copie du processus appelant, le "père".

Le processus fils :

- - exécute le même programme que le processus père;
- - utilise des copies des données du père;
- - mêmes fichiers ouverts que le père et mêmes pointeurs;
- - mêmes propriétaires réel et fictif que le père;
- - mêmes réactions aux signaux que le processus père;
- - même répertoire de travail que le père;
- - attaché au même terminal que le père;
- - identificateur de processus (PID) différent de celui du père.

En cas de succès `fork` renvoie 0 au processus fils, et renvoie le numéro du processus fils nouvellement créé au processus père.

Substitution de programme par un processus

Une famille de primitives **exec** permettent le lancement de l'exécution par le processus appelant d'un nouveau programme

Le texte du nouveau programme recouvre et efface celui de l'ancien programme exécuté.

=> pas de retour d'un exec réussi. Les appels à ces fonctions n'entraînent pas de création de processus mais seulement une modification de l'image exécutée.

execl(*char *ref, char *arg0, ..., char *argn*, 0).

Permet de lancer l'exécution du fichier dont la référence est *ref* avec les paramètres *arg1*, ..., *argn*. Le paramètre *arg0* est obligatoire et doit être la même chaîne de caractères que *ref*.

4.3.2 Fin d'un Processus

`void exit (int n);`

Termine le processus qui l'appelle. La valeur de *n* est disponible pour le processus père du processus appelant et il peut l'obtenir par la fonction wait (cf. § suivant). La fonction `exit` provoque la fermeture de tous les fichiers ouverts par le processus et la libération de toutes les ressources détenues par le processus (mémoire, etc). Les processus fils du processus qui se termine prématurément sont orphelins mais sont adoptés par le processus "init" (processus 1). L'entrée correspondant au processus dans la table des processus est libérée. L'identificateur de processus ne sera pas réutilisé avant un certain temps pour éviter des confusions.

Il n'y a pas de retour pour cet appel système.

4.3.3 Attente de la Fin d'un Fils

```
int wait(int *n);
```

Permet de suspendre un processus jusqu'à la réception d'un signal ou qu'un de ses processus fils se termine ou s'arrête.
Renvoie le numéro du processus qui s'est terminé.
Si `n` est un pointeur non nul, la valeur de l'entier `*n` contient des informations sur l'arrêt du processus fils.
S'il n'y a pas de processus fils, la primitive `wait` retourne immédiatement la valeur -1.

Exemple : Donner les diagrammes et les programmes en C de :

-bash-2.05b$ ls

-bash-2.05b$ ls &

-bash-2.05b$ ls ; more f.c

4.3.5 Quelques Primitives de gestion de processus

`sleep(int n)` : met le processus en sommeil pour n secondes.
`getpid()` : fournit le numéro du processus.
`getppid()` : fournit le numéro du processus père.
`getuid()` : donne le propriétaire.
`getgid()` : donne le groupe du processus.
`setuid(int id)` : change le propriétaire.
`setgid(int gid)` : change le groupe du processus.

Communication inter processus

Communication par fichiers

Partage d’un fichier par 2 processus non apparentés

Partage d’un fichier par 2 processus apparentés

Bilan de la communication par fichiers

Communication par pipes

Communication inter processus

Communication par fichiers

1. Partage d'un fichier par 2 processus non apparentés (chaque processus a son propre pointeur sur le fichier) ➔ - 2 rédacteurs sur un même fichier n'a pas de sens.

2. Partage d'un fichier par 2 processus apparentés (les processus se partagent les pointeurs en écriture et lecture sur le fichier) ➔ - toutes les configurations de processus sont envisageables.

Bilan de la communication par fichiers :

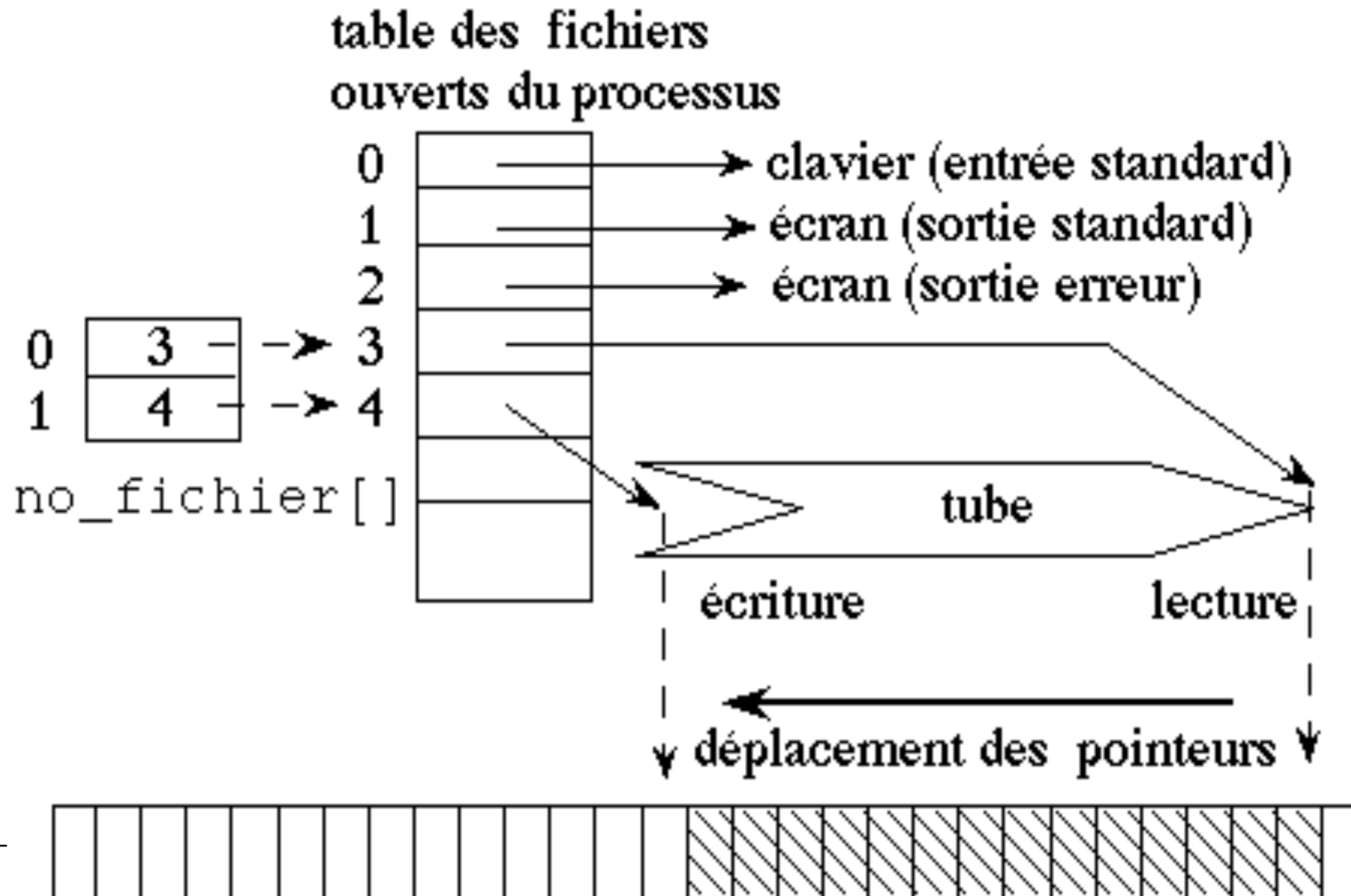
La synchonisation entre rédacteurs et lecteurs n'étant pas gérée ➔

- dépassement de la capacité physique possible,
- arrêt prématuré de la communication entre les processus.

4.4 Communication entre Processus par Tubes

Un tube est un fichier particulier géré par le système. Il existe deux catégories de tubes, les tubes anonymes utilisés par des processus apparentés , et des tubes nommés utilisables par des processus non forcément de même famille.

Ci-dessous l'état de la table des fichiers après la création d'un pipe :



Communication entre Processus par Tubes anonymes(suite)

```
Int pipe (int no_fichier[2]);
```

crée et ouvre le tube à la fois en lecture et en écriture et rend deux numéros de fichiers ouverts, l'un correspondant à l'ouverture en lecture et l'autre à l'ouverture en écriture.

Retourne 0 si la création du tube a bien eu lieu et -1 sinon.

Exemple :

```
Int retour, no_fichier[2];
 Retour = pipe(no_fichier);
```

Un tube ne peut être utilisé que par le processus qui l'a créé ou par ses descendants (du fait de son absence de nom).

L'écriture et la lecture dans un tube se font en utilisant les appels normaux `read()` et `write()` avec comme paramètre le numéro de fichiercorrespondant.

Tubes anonymes(suite)

Il n'y a qu'un seul pointeur de position pour les lectures, qui est partagé par tous les processus "lecteurs".

De même il n'y a qu'un seul pointeur de position en écriture qui est partagé par tous les processus "écrivains".

Pour assurer le caractère FIFO, les modifications de pointeur de position par `lseek()` sont interdites.

tube vide : pointeur de lecture est égal au pointeur d'écriture.

tube plein : pointeur d'écriture -pointeur de lecture = Taillemax.

Lors d'une lecture, le nombre d'octets effectivement lus est le minimum du nombre demandé et du nombre disponible. Le système se charge de suspendre tout processus qui tente de lire dans un tube vide jusqu'à l'introduction de nouveau octets ou la fermeture définitive du tube en écriture. De même pour l'écriture.

Tubes anonymes (suite)

Une "extrémité" d'un tube peut être fermée par appel de `close()` sur le numéro de fichier ouvert correspondant.

