Programmation Unix

Les Appels Système

Gestion des fichiers

Description d'un volume

BLOC DE BOOT	Contient un boot strap
SUPER BLOC	Caractéristiques du volume
i-list	Liste de i-nodes (index nodes) ➢ cette liste a une taille finie: nombre fini de fichiers
Blocs d'informations et blocs libres	Blocs contenant les données des fichiers et blocs disponibles.

1

Structure du super bloc

taille de la i-list
taille du volume
taille liste des blocs libres
liste partielle des blocs libres
nombre d'inodes libres
liste partielle des i-nodes libres
verrou liste des blocs libres
verrou liste des i-nodes libres
Date
nombre total des blocs libres
nombre total des i-nodes libres
1er facteur d'entrelacement

Verrous: éviter que plusieurs processus ne manipulent simultanément ces listes

Indique comment sont numérotés les blocs afin d'optimiser le déplacement des têtes de lecture-écriture

2

Structure d'une i-node

Type de fichier
Protection
Identité du propriétaire (uid)
Groupe du propriétaire (gid)
Nombre de liens
Taille du fichier
Date de dernière modification
Date de la dernière lecture
Zones des adresses des blocs de données

L'i-node décrit le fichier et indique sa localisation physique

- Il est chargé en mémoire dès qu'on utilise le fichier.
- **Le nom du fichier ne figure pas !!**
- Chaque i-node est identifiée dans la i-liste du volume par un N°

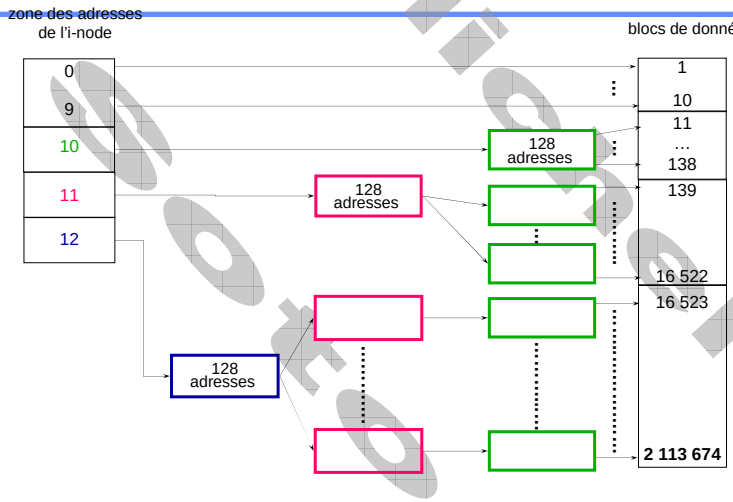
3

Zone des adresse des blocs de données d'une i-node

Adresse du bloc 0
Adresse du bloc 1
Adresse du bloc 2
Adresse du bloc 3
Adresse du bloc 4
Adresse du bloc 5
Adresse du bloc 6
Adresse du bloc 7
Adresse du bloc 8
Adresse du bloc 9
Adresse du bloc d'indirection simple
Adresse du bloc d'indirection double
Adresse du bloc d'indirection triple

4

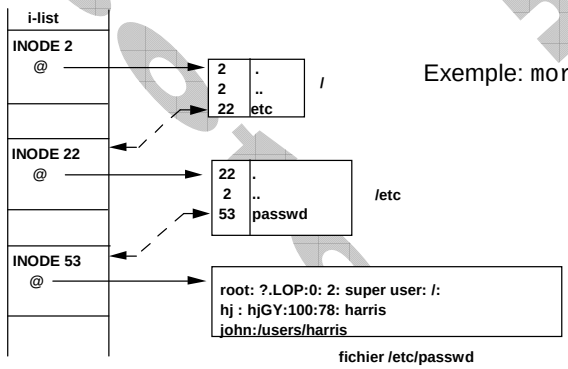
Zone des adresses des blocs de données d'une i-node



5

Utilisation d'un fichier

- Les répertoires servent à maintenir la correspondance `N°i-node←→Nom de fichier`
- Les répertoires sont des fichiers et possèdent donc une i-node
- Le répertoire racine « / » possède toujours l'i-node n°2



Exemple: `more /etc/passwd`

6

Tables en mémoire exploitées par le noyau `u_ofile`

- **`u_ofile`:** table des descripteurs de fichiers d'un processus
  - Chaque processus possède la sienne
    - Vision PAR PROCESSUS des fichiers ouverts
- Lors de l'ouverture d'un fichier, le noyau lui associe une entrée dans cette table
- Se trouve dans le structure U associée à chaque processus

```
struct user{
       struct inode *u_cdir; <-- pointe sur le répertoire courant
       struct inode *u_rdir; <-- pointe sur le répertoire racine
                                 du volume
       short u_cmask <-- protection
       struct *u_ofile[NOFILE] <-- NOFILE: nombre de fichiers
                                   maximun ouverts par un processus
       ....
      }
```

7

Tables en mémoire exploitées par le noyau `u_ofile`

- Descripteurs standards dans la `u_ofile`
  - 0 (`stdin`) : flux correspondant à l'entrée standard
  - 1 (`stdout`) : flux correspondant à la sortie standard
  - 2 (`stderr`) : flux correspondant à la sortie d'erreur standard
- Exemple

```
...
fprintf(stderr,"erreur numero %d", errno);
…
```

- Chaque entrée associée à un fichier ouvert pointe vers une entrée de la table des ouvertures de fichiers: `file table`

8

Tables en mémoire exploitées par le noyau `file table`

- Contient les informations sur tous les fichiers ouverts dans le système par l'ensemble des processus à un instant donné.
  - ➢ Une entrée utilisée par ouverture de fichier
- Permet à plusieurs processus de même filiation de partager un fichier ouvert

```
struct file{
  char f_flag   <-- mode d'ouverture écriture, lecture, pipe
  cnt_t f_count <-- nombre de processus qui accèdent au fichier
  struct inode *f_inode <-- pointeur vers la table des i-nodes
    .........
}
```

9

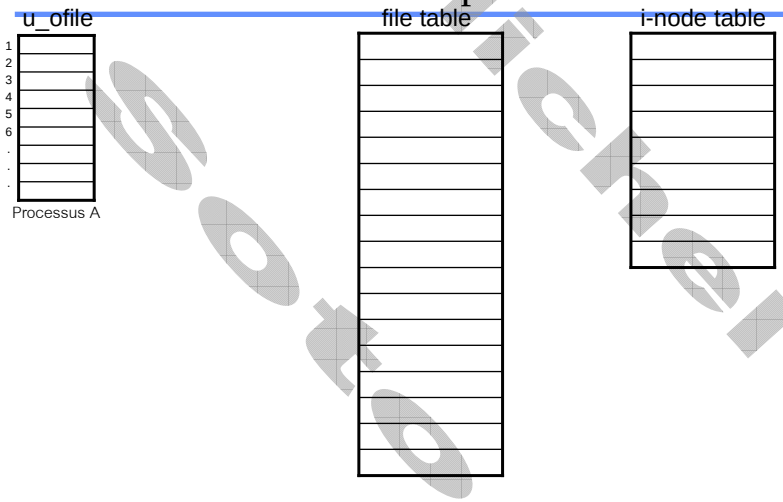
Tables en mémoire exploitées par le noyau
`i-node table`