Un tube fermé à une extrémité par un processus ne peut être ré-ouvert par ce même processus (absence de nom de tube).

tube définitivement fermé en écriture : tous les processus qui possédaient le numéro de fichier ouvert en écriture l'ont fermé en écriture.

tube définitivement fermé en lecture : tous les processus qui possédaient le numéro de fichier ouvert en lecture l'ont fermé en lecture.

Écrire dans un tube définitivement fermé en lecture n'a pas de sens. Par conséquent toute écriture dans un tube définitivement fermé en lecture provoque l'envoi d'un signal "tube fermé" (SIGPIPE) au processus demandeur. Sauf mention contraire ce signal provoque sa mort (voir signaux).

Tubes anonymes(suite)

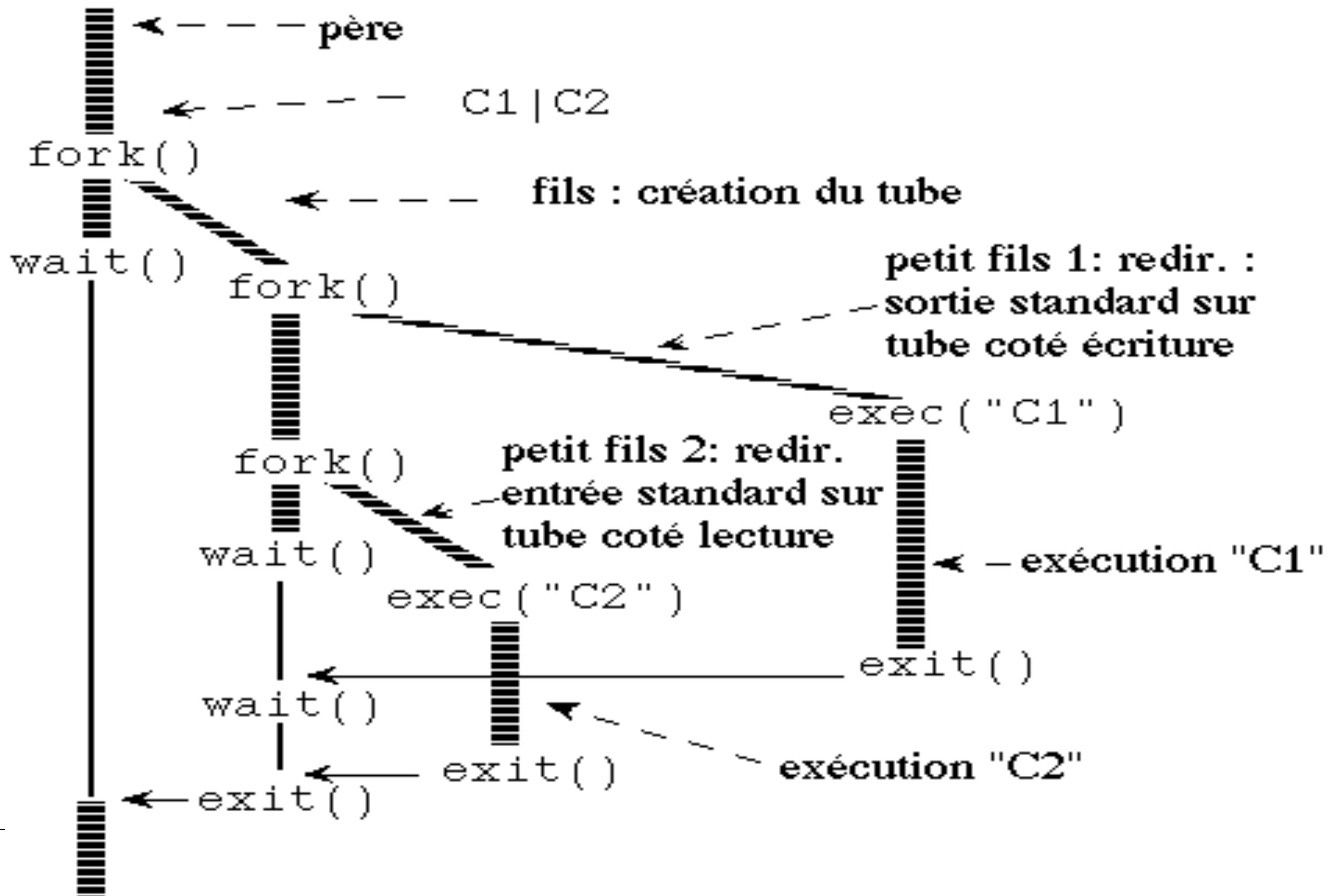
Exemple :
Le programme suivant donne un exemple de communication par pipe entre un processus et son fils:

```
void main(void) {

    int tube[2]; char message[50];
    pipe(tube);
    switch(fork()) {
        case 0 : close(tube[0]); //fils - redacteur
            write(tube[1],"message du fils",15);
            close(tube[1]); //EOF dans tube
            break;
        default : close(tube[1]); //pere - lecteur
            read(tube[0],message,15);
            message[20]=0; //fin de chaîne de caractères
            printf("Message lu par le pere : %s\n",message);
            close(tube[0]);
    }
}
```

Exemple d'utilisation des mécanismes processus par le shell

$ ls | more



4.4.2.Communication entre Processus par un Tube nommé

```
int mknod (char *ref, S_IFIFO|0666, int flg);
```

créé le tube *ref* à la fois en lecture et en écriture.

Retourne 0 si la création du tube a bien eu lieu et -1 sinon.

Exemple :

```
Retour = mknod ("fifo", S_IFIFO|0666, 0);
```

Un tube nommé peut être utilisé par le processus qui l'a créé et par ceux qui sont autorisés.

L'ouverture d'un pipe nommé se fait, comme pour un fichier, avec la fonction *open()*.

L'écriture et la lecture dans un tube se font en utilisant les appels normaux `read()` et `write()` avec comme paramètre le numéro de fichier correspondant.

Exemple1 : Pipe nommé et processus

4.4 Les Signaux(Synchronisation de processus)

4.4.0 Introduction : pourquoi les signaux ? et comment sont ils gérés ?

4.4.1 Caractéristiques des Signaux

Avertir un processus qu'un événement asynchrone exceptionnel important s'est produit.

=> un processus peut réagir à cet événement sans être obligé de le tester

- Mécanisme utilisé par le système pour informer les processus d'erreurs lors des appels systèmes ou d'une erreur d'instruction : overflow, adresse illégale…
- Le signal est envoyé par la routine de traitement de l'exception causée par l'erreur.
- Utilisés pour avertir un processus que l'utilisateur a frappé une touche du clavier du terminal auquel il est attaché.
- Tout processus peut envoyer un signal à un autre processus (moyennant certaines autorisations) par la fonction système `kill`.
- La réception d'un signal provoque le plus souvent la fin du processus qui le reçoit, sauf si installation préalable dune fonction de traitement par `signal`.

4.4.2 Principaux Signaux

nom numéro signification
SIGHUP 1 (hang up) émis à tous les processus associés à un terminal lorsque celui-ci se déconnecte.
SIGINT 2 (interrupt) signal d'interruption émis à tous les processusassociés à un terminal lorsque le caractère d'interruption (en général <CTRL-C>) est tapé.
SIGQUIT 3 (quit) signal d'interruption émis à tous les processus associés à un terminal lorsque le caractère pour quitter une application (en général <CTRL-\>) est tapé.
SIGILL 4 (illegal) émis en cas d'instruction illégale.
SIGTRAP 5 (trap) émis après chaque instruction en cas de traçage de processus.

Principaux signaux (suite)

SIGIOT 6 (input/output trap) émis en cas d'erreur matérielle.
SIGKILL 9 (kill) tue un processus, quel que soit son état.
SIGSEGV 11 (segmentation violation) émis en cas de violation de la segmentation mémoire.
SIGSYS 12 (system) émis en cas d'erreur de paramètre dans un appel système.
SIGPIPE 13 (pipe) émis en cas d'écriture sur un tube sans lecteur.
SIGALRM 14 (alarm) signal associé à une horloge.
SIGTERM 15 (termination) terminaison normale d'un processus.

-bash-3.00 $ kill -l **// liste tous les signaux du système**

Gestion des signaux par Unix

Signaux (suite)

Exemple :

Le programme suivant met en place un traitement pour les signaux `SIGINT` et `SIGQUIT`, puis se suspend pendant 120s avant de se terminer. Il réagit aux signaux `SIGINT` et `SIGQUIT` mais ne meurt pas .

```
#include <signal.h>
void trait_sigint(){
printf("bien recu SIGINT, mais je m'en moque\n");
}
void trait_sigquit() {
printf("bien recu SIGQUIT,mais je m'en moque\n");}
main(){
signal(SIGINT, trait_sigint);
signal(SIGQUIT, trait_sigquit);
sleep(120);
print("je meurs de ma belle mort\n"); }
```

Autres appels systèmes associés aux signaux

* `int pause();`

  Suspend le processus jusqu'à réception d'un signal. Suivant la nature du signal et les traitements de signaux mis en place, il reprend son exécution ou traite le signal puis reprend son exécution, ou bien se termine directement.

* `int alarm (`*`int sec`*`);`

  Programme l'envoi par le système du signal SIGALRM au processus appelant dans *`sec`* secondes.

5. Communication inter processus

Introduction :

IPC (Inter Process Communications) UNIX :

– Messages(envoi de flots de données formattés entre processus)

– segments de mémoire partageables(espace mémoire partagé)

- sémaphores (synchronisation de processus)

Par nature un objet IPC est partageable par plusieurs processus

Déclaration au niveau de la totalité des utilisateurs

Droits d'accès spécifique pour chaque catégorie de processus (user, group, other)

deux types d'accès : lecture ou modification

identifié par un nom externe ou identifiant (entier de type key_t)

utilisé au moyen d'un nom interne ou désignateur (entier)

Comparable au Système de Gestion de Fichiers.