- Permet la localisation physique du fichier.
- L'`i-node` de la `i-list` du fichier ouvert est chargée dans une entrée de cette table lors de sa première ouverture.
  - Vision GLOBALE des fichiers ouverts
  - Une entrée utilisée par fichier ouvert

```
struct inode {
  flag   <-- indique si l'i-node est
             vérouillée, modifiée,...
   count <-- nombre de références
   dev   <-- device de résidence
   number <-- son numéro (sa place dans la i-list)
  }
```

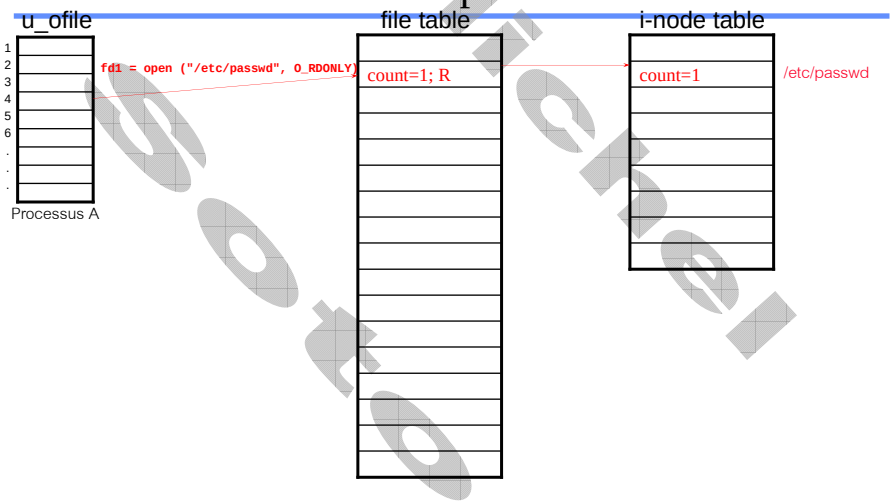

10

`u_ofile`, `file table`, `i-node table`
exemples



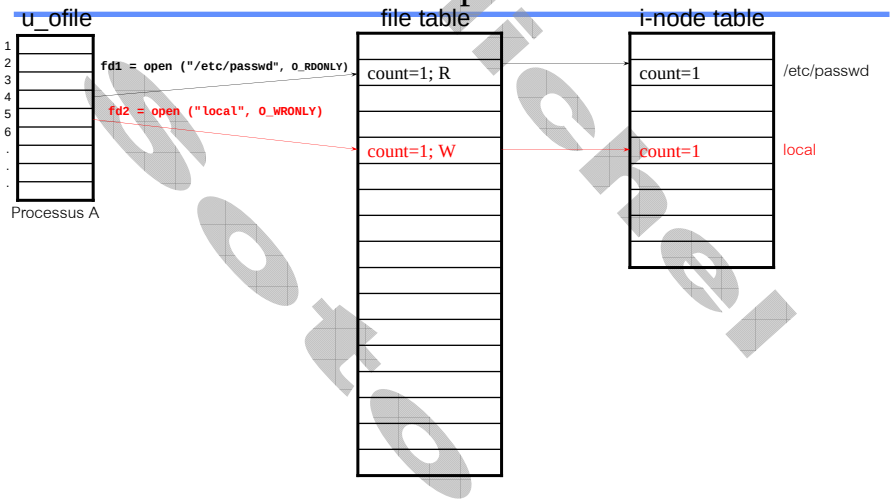
11

u_ofile, file table, i-node table exemples



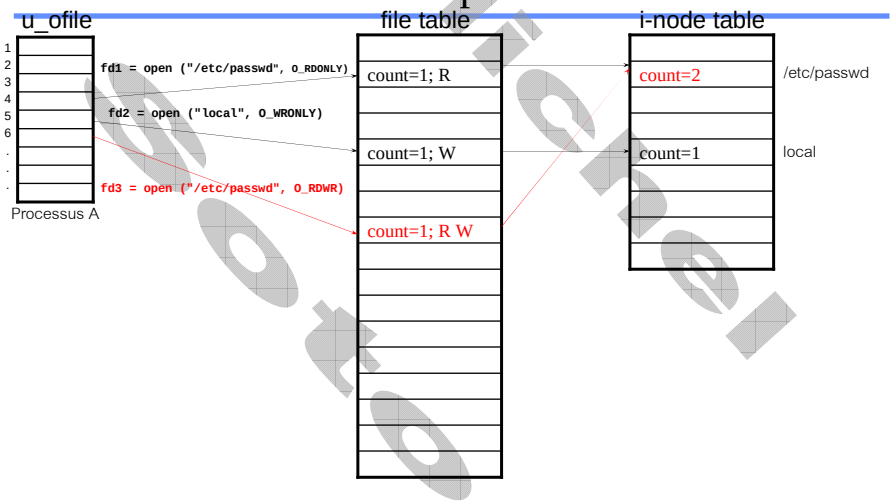
12

u_ofile, file table, i-node table exemples



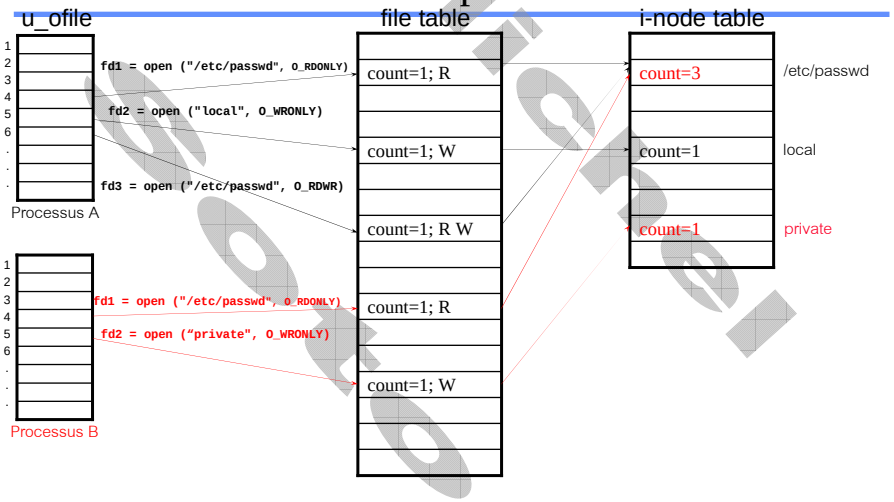
13

u_ofile, file table, i-node table exemples



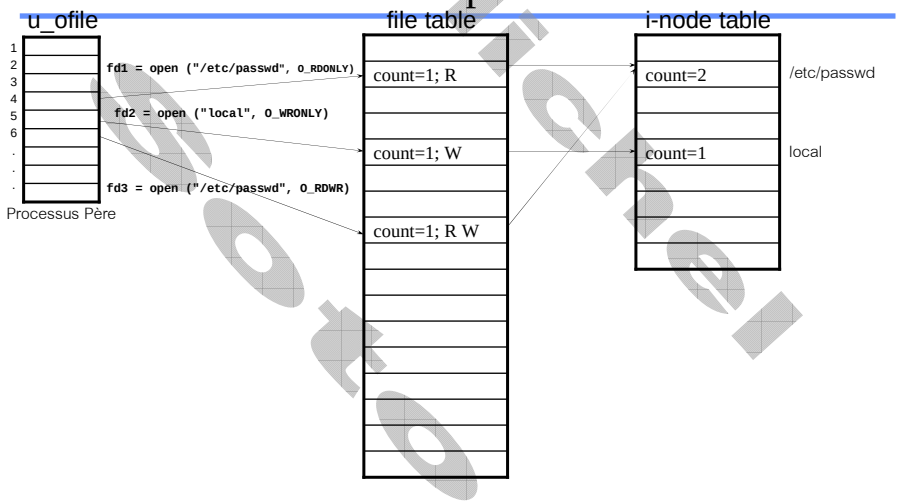
14

u_ofile, file table, i-node table exemples



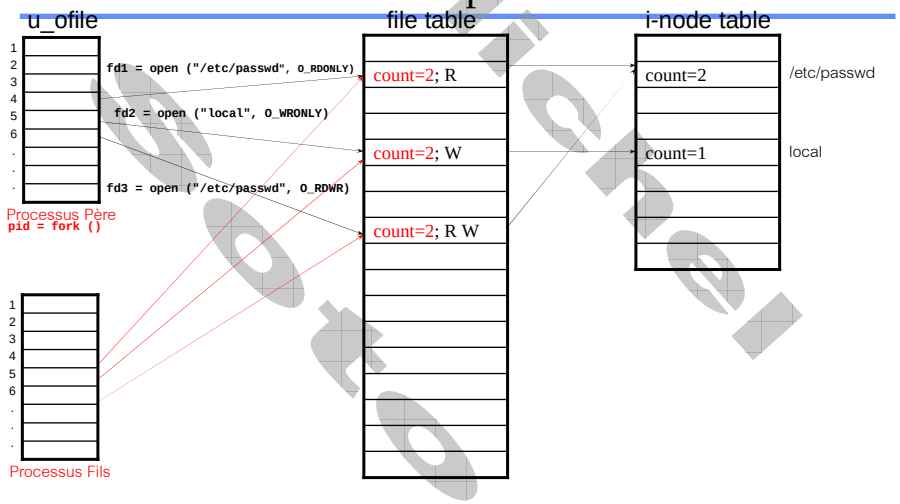
15

u_ofile, file table, i-node table exemples



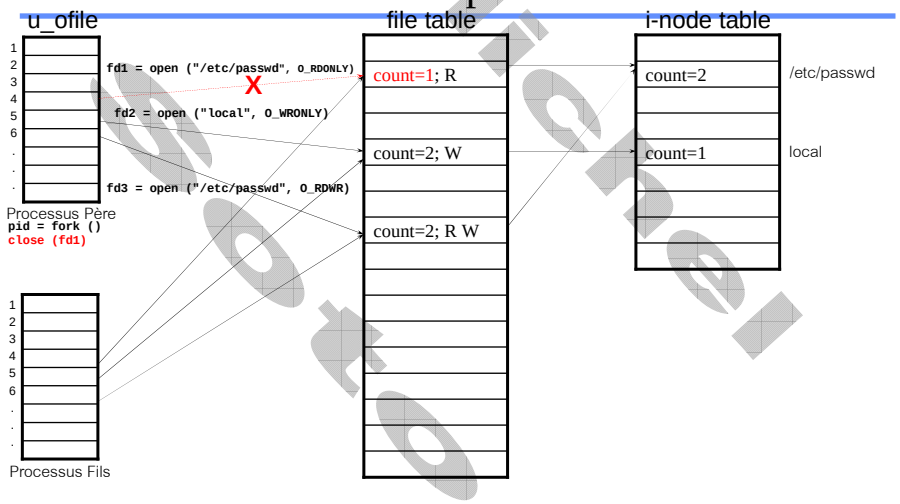
16

u_ofile, file table, i-node table exemples



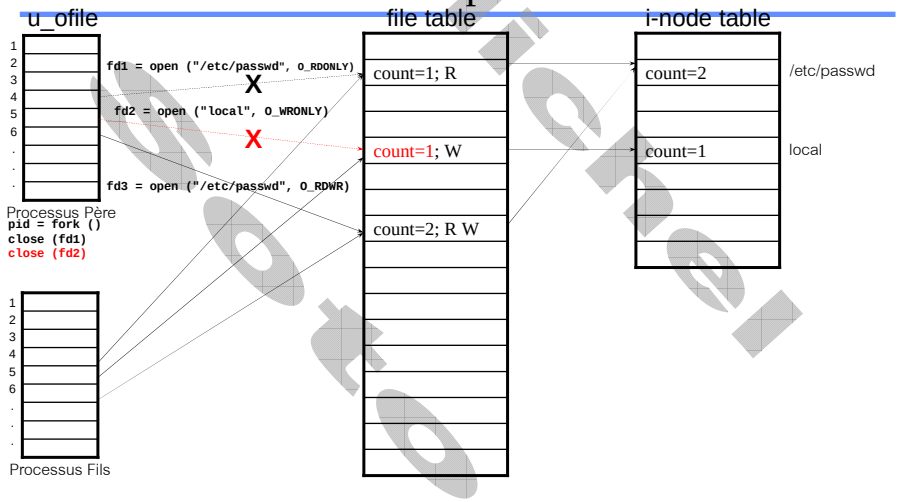
17

u_ofile, file table, i-node table exemples



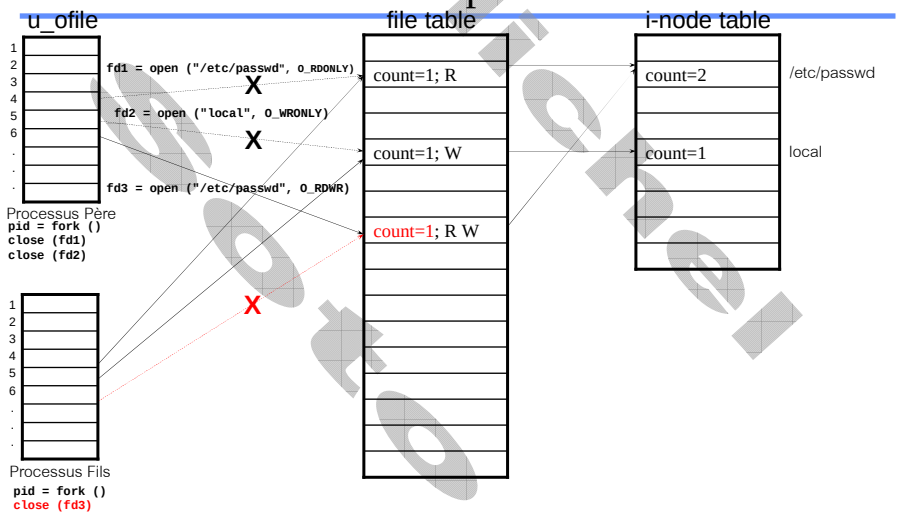
18