Caractéristiques d’un IPC : ??? = {msg, shm, sem}
Msg : Messages
Shm : Shared Memory
Sem : Semaphores

- Clé numérique dite externe (choisie par l’utilisateur propriétaire)
- Clé interne ou descripteur (entier renvoyé par le système)
- Un appel système ???get(cle externe,…) qui renvoie la clé interne
- Un appel système ???op(cle interne,…)qui opère sur l’IPC
- Un appel système ???ctl(cle interne,…)qui contrôle l’IPC

```
/* Common IPC Structures */

struct ipc_perm {

 uid_t uid; /* owner's user id */
 gid_t gid; /* owner's group id */
 uid_t cuid; /* creator's user id */
 gid_t cgid; /* creator's group id */
 mode_t mode; /* access modes */
 ushort_t seq; /* slot usage sequence number */
 key_t key; /* key */

};
```

5.1 Messages sous UNIX

5.1.1 Caractéristiques des messages sous UNIX

- Flots de données formatés(type, données). Un fichier est une suite d'octets.
- Envoyés dans une FIFO connue par le rédacteur et par le lecteur,
- La FIFO est identifiée par une clé (entier long),
- La FIFO est représentée par une entrée dans la table des descripteurs de fichiers, comparable aux pipes anonymes
- Les fonctions d'envoi et de reception de messages permettent de synchroniser les rédacteur et lecteur. Comparable aux pipes

Si FIFO pleine le rédacteur attend
Si FIFO vide le lecteur attend

5.1.2 Fonctions d'accès :

Les fichiers suivants sont à inclure :

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

Création d'une FIFO de messages:

int msgget(key_t key, int msgflg);

msgget() permet, en donnant un nom externe, de récupérer le nom interne d'une file de messages après création éventuelle si elle n'existait pas;

Exemple : int msgid=msgget(atoi(argv[1], IPC_CREAT|0666)

Envoi de message:

*int msgsnd(int msgid, struct msgbuf *buf, int nboctets, int msgflg);*

msgsnd() permet, à partir d'un nom interne(*msgid*), de pratiquer l'opération d'envoi d'un message de nboctets dans la structure *buf* selon les modalités fixées dans *msgflg*.

La structure suivante représente l'entête d'un message :

```
Struct msgbuf
 { long int mtype;    //type du message (PID en général)
    char *mtext [1];
}
```

msgsnd() renvoi 0 si succès et -1 sinon

Exemple : msgsnd(msgid, message, sizeof(message), 0)

Reception de message:

*int msgrcv(int msgid, struct msgbuf *buf, int nboctets, long mtype , int msgflg);*

msgrcv() permet, à partir d'un nom interne(*msgid*), de pratiquer l'opération de reception d'un message de nboctets dans la structure *buf* selon les modalités fixées dans *msgflg*. mtype contient le type du message reçu.

La structure suivante représente l'entête d'un message :

```
struct msgbuf
 { long int mtype;    //type du message (PID en général)
    char *mtext [1]; // si mtype=0, lecture de tout message
}
```

msgsrcv() renvoi le nbre d'octets reçus si succès et -1 sinon

Exemple : msgrcv(msgid, message, sizeof(message),getpid(), 0)

Contrôle de message:

*int msgctl(int msgid, int cmd, struct msgid_ds *sbuf);*

msgctl() permet, à partir d'un nom interne(*msgid*), de pratiquer l'opération de contrôle d'une file message selon le mode précisé dans *cmd*. Le compte rendu de cette action se trouve dans *sbuf*.

La structure *struct msgid_ds* contient le descripteur de la file de messages (UID, GID, permissions, taille, PID du créateur, PID du dernier utilisateur…)

msgctl() renvoi 0 si succès et -1 sinon

Exemple : msgctl(msgid, IPC_RMID, 0); destruction de la FIFO

Exemple : Clients / serveur par FIFO

5.2 Mémoire partagée sous UNIX

5.2.0 Généralités

Moyen pour le partage de données (documents, …)

Moyen de communication inter processus rapide :

- pas d’appel système,
- pas de gestion de structure,
- exécution en mode utilisateur.

5.2.1 Fonctions système et structures associées permettant l'utilisation des segments de mémoire partageables sous unix

Un segment de mémoire partageable peut être simultanément attaché (en fait incorporé) à l'espace d'adresses virtuelles de plusieurs processus.
Il peut également être incorporé plusieurs fois, mais à des adresses virtuelles différentes, dans l'espace d'adresse d'un processus.

5.2.2 structure shmid_ds associée à chaque segment partageable

```
struct shmid_ds {
struct ipc_perm shm_perm; /* operation permissions*/

int shm_segsz; /* size of segment in bytes */

pid_t shm_lpid; /* pid of last shmop */

pid_t shm_cpid; /* pid of creator */

shmatt_t shm_nattch; /* current # attached */

time_t shm_atime; /* last shmat time */

time_t shm_dtime; /* last shmdt time */

time_t shm_ctime; /* last change time */
};
```

5.2.3 Fonctions associées aux segment de mémoire partageable :

Les fichiers suivants sont à inclure :

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

Création d'une mémoire partagée :

int shmget(key_t key,   u_int size, u_int flags);

Avant toute utilisation, un segment de mémoire partageable doit être créé par la fonction shmget().
Partant d'un nom externe key, cette fonction restitue un nom interne, qui peut être celui d'un segment existant ou celui d'un segment créé à cette occasion.
IPC_PRIVATE permet d'obtenir un nouveau segment.
Le processus créateur définit les droits d'accès et la taille du segment en octets.

Exemple : shmget(12, 1024, IPC_CREAT)

Contrôle d'une mémoire partagée :

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

la fonction shmctl() permet de procéder à des opérations de contrôle :

récupération ou modification du propriétaire,
modification des droits d'accès, ou
destruction du segment partageable qui autrement subsiste jusqu'à la prochaine réinitialisation du système.

Cette fonction peut aussi être utilisée par les autres processus pour obtenir divers renseignements sur un segment partageable.

Exemple : shmctl(shmid,IPC_RMID, 0); //destruction

Attachement d'une mémoire partagée :

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

Une fois créé, un segment partageable peut être attaché à l'espace d'adresses virtuelles d'un processus par la fonction shmat() à condition d'avoir les droits d'accès nécessaires.

Le segment peut être attaché à une adresse virtuelle donnée ou à un emplacement disponible quelconque

Lorsqu'il est attaché à l'espace d'adresses virtuelles d'un processus, ce dernier peut y lire ou y écrire s'il possède les droits d'accès correspondants. (0:R/W)

Un même segment partageable peut être attaché plusieurs fois à des adresses différentes dans l'espace d'adresses virtuelles d'un processus.

Exemple : int *addr = shmat(shmid,0, 0); //addr : @ du segment

Le contrôle d'accès au segment est possible par protection matérielle.

Détachement d'une mémoire partagée :

*int shmdt(caddr_t *addr);*

Il peut en être détaché (enlevé de l'espace d'adresses virtuelles) en utilisant la fonction shmdt().

Exemple : Rédacteur/Lecteur dans un segment de mémoire partagé (à fournir)

5.3 Sémaphores

5.3.0 Généralités

Mécanisme empechant 2 processus ou plus d’accéder simultanément à une ressource partagée, si ces processus testent le sémaphore.

Un sémaphore est booléen ou à compte et est caractérisé par 2 opérations P (Prendre) et V (Vendre).

Ces primitives doivent être fournies par le noyau qui peut :

- partager les données entre les processus,
- peut endormir un processus quand il bloque en exécutant P,
- peut réveiller un processus en attente d’un événement,
- peut exécuter des opérations indivisibles.

Exemples : . P et V en version utilisateur
. P et V en version noyau.

5.3 Sémaphores UNIX

5.3.1 Caractéristiques des sémaphores UNIX

Fonctionnalité identique aux sémaphores de Dijkstra
Paquet de sémaphores (nombre choisi par l'utilisateur)
un paquet est désigné par un un identifiant (entier de type key_t)
un sémaphore d'un paquet est identifié par son numéro
un seul appel système mais avec différentes options :
V avec augmentation du compteur choisie par l'utilisateur
(non limitée a +1)
P avec diminution du compteur choisie par l'utilisateur
(non limitées à -1)
option NOWAIT (permet de retourner immédiatement en cas de blocage)
Z (permet a un processus d'attendre que le compteur soit 0
Possibilité de consulter et de modifier le compteur à tout instant
Dispositions pour réagir à un signal
Dispositions pour réagir à la destruction d'un processus

5.3.2 structure de données associée à un groupe de sémaphore

```
struct semid_ds {
struct ipc_perm sem_perm; /* operation permission */
struct sem *sem_base; /* ptr to first sema. in set*/
ushort_t sem_nsems; /* # of semaphores in set */
time_t sem_otime; /* last semop time */
time_t sem_ctime; /* last change time *
};
```

Structure de données associée à un semaphore:

```
struct sem {
ushort_t semval; /* semaphore text map address */
pid_t sempid; /* pid of last operation */
ushort_t semncnt; /* # awaiting semval > cval */
ushort_t semzcnt; /* # awaiting semval = 0 */
};
```

5.3.3 Fonctions d'accès :

Les fichiers suivants sont à inclure :

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

Création de sémaphore :

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

semget() permet, en donnant un nom externe, de récupérer le nom interne d'un paquet de sémaphores après création éventuelle s'il n'existait pas;

IPC_PRIVATE peut être utilisé pour obtenir un paquet de sémaphores inexistant auparavant.

Exemple : int semgid=semget(12,1, IPC_ |0666) CREAT

Manipulation de sémaphore:

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

semop() permet, à partir d'un nom interne, de pratiquer les opérations P,V;

```
struct sembuf
{
  unsigned short int sem_num;   /* semaphore number */
  short int sem_op;             /* semaphore operation */
  short int sem_flg;            /* operation flag  IPC_NOWAIT ; SEM_UNDO(Utile
si un processus a verrouillé un sémaphore et se termine prématurement*/
} *sops, sem[2];

sops -> sem_op = -1;  // P        sem[0].sem-op=-1;     //P sur 1er sem
sops -> sem_op = 1;   // V        sem[1].sem_op=-1;     //P sur 2nd sem
                                  semop(semid, sem, 2);
```

Consultation de sémaphore :

*int semctl(int semid, int semnum, int cmd, union semun { int val, struct semid_ds *buf, u_short *array,} arg);*

semctl() permet , à partir d'un nom interne, d'affecter les différentes valeurs relatives à un sémaphore ou au contraire de les récupérer :

- le nom du propiétaire,
- la date et l’heure de la derniere modification,
- ….

De connaître le nombre des processus en attente de connaître le PID du dernier processus ayant modifié le sémaphore.

5.3.3 Exemples

- Producteur/consommateur dans une mémoire partagée,

- Situation d'interblocage("deadlock") entre 2 processus P1 et P2 utilisant simultanément 2 ressources R1 et R2.

5.3.4 Limites des IPC

- si un processus "devine" la clé d'une IPC et si les droits le permettent, il peut l'utiliser,
- le noyau ne peut "nettoyer" les IPC, car il ne maîtrise pas l'intention de tous les processus,
- l'espace des noms défini pour les IPCs est trop restrictif et ne peut être étendu au travers du réseau (restreints à un usage sur une machine unique).

5.3.5 Intérêt des IPC

apportent de meilleurs performances aux applications coopérants étroitement que les possiblités du système de fichiers standards.

6. Gestion Mémoire - Introduction

La gestion de la mémoire a deux objets :

- le **partage** de la mémoire physique entre les programmes et les données des processus prêts (mémoire physique = ensemble de mots adressables individuellement dans un ordre aléatoire) ;
- la mise en place des paramètres de calcul d'adresse qui permet de **transformer** une adresse virtuelle en adresse physique.

adresse physique : numéro d'un octet de la mémoire physique
adresse virtuelle : numéro d'un octet dans un espace sans rapport avec la mémoire physique

- < numéro d'octet dans un espace logique>
- < numéro de segment, déplacement dans le segment>
- < numéro de page, déplacement dans la page>

Les adresses physiques et/ ou les adresses virtuelles sont calculées au préalable par les utilitaires de préparation des programmes :

- compilateurs ,
- assembleur,
- éditeurs de liens).

Pour ne pas ralentir les accès à la mémoire, la transformation adresse virtuelle vers adresse physique doit impérativement être faite par le matériel lors de l'exécution des instructions

Gestion Mémoire - Introduction

IL existe plusieurs techniques de gestion de la mémoire :

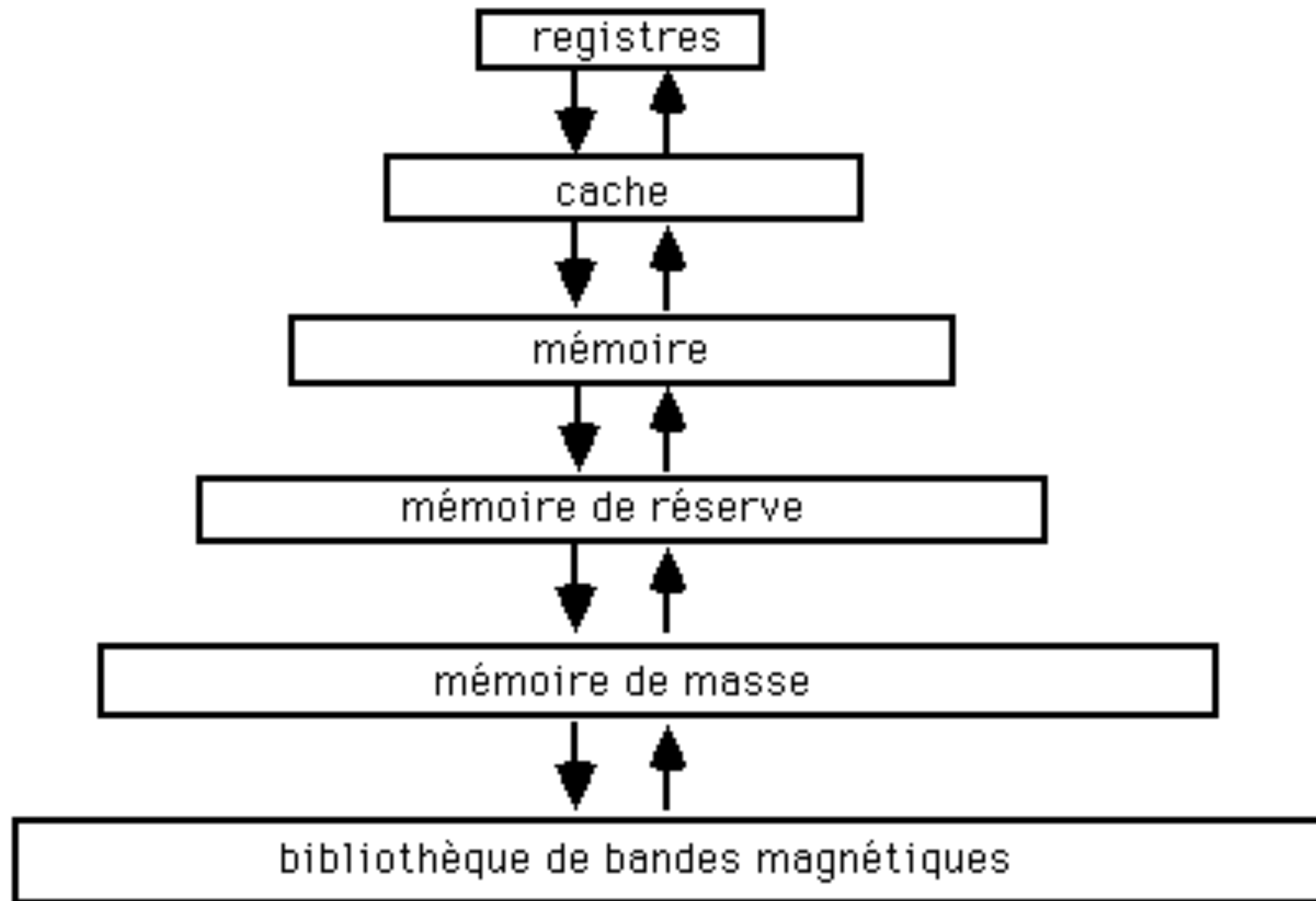
- **Swapping** : Le processus est représenté sur un espace mémoire contiguë. L'intégralité du processus est généralement transféré.

- **Segmentation** : Le processus est représenté sur des segments mémoire de tailles **variables** et non forcément contiguës.

Données et programme se trouvent dans des segments différents pour leur affecter des propriétés différentes(accès en r-x pour le programme et rw- pour les données). Une table des segments est crées pour chaque processus. Chaque entrée de la table contient la longueur et l'adresse du segment en mémoire.

- **Pagination :** Le processus est représenté sur des pages mémoire de tailles fixes et non forcément contiguës.

Données ou programme peuvent être mémorisés dans plusieurs pages. Une table des pages est crée pour chaque processus. Chaque entrée de la table contient la longueur et l'adresse de la page en mémoire.



Hiérarchie de Mémoires

Les mémoires sont hiérarchisées des plus rapides aux moins rapides. Le principe de localité est appliqué à tous les niveaux .

6 .1 Mémoire Virtuelle

Les problèmes dus à la nécessité d'avoir la totalité d'un programme en mémoire centrale pour son exécution :

- sous-utilisation de la mémoire : la totalité du programme (procédures, tableaux, options) est rarement utilisée en même temps et en permanence ;
- impossibilité d'exécuter certains programmes : on peut avoir besoin d'exécuter des programmes qui sont plus gros que la mémoire centrale.

L'utilisation de mémoire virtuelle est une technique permettant l'exécution d'un programme sans qu'il se trouve en totalité en mémoire centrale.

Les avantages attendus sont:

- une dépendance moindre des programmes par rapport à la taille de la mémoire effectivement utilisable d'où programmation simplifiée ;
- davantage de processus en mémoire centrale, d'où un meilleur taux d'utilisation du processeur sans augmenter le temps de réponse ;

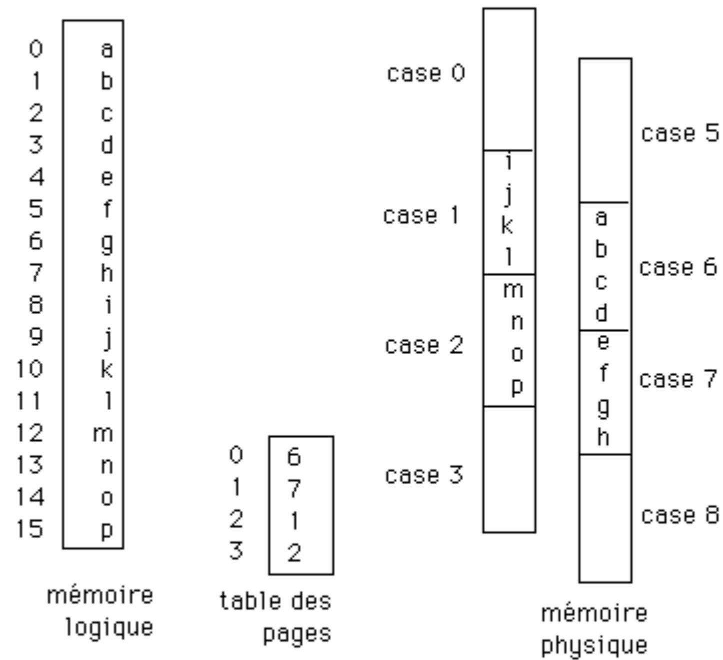
moins d'entrées/sorties que pour les systèmes de swapping.