u_ofile, file table, i-node table exemples



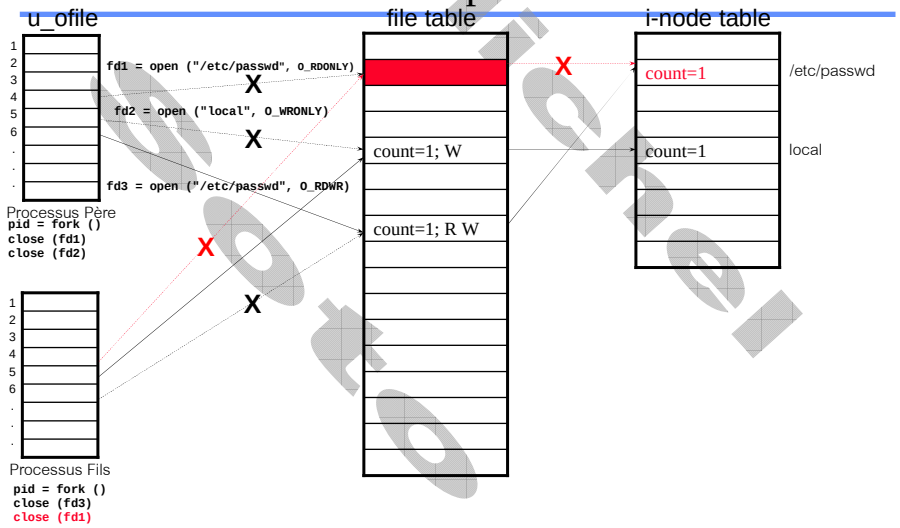
19

u_ofile, file table, i-node table exemples



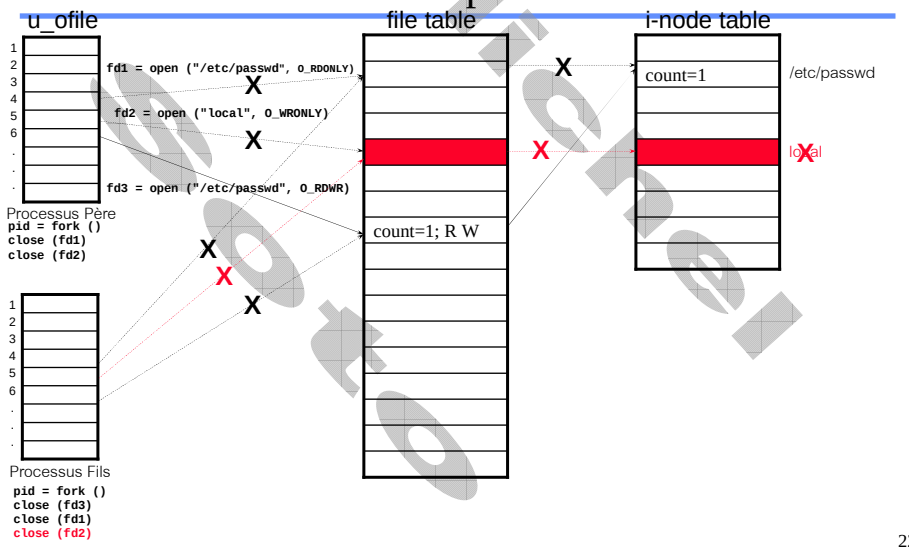
20

u_ofile, file table, i-node table exemples



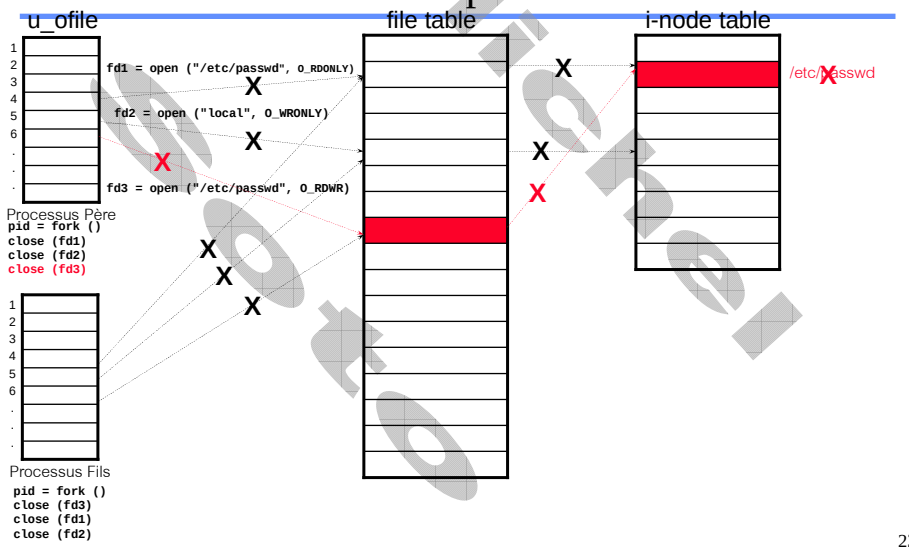
21

u_ofile, file table, i-node table exemples



22

u_ofile, file table, i-node table exemples



23

Types de fichiers

Le type du fichier détermine les opérations possibles

- fichier régulier ou ordinaire:
  - non structuré
    - contenu: texte, binaire, image, document, etc.
- répertoire:
  - noeud de l'arborescence
    - Contenu: fichiers réguliers, répertoires
- FIFO (tube)
  - communication unidirectionnelle entre processus d'une même machine

24

Types de fichiers (suite)

- socket AF_UNIX,
  - communication bidirectionnelle entre processus d'une même machine
- lien symbolique.
  - contenu : un nom de fichier
- fichier spécial
  - périphérique
    - mode bloc: disque
    - mode caractère: clavier, écran

Le type de fichier est encodé dans le champs `st_mode` de la structure `stat`.

25

Les fichiers

- Décrits par des attributs (contenus dans l'inode)
- Obtention des attributs d'un fichier

```
#include <sys/stat.h>
#include <unistd.h>
    int stat(const char *file_name, struct stat *buf);