Gestion de la mémoire sous Unix

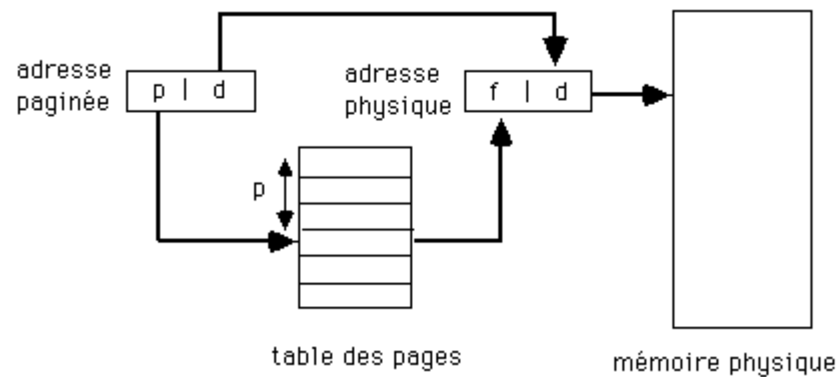
Deux types de gestion mémoire utilisés dans les différentes versions d'Unix :
*** le va-et-vient (swapping pur) jusqu'à version 6.**
Pour exécuter un processus, il faut pouvoir amener toutes ses régions en mémoire centrale .
Un processus présent en mémoire centrale est éventuellement déchargé en totalité s'il est bloqué ou si un autre processus attend depuis plus de trois secondes et la mémoire centrale disponible ne permet pas d'introduire ses régions en mémoire centrale.
Les chargements et déchargements des processus entre la mémoire centrale et la mémoire de réserve sont la tâche de l'unique processus système UNIX : le processus 0 ou swapper.
Les régions partagées entre plusieurs processus ne sont pas touchées par les va-et-vient tant qu'il reste un processus partageur en mémoire centrale.
*** pagination à la demande, depuis System V et BSD 4.3.**
Seules les pages nécessaires à l'exécution d'un processus sont amenées en mémoire centrale depuis la mémoire de réserve.

Algorithme de remplacement de page : chaque case est dotée d'un bit de référence qui est mis à 1 à chaque référence et consulté périodiquement par le processus voleur de pages.

PaginationCalcul d'Adresse en Pagination



Les adresses virtuelles ou paginées sont des couples <p,d>.



Implantation de la Table des Pages

L'implantation de la table des pages peut se faire :

* par un ensemble de registres spécialisés changés à chaque commutation de processus (solution rapide) ;

* en mémoire avec un registre pointeur de la table des pages (solution lente) ;

* en utilisant des registres associatifs.

Important : en pagination simple, la totalité du programme doit se trouver en mémoire centrale lors de l'exécution.

Avantages et Inconvénient de la Pagination

La pagination présente plusieurs avantages :

* l'utilisateur n'a plus à s'occuper de la partition de son programme ;

* le partage de pages de données et de code compilé séparément est effectué avec les mêmes restrictions que pour les segments ;

* la gestion des pages est simple (taille des zones constante).

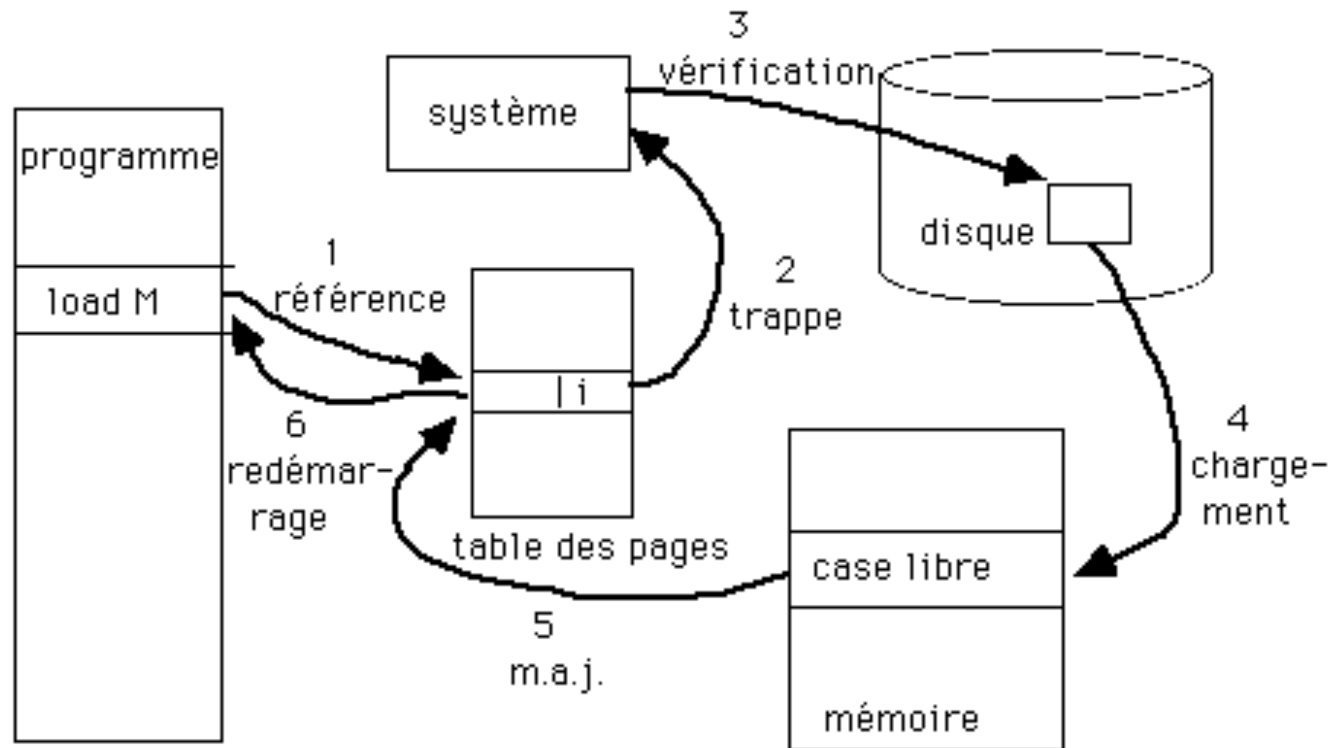
Toutefois, la fragmentation interne se fait dans la dernière page du programme.

Traitement d'une Faute de Page

Stratégies de remplacement de page:

FIFO (« First In First Out ») : la plus vieille

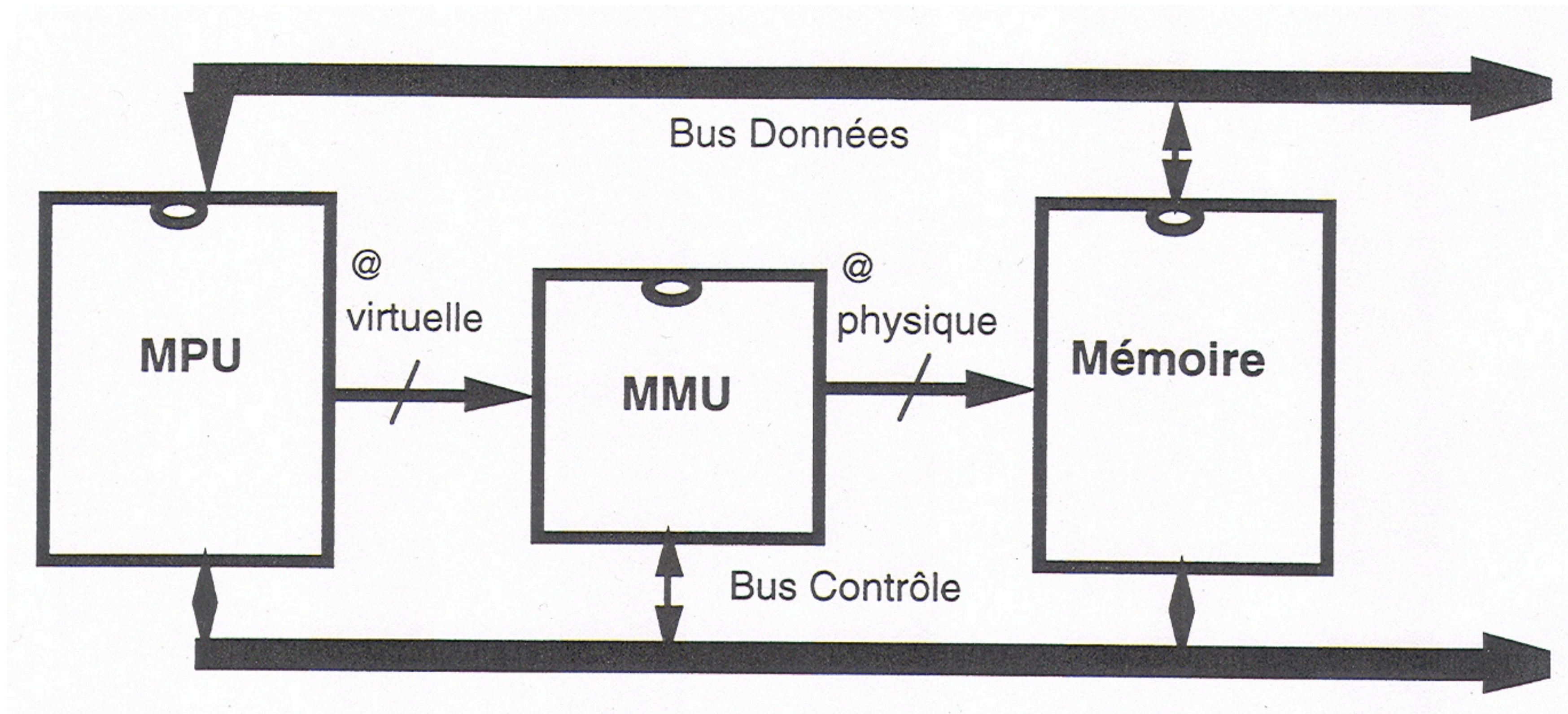
LRU (« Least Recently Used ») : la mois récemment utilisée



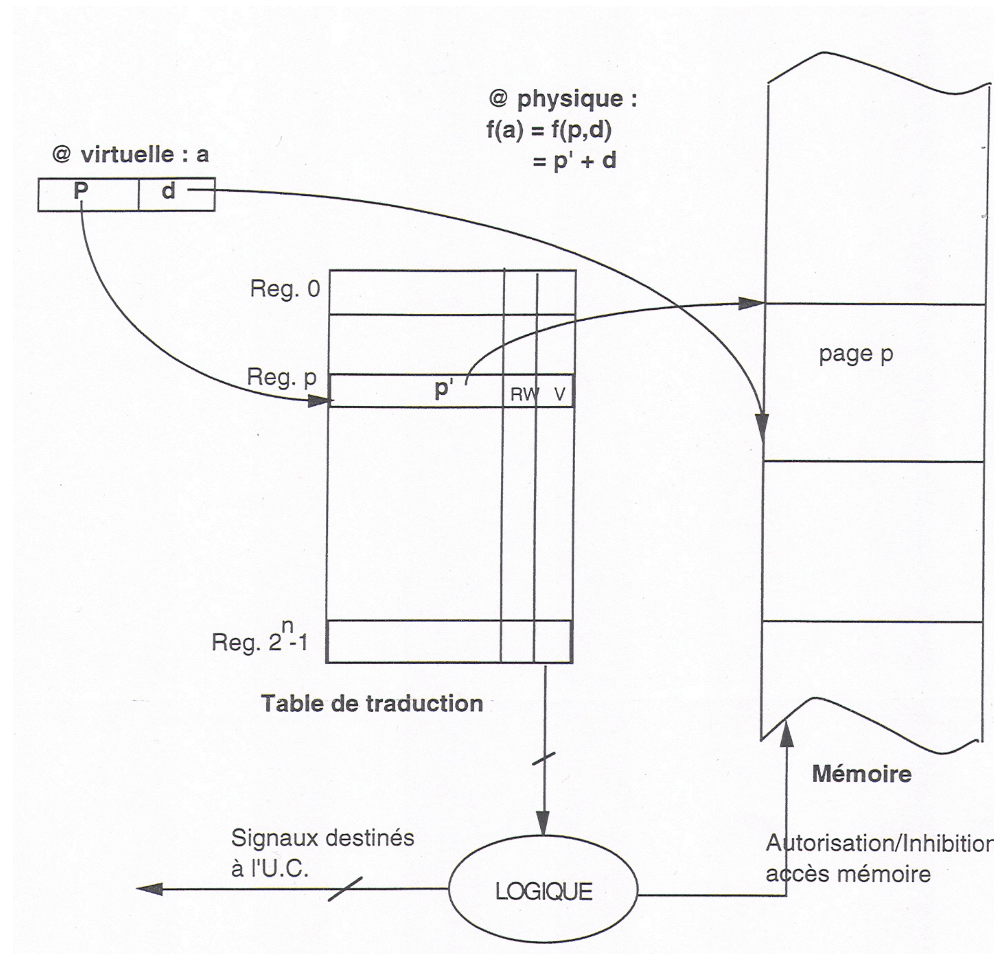
LFU (« Least Frequently Used ») : la mois fréquemment utilisée

Ou la page du processus le moins prioritaire.

Mémoire Virtuelle - Schéma de principe



Mémoire virtuelle = Mémoire centrale + Mémoire secondaire



Mémoire Virtuelle - Mécanisme de calcul d'adresse

Exemple : Traduction d'adresse virtuelles en adresses physiques

Exemple de MMU (« Memory Management Unit »)

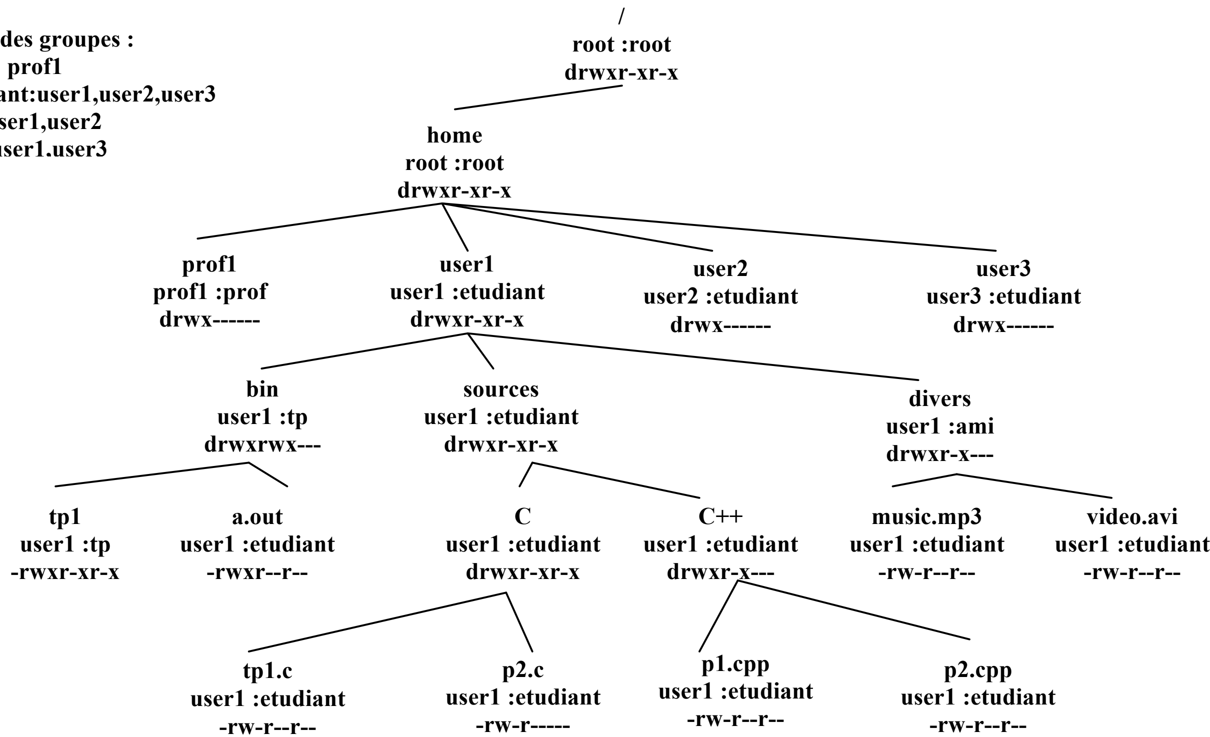
6. Sujets des TDs

TD 1 : Le système de gestion de fichiers

But : L'objet de ce TD est l'étude des primitives de gestion de fichiers par le système Unix

Prérequis : Une connaissance du langage C et des fonctions d'entrée/sortie de la bibliothèque est requise.

Exercice 1 : Soit l'arborescence de fichiers suivante **:**



Remplir les cases vides du tableau ci-dessous par "oui" si la commande est possible pour l'utilisateur indiqué et par "non" sinon.

Code	Commande shell	prof1	user2	user3
C1	ls /home/user1/bin			
C2	/home/user1/bin/tp1			
C3	ls /home/user1/sources/C			
C4	ls /home/user1/sources/C/tp1.c			
C5	more /home/user1/sources/C/tp1.c			
C6	ls /home/user1/sources/C/p2.c			
C7	more /home/user1/sources/C/p2.c			
C8	ls /home/user1/sources/C++			
C9	ls /home/user1/sources/C++/p2.cpp			
C10	more /home/user1/sources/C++/p2.cpp			
C11	ls /home/user1/divers			
C12	ls /home/user1/divers/music.mp3			
C13	ls /home/user1/divers/video.avi			

Exercice 2 :

Ecrire la fonction *getchar1()* qui renvoie un caractère lu sur l'entrée standard, ou la constante EOF de fin de fichier.
Ecrire une version bufferisée de *getchar2().*

Exercice 3 :

Ecrire un programme qui affiche en clair le type du fichier demandé(répertoire, fichier ordinaire, …) ainsi que ses protections(lecture, écriture et exécution).

Exercice 4 :

Ecrire une commande similaire à *ls* qui liste les noms des fichiers contenus dans le répertoire dont le nom est passé en paramètre de votre commande. Celle-ci devra avoir la syntaxe suivante :

Liste repertoire

TD 2 : Les primitives de gestion de processus

But :
L'objet de ce TD est l'étude des primitives de gestion de processus par le système Unix
Prérequis :
Une maîtrise de la notion de primitive système et la pratique des processus en *Shell* sont requises.
Exercice 1 :
a) Ecrire un programme C où le processus père crée *n* processus fils. Chaque fils affiche son PID et son PPID(PID de son père) et se termine en renvoyant un entier qui représente son rang dans la famille. Le père attend chacun de ses fils en affichant pour chacun son PID et son code de retour.

b)Soit le programme suivant :

```
main() {
for(int i=0 ; i<n ; i++) fork() ;
}
```

Donner le relation entre *n* du a) et *p* (nombre de processus).