    int fstat(int filedes, struct stat *buf);
    int lstat(const char *file_name, struct stat *buf);
```

struct stat

```
dev_t     st_dev;       /* device file resides on */
ino_t     st_ino;       /* the file serial number */
mode_t    st_mode;      /* file mode */
nlink_t   st_nlink;     /* number of hard links to the file */
uid_t     st_uid;       /* user ID of owner */
gid_t     st_gid;       /* group ID of owner */
dev_t     st_rdev;      /* the device identifier (special files only)*/
off_t     st_size;      /* total size of file, in bytes */
time_t    st_atime;     /* file last access time */
time_t    st_mtime;     /* file last modify time */
time_t    st_ctime;     /* file last status change time */
long      st_blksize;   /* preferred blocksize for file system I/O*/
long      st_blocks;    /* actual number of blocks allocated */
```

26

Macros de détermination du type d'un fichier

- Macros définies dans `types.h`

`S_ISREG()`: fichier régulier
`S_ISDIR()`: fichier répertoire
`S_ISCHR()`: fichier spécial caractère
`S_ISBLK()`: fichier spécial bloc
`S_ISFIFO()`: FIFO
`S_ISLNK()`: lien symbolique
`S_ISSOCK()`: socket

L'argument de chacune des macros est le champs `st_mode` de la structure `stat`.

27

Exemple

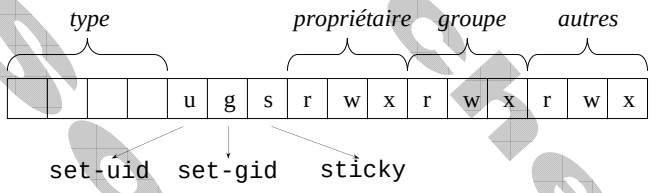
```
#include <sys/types.h>
…
main (int argc, char *argv[]) {
   struct stat buf; char *ptr;
   …
   if (stat(argv[1], &buf) != 0) { perror("Echec stat : "); exit(0); }
   …
   if (S_ISREG(buf.st_mode)) ptr = "regular";
   else if (S_ISDIR(buf.st_mode)) ptr = "directory";
   else if (S_ISCHR(buf.st_mode)) ptr = "character special";
   else if (S_ISBLK(buf.st_mode)) ptr = "bloc special";
   else if (S_ISFIFO(buf.st_mode)) ptr = "fifo";
   else if (S_ISLNK(buf.st_mode)) ptr = "symbolic link";
   else if (S_ISSOCK(buf.st_mode)) ptr = "socket";
   else ptr = "** unknown mode **";

   printf("%s\n", ptr);
   exit(0);
}
```

28

Droits d'accès aux fichiers

- Champs `st_mode`



mode	*Description*
`S_ISUID`	set_user_ID on execution
`S_ISGID`	set-group-ID on execution
`S_ISVTX`	saved-text (sticky bit)
`S_IRWXU`	read, write, and execute by user (owner)
`S_IRUSR`	read by user (owner)
`S_IWUSR`	write by user (owner)
`S_IXUSR`	execute by user (owner)

mode	*Description*
`S_IRWXG`	read, write, and execute by group
`S_IRGRP`	read by group
`S_IWGRP`	write by group
`S_IXGRP`	execute by group
`S_IRWXO`	read, write, and execute by other (world)
`S_IROTH`	read by other (world)
`S_IWOTH`	write by other (world)
`S_IXOTH`	execute by other (world)

29

Test des droits d'accès

- Chaque fois qu'un processus ouvre, crée ou supprime un fichier, le noyau procède aux tests suivants :

1. si ID user effectif du processus est 0 (superuser) l'accès est autorisé,
2. si ID user effectif du processus est égal à ID propriétaire du fichier :
   a. si le mode d'accès correspond aux droits d'accès propriétaire, l'accès est autorisé,
   b. sinon l'accès est refusé,
3. si ID groupe effectif du processus, ou l'un des IDs groupes supplémentaires, du processus est égal à ID groupe du fichier :
   a. si le mode d'accès correspond aux droits d'accès du groupe, l'accès est autorisé,
   b. sinon l'accès est refusé,
4. si le mode d'accès correspond aux droits d'accès des autres, l'accès est autorisé, sinon l'accès est refusé.

30

Test d'accessibilité

- Test d'accessibilité basé sur les IDs utilisateur et de groupe réels

```
#include <unistd.h>
   int access (const char *pathname, int mode);
```

mode	*Description*
R_OK	test for read permission
W_OK	test for write permission
X_OK	test for execute permission
F_OK	test for existence of file

31

Les appels système de base

- Ouverture

```
#include <unistd.h>
   int open(const char *pathname, int flags [, mode_t mode]);
```

  - Retourne un descripteur local de fichier
  - `flags` : `O_RDONLY, O_WRONLY, O_RDWR | O_CREAT, O_EXCL, O_APPEND, O_TRUNC, O_NONBLOCK, ...`
  - `mode` : permissions si un nouveau fichier est créé (modifiées par le `umask`).
- Lecture/Ecriture

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

  - Retournent le nombre d'octets effectivement lus/écrits
- Fermeture

```
int close(int fd);
```


Manipulation de l'*offset* et duplication de descripteurs

- Déplacement du pointeur courant d'un fichier régulier

```
#include <unistd.h>
   off_t lseek(int fildes, off_t offset, int whence);
```