Exercice 2 :
Ecrire un programme qui réalise la commande $ ls 2> /dev/null en :

- lançant la commande *ls* sur un répertoire passé en paramètre,
- redirigeant la sortie standard de *ls* sur */dev/null*.

Exercice 3 :
Donner le diagramme à base de fonctions système de gestion de processus(fork, exec, wait,exit) permettant de réaliser les commandes shell suivantes :

$ ls ; who | sort ;

TDs 3 & 4 : Les primitives de gestion de pipe et de signaux

But :
L'objet de ce TD est l'étude des primitives de communication inter processus via un pipe par le système Unix .
Prérequis :
Une maîtrise des primitives système de gestion de processus et celles de gestion de fichiers est requise.
Exercice 1 :
Le programme suivant donne un exemple de communication par pipe entre un processus et son fils:

```
#include <unistd.h>
void main(void) {
        int tube[2]; char message[50];
        pipe(tube);
        switch(fork()) {
                case 0 : close(tube[0]);
                        write(tube[1],"message du fils",15);
                        close(tube[1]);
                        break;
                default : close(tube[1]);
                        read(tube[0],message,15);
                        message[20]=0;
                        printf("Message lu par le pere : %s\n",message);
                        close(tube[0]);}}
```

Modifier ce programme de façon à ce que le processus père crée filtre pour son fils parmi les caractères qu'il lit , sur l'entrée standard, les lettres miniscules. les lettres ainsi filtrées sont transmises au fils par un pipe. Le fils les lit du pipe et les place dans un tableau afin de les afficher sur sa sortie standard.

Exercice 2 :Ecrire un programme C permettant d'engendrer un rédacteur et 2 lecteurs dans un pipe nommé "fifo". Le rédacteur est indépendant des 2 lecteurs, en revanche les 2 lecteurs sont père et fils.

Exercice 3 :Ecrire un programme composé de 2 processus, un père et son fils. Le fils doit exécuter une fonction "traite()" toutes les secondes. Le pere demande au fils , au bout d'une minute de s'arrêter. Quand le fils s'arrête , le père s'arrête aussi.

Exercice 4 : Modifier le programme précédent de façon à ce que le fils ecrive "abcd" dans un pipe anonyme hérité de son père. Le père lit et affiche le contenu du pipe avant de se terminer.

Exercice 5 : Modifier le programme précédent de façon à ce que le fils exécute une fonction "traite()" à la demande du père (reception du signal SIGUSR2) et informe le père de la fin de "traite()" (envoi au père du signal SIGUSR1). Le père relance le fils à chaque reception de SIGUSR1.

Exercice 6 :La fonction C mkfifo()(section 3C du manuel en ligne) permet de créer un tube nommé, c'est a dire un tube de communication analogue a celui crée par l'appel système pipe(), mais caractérisé par un chemin d'accès(au même titre qu'un fichier logique).
Commenter les fonctions suivantes et décrire le résultat de leur exécution.

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

```
#include <sys/stat.h>
#include <stdio.h>
#define TAILLEMAX 50

int creer_tube(void) {
        return mkfifo("essai-tube",S_IRWXU|S_IRWXG|S_IRWXO); }

void detruire_tube(void) {
        unlink("essai-tube"); }

void lecture(void) {
        int tube,longueur;
        char message[50];
        tube=open("essai-tube",O_RDONLY);
        longueur=read(tube,message,30);
        message[longueur]='\0';
        printf("%d a lu le message\n\t\t%s\n",getpid(), message) ;
        close(tube);
}
void ecriture(void) {
        int tube;
        char message[50];
        sprintf(message,"processus %d ",getpid());
        tube=open("essai-tube",O_WRONLY);
        write(tube,message,strlen(message ));
        close(tube);
}
```

Ecrire et tester un programme lecteur.c et un programme ecrivain.c pour tester les fonctions lecture() et écriture() en lançant simultanément plusieurs lecteurs et plusieurs écrivains.

TD 5 : Les primitives de gestion des files de messages

But :

L'objet de ce TD est l'étude des primitives de communication inter processus via les files de messages par le système Unix .

Prérequis :

Une maîtrise des primitives système de gestion de processus et celles de gestion de fichiers est requise.

Exercice 1 :

Ecrire un programme C où le père crée une file de messages et N fils. Chaque fils envoie au père, de façon continue, des messages de type 1, suivi de son PID et du texte de la requête. Le père reçoit ces messages et répond en envoyant des messages dont le type est le PID d'un fils suivi de la réponse à la question du fils. Chaque fils reçoit les messages et les affiche. La clé de la FIFO est fournie sur la ligne de commande. La fin de tous les processus est réalisée par l'interception, par le père, de SIGINT et met fin à ses fils et se termine aussi. Si un fils intercepte SIGINT, il en informe son père en lui envoyant SIGUSR1.

Exercice 2 :

Eclater le programme précédent en 2, de manière à ce que les processus deviennent indépendants et apporter à chaque programme les modifications nécessaires. La clé de la FIFO est fournie sur la ligne de commande.

TDs 6 & 7 : Les primitives de gestion des sémaphores et de la mémoire partagée

But :
L'objet de ce TD est l'étude des primitives de communication inter processus via les sémaphores et la mémoire partagée par le système Unix .
Prérequis :
Une maîtrise des primitives système de gestion de processus et celles de gestion de fichiers est requise.
Exercice 1 :
Ecrire deux programmes C :
Le 1er est celui du producteur et consommateur de données dans une mémoire partagée. Ce programme crée une mémoire partagée, l'initialise avec les valeurs de l'indice d'une boucle(allant de 0 à N-1) et se met en « course » avec un autre consommateur pour lire le contenu de la mémoire partagée. Le 1er processus détruit la mémoire partagée et se termine à la réception d'un signal quelconque.
Le 2nd programme est celui de cet autre consommateur.
Proposez une solution sans utilisation de sémaphores.
Exercice 2 :Ecrire les fonctions suivantes :
-int creatsem(key_t cle, int val) qui permet de créer un sémaphore à partir d'une clé numérique « cle » , qui l'initialise avec la valeur « val » et qui renvoie le N° interne du sémaphore ou –1 si erreur.
-void opsem(int semid, int op) qui réalise l'opération « op » (P ou V) sur le sémaphore « semid ».
-void P(int semid) et void V(int semid) qui font appel à la fonction opsem().
Exercice 3 :Ecrire deux programmes C :
Le 1er correspond au producteur dans une mémoire partagée de 2 Kmots. Ce producteur initialise chaque mot (4 octets) de la mémoire partagée avec la valeur de l'indice de la boucle d'initialisation. Après celle-ci le producteur devient consommateur en lisant et en affichant le contenu de la mémoire partagée. Le producteur détruit la mémoire partagée lors de la réception d'un signal quelconque (1 à 31) et met fin à son exécution.
Le 2nd programme joue le rôle de consommateur dès l'instant où le la production des données dans la mémoire partagée est terminée. Il lit et affiche le contenu de la mémoire partagée et se termine.
Exercice 4 :
a) Ecrire un programme C où le père et le fils manipulent deux variables X et Y implantés dans une mémoire partagée. X et Y sont initialisés à 1.
Le père exécute 10 fois les instructions X=X+1 ; Y=Y+1 ;
Le fils exécute également 10 fois : X=2*X ; Y=2*Y ;

Les opérations exécutées par ces 2 processus doivent être atomiques. Les valeurs de X et Y sont identiques à tout instant.

b)Modifier le programme précédent de façon à ce que le fils crée un autre fils qui lit 1à fois les variables X et Y et se termine.

7. Sujets des TPs

Universite d'Evry
Master 1 E3A

TP1 Unix - Gestion de fichiers

Sujet :

PARTIE A :
Le repertoire : /export/home/mmallem/ii71/1fichiers
contient des fichiers :
- entete (.h),
- des executables qui permettent d'afficher en clair le type du fichier demande(repertoire, ordinaire, ...), ainsi que les protections(lecture, ecriture, execution)- (stat et stat2),
- des executables qui permettent d'afficher les noms des fichiers contenus dans le repertoire dont le nom est passe comme parametre a ce programme(dir, dir2, dir3).

Question 1:

Ecrire un programme en C qui prend en compte le type et les protections du fichier, passé comme argument, et de les faire afficher en clair comme le fait la commande SHELL "ls -l".
Vous pouvez constater le resultat attendu en executant les fichiers "stat" et "stat2".

Question 2 :

Ecrire une commande similaire à ls qui liste les noms ainsi que leur numéros d'inodes des fichiers contenus dans le répertoire dont le nom est passé en paramètre de votre commande. Celle-ci devra avoir la syntaxe suivante :

$ dir repertoire
Vous pouvez constater le resultat attendu en executant les fichiers "dir" et "dir2".

Question 3 :
Il s'agit d'afficher pour chacun des fichiers, obtenus par le programme précédent, son type et ses permissions.
Vous pouvez constater le resultat attendu en executant le fichier "dir3".
Attention : ne fonctionne que si le chemin du fichier est complet!
Dans le repertoire courant, cela fonctionne, par contre s'il s'agit d'un autre repertoire il faut prevoir le chemin complet menant aux fichiers de celui-ci.