  - `whence` : `SEEK_SET (0)` par rapport au début du fichier
    `SEEK_CUR (1)` par rapport à la position courante
    `SEEK_END (2)` par rapport à la fin du fichier
  - `offset` : position par rapport à `whence`
- Duplication des descripteurs de fichiers

```
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

  - Utilisés pour rediriger les entrées/sorties standards vers des fichiers ou vers des tubes.

Le verrouillage - Caractéristiques

- Le verrouillage s'applique au fichier (et non au descripteur)
- Un verrou est associé à un processus et à un fichier
  - seul le propriétaire du verrou peut le modifier ou le supprimer
  - lorsqu'un processus se termine, tous les verrous qu'il détient sont supprimés
  - chaque fois qu'un descripteur est fermé par un processus, tous les verrous sur le fichier référencé par le descripteur pour le processus donné sont supprimés
- Héritage des verrous
  - les verrous ne sont jamais hérités par les processus fils à travers un *fork*
  - les verrous peuvent être hérités par un nouveau programme au travers d'un *exec* (cas de SVR4 et 4.3+BSD, POSIX.1 ne le requiert pas)
- La portée du verrou

Le verrouillage - Compatibilité

- Le type du verrou
  - partagé (*shared*) : plusieurs verrous de ce type peuvent cohabiter
  - exclusif (*exclusive*) : un verrou de ce type ne peut cohabiter avec aucun autre verrou (*exclusif* ou *partagé*)

La région possède déjà \ *Requête pour*	verrou partagé	verrou exclusif
aucun verrou	OK	OK
un ou plusieurs verrous partagés	OK	interdit
un verrou exclusif	interdit	interdit

Compatibilité entre les types de verrou

Le verrouillage impératif

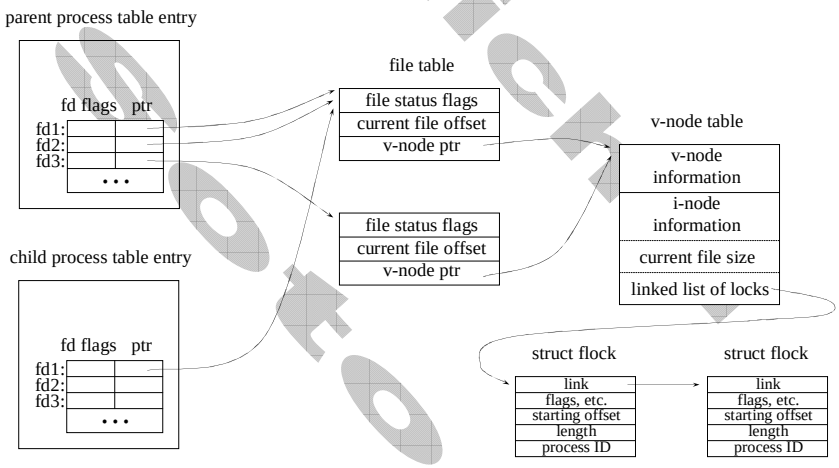
- Le mode opératoire du verrou
  - coopératif/consultatif (*advisory*) : pas d'influence sur le E/S
  - impératif (*mandatory*) : influence sur les E/S

	Descripteur bloquant, tentative de		*Descripteur non bloquant, tentative de*	
	lecture	*écriture*	*lecture*	*écriture*
un verrou partagé existe sur la région	OK	bloquée	OK	`EAGAIN`
un verrou exclusif existe sur la région	bloquée	bloquée	`EAGAIN`	`EAGAIN`

Effets du verrouillage impératif sur les lectures et écritures des autres processus

- L'indication du mode opératoire est mémorisée dans les i–noeuds

Implémentation 4.3+BSD



Les différentes formes de verrouillage

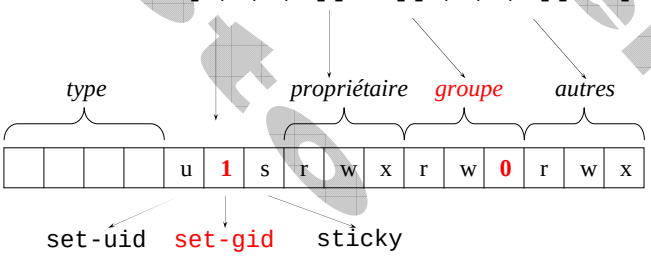
System	Advisory	Mandatory	`fcntl`	`lockf`	`flock`
POSIX.1	•		•		
XPG3	•		•		
SVR2	•		•	•	
SVR3, SVR4	•	•	•	•	
4.3BSD	•				•
4.3BSD Reno	•		•		•

Advanced Programming in the UNIX Environment, W. Richard Stevens, Addison-Wesley Professional Computing Series

`flock` : le verrouillage s'applique à la totalité du fichier
Verrouillage plus fin, voir `fcntl()`

Mise en œuvre du verrouillage impératif

- Doit être activé à la fois:
  - sur le système de fichiers
    - au moment du `mount (option mand)`
  - sur fichier à verrouiller
    - en supprimant le droit d'exécution du groupe
    - positionnant le bit `set-gid`
      - `chmod [2,3,6,7][0-7][0,2,4,6][0-7]`



Modification des caractéristiques d'un fichier

- La primitive `fcntl`

```
#include <unistd.h>
#include <fcntl.h>
int fcntl(int fd, int cmd);
    int fcntl(int fd, int cmd, … /* struct flock *flockptr */);
```

La primitive `fcntl` permet, en fonction de la valeur du paramètre `cmd`, de réaliser un certain nombre d'opérations sur le descripteur de fichier `fd`.

- Valeurs de `cmd`

POSIX.1

		Valeur retour
– F_DUPFD	/* duplication de descripteur */	nouveau descripteur
– F_GETFD, F_SETFD	/* modification des attributs du descripteur */	état de l'attribut
– F_GETFL, F_SETFL	/* consultation/modification du mode d'ouverture */	état des attributs
– F_GETLK, F_SETLK, F_SETLKW	/* manipulation des verrous */	
– F_GETOWN, F_SETOWN (BSD)	/* modification du propriétaire d'une socket */	propriétaire du fichier

40

Verrouillage par la primitive `fcntl`

- Gérer les accès concurrents (sur des portions de fichier)

Le troisième paramètre de la primitive est un pointeur sur une `struct flock`.

```
struct flock {  /* défini dans fcntl.h */
   short l_type; /* type de verrou : F_RDLCK partagé
                                       F_WRLCK exclusif
                                       F_UNLCK déverrouillage
                 */
   short l_whence; /* position (idem lseek) */
   short l_start;  /* position relative de début : l_whence */
   short l_len;    /* nombre d'octets verrouillés, si 0 →
                      jusqu'à fin fichier */
   int l_pid;      /* PID du processus auquel appartient le
                      verrou. Retourné par F_GETLK */
}
```

- Les situations d'interblocage sont détectées.

41

Exemple de verrouillage

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

main() {
   struct flock verrou;
   int fd;
   fd = open("mon-fichier", O_RDWR);
   verrou.l_type = F_WRLCK;   /* verrou exclusif */
   verrou.l_whence = SEEK_SET;/* par rapport au début du fichier */
   verrou.l_start = 0;        /* 0 par rapport au début du fichier */

// verrouillage de tout le fichier
   verrou.l_len = lseek(fd, 0, SET_END);/* récupérer la taille du fichier */

   while (fcntl(fd, F_SETLK, &verrou) == -1) {
          if (errno == EACCES || errno == EAGAIN) continue;
   }
   /* accès au fichier */
  ...
}
```

42

Algorithme de lecture

dans un fichier ordinaire avec la primitive `read`

1. Il n'existe pas de verrou exclusif impératif sur le fichier dans la portée de la lecture
   a. Si non fin de fichier, lecture du nombre spécifié de caractères ou jusqu'à la fin du fichier
   b. Si fin de fichier, aucune lecture (`0` en retour)

2. Il existe un verrou exclusif impératif sur le fichier dans la portée de la lecture
   a. Si mode de lecture bloquant (`O_NONBLOCK` et `O_NDELAY` non positionnés), processus bloqué jusqu'à suppression du verrou ou réception signal
   b. Si mode de lecture non bloquant (`O_NONBLOCK` ou `O_NDELAY` positionnés), retour immédiat et pas de lecture (`-1` en retour et `errno` = `EAGAIN`)

43

Algorithme d'écriture

dans un fichier ordinaire avec la primitive `write`

1. Il n'existe pas de verrou impératif (partagé ou exclusif) sur le fichier dans la portée de l'écriture
   a. Écriture dans le fichier (nombre caractères écrits en retour)
   b. Si l'indicateur `O_SYNC` non positionné alors écriture dans le cache du noyau, sinon écriture sur le disque

2. Il existe un verrou impératif (partagé ou exclusif) sur le fichier dans la portée de l'écriture
   a. Si mode d'écriture bloquant (`O_NONBLOCK` et `O_NDELAY` non positionnés), processus bloqué jusqu'à suppression du verrou ou réception signal
   b. Si mode d'écriture non bloquant (`O_NONBLOCK` ou `O_NDELAY` positionnés), retour immédiat et pas d'écriture (`-1` en retour et `errno` = `EAGAIN`)

Modification de l'attribut d'un descripteur de fichier

- Fermeture d'un descripteur lors d'un recouvrement (`exec`)

  La valeur, *par défaut*, de l'indication (`FD_CLOEXEC`) de fermeture automatique ou de maintien d'ouverture d'un descripteur lors d'un recouvrement correspond à un *maintien de l'ouverture*.

Exemple

```
int fd, attr_fd;
…
attr_fd = fcntl(fd, F_GETFD);   /* récupère les attributs du descripteur fd*/
attr_fd = attr_fd | FD_CLOEXEC; /* positionnement de l'indicateur */
fcntl(fd, F_SETFD, attr_fd);    /* maj des nouveaux attributs */
…
```

Modification des attributs d'états d'un fichier

- Rendre les E/S non bloquantes

  La valeur, *par défaut*, de l'indication (`O_NDELAY` ou `O_NONBLOCK` - POSIX) d'E/S en mode bloquant ou non bloquant correspond à un *mode bloquant*.

Exemple

```
int mode_courant, mode_non_bloquant;
...
/* récupération de l'état de stdin */
mode_courant = fcntl(stdin, F_GETFL);
/* modification du mode des entrées standards */
mode_non_bloquant = mode_courant | O_NONBLOCK;
/* forcer les entrées standards à être réalisées en mode non bloquant */
fcntl(stdin, F_SETFL, mode_non_bloquant);
...
```


46

Modification des attributs d'états d'un fichier (2)

- Forcer (après coup) les écritures à se faire en fin de fichier

  La valeur, *par défaut*, de l'indication (`O_APPEND`) d'*écriture en mode ajout* en fin de fichier ou non correspond à ce dernier mode.

Exemple

```
int fd, ajout_pos_courante, ajout_fin;
...
/* récupération du mode d'ouverture */
ajout_pos_courante = fcntl(fd, F_GETFL);
/* modification du mode d'écriture */
ajout_fin = ajout_pos_courante | O_APPEND;
/* forcer les écritures à être réalisées en fin de fichier */
fcntl(fd, F_SETFL, ajout_fin);
...
```


47

Modification des attributs d'états d'un fichier (3)

- Modification du mode de synchronisation des écritures

  La valeur, *par défaut*, de l'indication (`O_SYNC`) d'écriture en mode synchrone ou asynchrone correspond à un *mode asynchrone* (`O_ASYNC`) .

Autres opérations

- Reconnaissance des signaux `SIGIO` (présence d'E/S) et `SIGURG` (message urgent sur `socket`)
  - `cmd` : `F_GETOWN` /* obtient le PID du propriétaire */
    `F_SETOWN` /* établit la possession d'une `socket` */
- Duplication de descripteur
  - `cmd` : `F_DUPFD`

48

Manipulation des répertoires

- Ouverture de répertoire

```
#include <dirent.h>
    DIR *opendir(const char *pathname);
```

  - Ouvre le répertoire référencé par `pathname` en lecture et alloue un objet de type `DIR` dont l'adresse est renvoyée en retour.
  - Retourne un pointeur ou `NULL` en cas d'erreur.

- Fermeture de répertoire

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

  - Libère les ressources allouées lors de l'appel à `opendir`.
  - Retourne 0 en cas de succès et -1 en cas d'erreur.

49

Manipulation des répertoires (2)

- Lecture d'une entrée de répertoire

```
#include <dirent.h>
   struct dirent *readdir(DIR *dp);
```

  - Lecture de l'entrée suivante dans le répertoire référencé par `dp`.
  - Retourne un pointeur ou `NULL` en fin de fichier et en cas d'erreur.

```
struct dirent {
       ino_t d_ino;                /* i-node number */
       off_t d_off;                /* offset to this dirent */
       unsigned short d_reclen;    /* length of this d_name */
       char d_name [NAME_MAX+1];   /* null-terminated filename */
       unsigned short d_type       /* type */
}
```

- Repositionnement du pointeur de lecture

```
#include <dirent.h>
   void rewinddir(DIR *dp);
```


Exemple

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

main(int argc, char **argv) {
   DIR *dp;
   struct dirent *dirp;

   if ( (dp = opendir (argv[1])) == NULL )
       printf("Erreur Ouverture\n");

   while ( (dirp = readdir(dp)) != NULL ) {
       if ( strcmp(dirp->d_name, ".") == 0 || strcmp(dirp->d_name, "..") == 0 )
       continue;

       printf("fichier trouvé : %s\t de type : %d\n", dirp->d_name, dirp->d_type);
   }

closedir(dp);
}
```

Création et suppression de répertoires

- Création de répertoire

```
#include <sys/types.h>
#include <sys/stat.h>
   int mkdir(const char *pathname, mode_t mode);
```

  - Crée un nouveau répertoire vide. Les entrées "`.`" et "`..`" sont automatiquement créés.
  - Retourne 0 en cas de succès et -1 en cas d'erreur.

- Suppression de répertoire (vide)