Universite d'Evry
Master 1 E3A

TP2 Unix - Gestion des processus

Sujet :

PARTIE A : Creation et synchronisation de processus
Le programme "fork.c" se trouvant dans le repertoire :
ens-unix $ /export/home/mmallem/ii71/2processus
permet d'afficher les messages du pere et des fils qu'il cree.

Question 0 :
La compilation du programme "fork.c" est obtenue par la commande :
$ cc fork.c -o fork
L'execution de "fork" est realisee par la commande :
$ ./fork Commenter ce qui se passe.
Question 1 :
Modifier le programme fork.c de maniere a ce que chaque processus fils se terminera toujours avant son pere. Commenter ce qui se passe.
Vous pouvez constater le resultat attendu en executant le fichier "fork2".
Question 2 :
Modifier le programme fork2.c, que vous aurez ecrit, de maniere a ce que chaque processus fils se terminera toujours avant son pere et que le fils renverra un code(1: pour fils1 et 2: pour fils2) au pere que celui-ci affichera. Commenter ce qui se passe.
Vous pouvez constater le resultat attendu en executant le fichier "fork3".

Les Fonctions a utiliser
exit(i)
termine un processus, i est un octet renvoyé dans un 'int' au processus père.
wait(&etat)
met le processus en attente de la fin de l'un de ses processus fils .
La valeur de retour de 'wait' est le numéro du processus fils venant de se terminer.
Lorsqu'il n'y a plus (ou pas) de processus fils à attendre, la fonction wait renvoie -1.
Chaque fois qu'un fils se termine le processus père sort de 'wait', et il peut consulter 'etat' pour obtenir des informations
sur le fils qui vient de se terminer.
'etat' est un pointeur sur un mot de deux octets. L'octet de poids fort contient la valeur renvoyée par le fils (i de la
fonction exit(i)), et l'octet de poids faible contient 0.

En cas de terminaison anormale du processus fils, l'octet de poids faible contient la valeur du signal reçu par le fils.
Cette valeur est augmentée de 80 en hexadécimal (128 décimal), si ce signal a entrainé la sauvegarde de l'image mémoire du processus dans un fichier 'core'.

____

PARTIE B : Creation et transformation de processus

Ecrire un programme C "exec-ls.c" qui permet de remplacer le code du processus fils par celui de la commande 'ls'. La sortie de 'ls' est redirige dans le fichier “ls.out”.

L'execution du programme suivant donne une idee du resultat attendu :

$ ./exec-ls

Question 1 :

Modifier ce programme de maniere a ce que le resultat de 'ls' s'affichera a l'ecran.

Vous pouvez constater le resultat attendu en executant le fichier "exec-ls2".

Question 2 (preparation du TP3):

Ecrire un programme qui realise la commande $ ps -a | more:

Vous pouvez constater le resultat attendu en executant le fichier "ps-more".

Universite d'Evry
Master 1 E3A

TP3 Unix - Communication entre processus par pipe

Sujet :

Le programme "pipe.c" se trouvant dans le repertoire :
ens-unix $ /export/home/mmallem/ii71/3pipes/
permet au fils de communiquer a son pere un message par le biais d'un pipe.
La compilation de ce programme est obtenue par la commande :
$ cc pipe.c -o pipe

L'execution de "pipe" est realisee par la commande :
$ ./pipe

Question 1 :
Que se passe t il si l'instruction close(p[1]) est supprimee du code du fils et pourquoi ?

Question 2 :
Que se passe t il si l'instruction close(p[1]) est supprimee du code du pere et pourquoi?

Question 3 :
Modifier le programme pipe.c de maniere a ce que le pere filtre pour son fils parmi les caracteres qu'il lit , sur l'entree standard , les lettres miniscules. Les lettres ainsi filtrees sont transmises au fils par un pipe. Le fils les lit du pipe et les transmet à son fils(petit fils) via un deuxieme pipe. Le petit fils les lit du pipe et les place dans un tableau afin de les afficher sur sa sortie standard.

Vous pouvez constater le resultat attendu en executant le fichier "pipe2".

Question 4 :

Ecrire un programme C permettant d'engendrer un rédacteur et 1 lecteur dans un pipe nommé "fifo". Le rédacteur est independant du lecteur.
Le redacteur ecrit la chaine "abbccc", trois fois, dans le pipe et le lecteur se charge de vider le pipe.

Vous pouvez constater le resultat attendu en executant le fichier "fifo1".

Question 5 :

Ecrire un programme C permettant d'engendrer un rédacteur et 2 lecteurs dans un pipe nommé "fifo". Le rédacteur est independant des 2 lecteurs. Ces derniers sont pere et fils.
Le redacteur ecrit la chaine "abbccc", trois fois, dans le pipe et les lecteurs se chargent de vider le pipe de façon asynchrone en faisant afficher leur résultat respectif.
Vous pouvez constater le resultat attendu en executant le fichier "fifo2".
Question 6 (facultative):
Ecrire un programme qui réalise la commande shell :
$ ps -e | more

Universite d'Evry
Master 1 E3A

TP4 Unix - Communication et synchronisation entre processus par pipe et signaux

Sujet :

Le programme "signal.c" se trouvant dans le repertoire :
ens-unix $ /export/home/mmallem/ii71/4signaux/
permet de generer 2 processus un pere et son fils. Le fils execute une fonction traite()
toutes les secondes et s'arrete quand le pere le lui demande.
Le pere s'endort pendant un certain temps. A son reveil, il demande au fils de se terminer.

La compilation de ce programme est obtenue par la commande :
$ cc signal.c -o signal

Son execution est realisee par la commande :
$ ./signal

Question 1 :

Pendant combien de temps le pere s'endort il et comment fait il pour reveiller le fils ?

Question 2 :

A quoi servent les fonctions alarm() et pause()?

Question 3 :

Que se passe t il si la fonction signal()est supprimee du code du fils et pourquoi ?

Question 4 :

Que se passe t il si la fonction signal()est supprimee du code du pere et pourquoi ?

Question 5 :

Modifier le programme signal.c de façon a ce que le fils ecrive dans le pipe a chaque reception du signal SIGALRM le message "abcd".
Le pere a son reveil recupere le message du fils du pipe et met fin au fils.

Vous pouvez constater le resultat attendu en executant le fichier "signal2".

Question 6 : Ecrire un programme C dans lequel le fils recherche dans le fichier "annuaire" toutes les lignes commençant par le caractère, passe en parametre, les stockent dans un pipe anonyme.
Le contenu du pipe est lû par le père qui l'affiche après l'avoir lû.
Le père envoie ensuite le signal SIGUSR1 au fils. Ce dernier accuse réception en affichant le message :
"Fils de PID a bien recu SIGUSR1 du pere de PID ".

Vous pouvez constater le resultat attendu en executant le fichier "signal3 caractere".

Question 7 : Modifier le programme precedent de maniere à ce que le fils fasse la recherche d'un abonne dans "annuaire" a chaque fois que le pere le lui demande. Ce dernier le fait 3 fois en envoyant au fils, a chaque fois le SIGUSR2. A chaque fois que le fils a stocke le resultat de sa recherche dans le pipe, il en informe son pere en lui envoyant SIGUSR1. Apres 3 iterations le pere envoie a son fils SIGUSR1 pour lui demander de se terminer et se termine aussi. Dans l'iteration i+1 le fils recherche les abonnes dont le nom commencent par argv[1][0]+1.

Universite d'Evry
Master 1 E3A

TP5 Unix

Sujet : **PARTIE A : Communication entre processus par messsages**

Le programme "messecr_lec.c" se trouvant dans le repertoire :
ens-unix $ /export/home/mmallem/ii71/5ipc/messages/
permet a un processus d'envoyer des messages dans une FIFO, de les relire et les afficher .

L'execution de ce programme est realisée par la commande :
```
bash $ ./messecr_lec CLE texte_message
```

Question A.1 :

Creer 2 programmes l'un pour l'ecrivain(messecr.c) et un autre pour le lecteur(messlec.c) sur une meme FIFO de messages.

L'execution de ces programmes est realisée par les commandes :
```
bash $ ./messecr CLE texte_message && ./messlec CLE &
```

Question A.2 :

Que se passe t il si l'on execute les commandes suivantes et pourquoi:
```
bash $ ./messecr_lec CLE texte_message &&  ./messlec CLE &
```

PARTIE B : Communication entre processus par memoire partagée

Sujet :

Les programmes "initsem.c" et "produc.c"se trouvant dans le repertoire :
ens-unix $ /export/home/mallem/cours/ii71/5ipc/semaphores/
permettent de creer des semaphores et un producteur de donnees.
Ce dernier stocke les donnees dans une memoire partagee.
L'execution de ces programmes est realisee par les commandes :

```
$ ./initsem CLE_SEM_PROD CLE_SEM_CONSOM
```

Question B.1 :
Creer le programme("consom.c") du consommateur de ces donnees.
L'execution des programmes du producteur et du consommateur est realisee par les commandes :

```
$ ./produc CLE_SEM_PROD CLE_SEM_CONSOM CLE_MEM_PARTAGEE &
./comsom CLE_SEM_PROD CLE_SEM_CONSOM CLE_MEM_PARTAGEE
```

Question B.2 :
Que se passe t il si l'on execute les commandes suivantes et pourquoi

```
$ ./produc CLE_SEM_PROD CLE_SEM_CONSOM CLE_MEM_PARTAGEE ;
./comsom CLE_SEM_PROD CLE_SEM_CONSOM CLE_MEM_PARTAGEE
```

ou

```
$ ./produc CLE_SEM_PROD CLE_SEM_CONSOM CLE_MEM_PARTAGEE &&
./comsom CLE_SEM_PROD CLE_SEM_CONSOM CLE_MEM_PARTAGEE
```