```
#include <unistd.h>
   int rmdir(const char *pathname);
```

  - Retourne 0 en cas de succès et -1 en cas d'erreur.

Propriété des nouveaux fichiers et répertoires

Les règles de la propriété d'un nouveau répertoire sont identiques à celles de la propriété d'un nouveau fichier.

- L'identifiant utilisateur (`UID`) d'un nouveau fichier est établi à l'identifiant utilisateur effectif (`EUID`) du processus.
- `POSIX.1` permet à toute implémentation de choisir l'une des deux options suivantes pour déterminer l'identifiant de groupe (`GID`) d'un nouveau fichier :
  - l'identifiant de groupe d'un nouveau fichier peut être l'identifiant de groupe effectif (`EGID`) du processus,
  - l'identifiant de groupe d'un nouveau fichier peut être l'identifiant de groupe (`GID`) du répertoire dans lequel le fichier est créé.

`SVR4`
`set-group-ID` du répertoire parent positionné → identifiant de groupe du répertoire parent
sinon → identifiant de groupe effectif du processus
(Héritage du `set-gid-bit` du parent pour le nouveau répertoire suite à un `mkdir(…)`)
`4.3+BSD`
Utilise toujours l'identifiant de groupe du répertoire parent

Communication par tubes

- Tube : mécanisme de communication appartenant au système de fichiers.
  → nœud (type : `S_IFIFO`), descripteur, `read`, `write`, …
- Canal unidirectionnel (une entrée, une sortie)
  → deux entrées dans la table des fichiers ouverts
- Lecture destructrice
- Communication d'un flot continu de caractères
- Gestion en mode FIFO
- Capacité finie (nombre d'adresses directes) → tube plein
- Nombres de lecteurs/écrivains

54

Tubes ordinaires

- Compteur de liens nul (aucune référence à ce nœud)
- Supprimé lorsque aucun processus ne l'utilise
- Impossibilité d'ouvrir un tube (pas de `open()`)
- Connaissance de l'existence → possession d'un descripteur
  (création ou héritage)
- Communication entre processus ayant un ancêtre commun
- Perte d'accès à un tube irréversible

55

Création d'un tube ordinaire

- La primitive de création `pipe()`

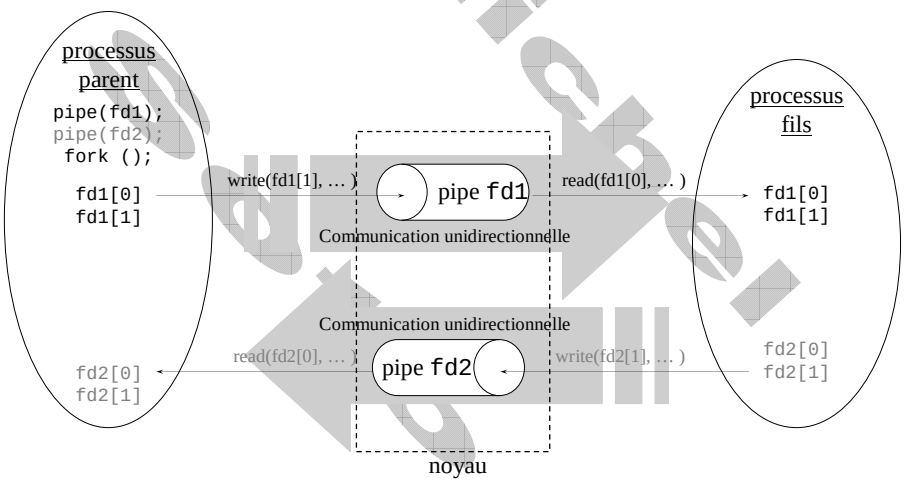
```
#include <unistd.h>
   int pipe(int filedes[2]);
```

  - alloue un nœud, deux entrées dans la table des fichiers ouverts et deux descripteurs dans la table du processus appelant,
  - retourne 0 en cas de succès et -1 sinon.

- Manipulation de tubes ordinaires
  - `read`, `write`, `close`, `fstat`, `fcntl`

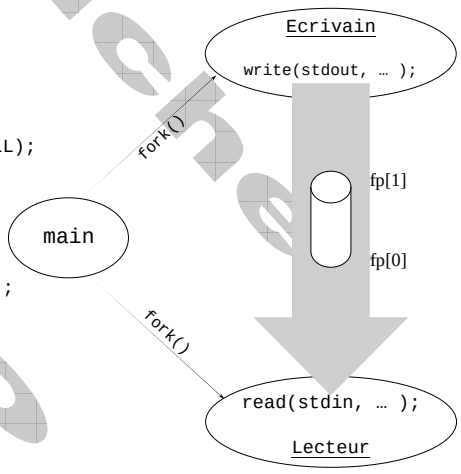
  Opération interdite : `lseek`

Héritage des descripteurs d'un tube



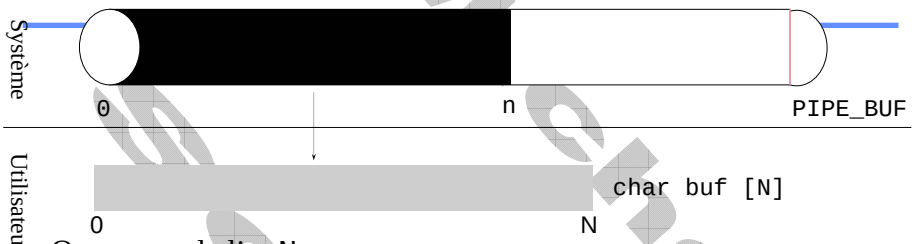
Exemple d'utilisation d'un tube ordinaire

```
main(void)
{
  int fp[2];
  pipe(fp);
  if (fork() == 0) {
    close(1);
    dup2(fp[1], 1);
    close(fp[0]); close(fp[1]);
    execl("Ecrivain", "Ecrivain", NULL);
    exit(1);
  }
  if (fork() == 0) {
    close(0);
    dup2(fp[0],0);
    close(fp[0]); close(fp[1]);
    execl("Lecteur", "Lecteur", NULL);
    exit(1);
  }
  close(fp[0]); close(fp[1]);
  wait(&ret1);
  wait(&ret2);
}
```



58

Algorithme de lecture dans un tube



On essaye de lire N octets :

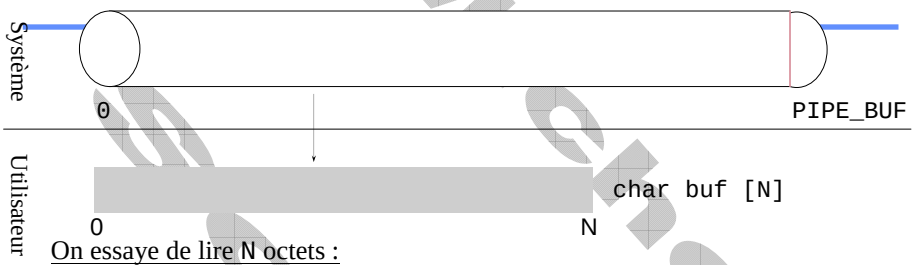
1. Le tube n'est pas vide et contient n octets

   min(n, N) octets sont lus,

   la primitive renvoie le nombre réel d'octets lus.

59

Algorithme de lecture dans un tube

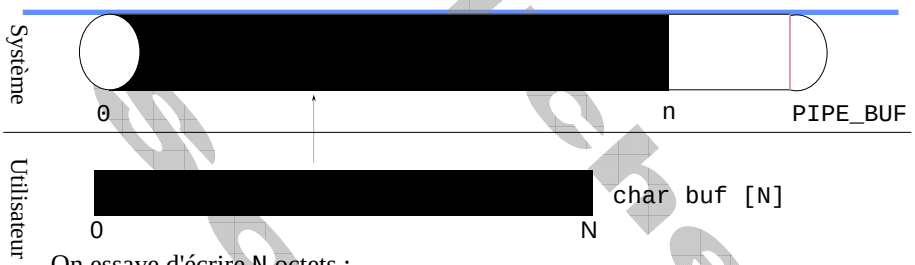


On essaye de lire N octets :

2. Le tube est vide
   a. Le nombre d'écrivains est nul (la fin de fichier est atteinte)
      - aucun octet n'est lu,
      - la primitive renvoie 0.
   b. Le nombre d'écrivains n'est pas nul
      - Si la **lecture** est **bloquante**, le processus est mis en sommeil jusqu'à ce que le tube ne soit plus vide.
      - Si la **lecture** n'est **pas bloquante**, la primitive renvoie -1 et `errno` = `EAGAIN`.

60

Algorithme d'écriture dans un tube

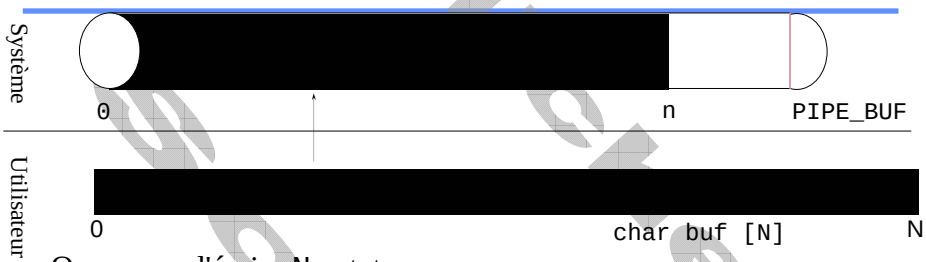


On essaye d'écrire N octets :

1. Le nombre de lecteurs est nul
   a. le signal `SIGPIPE` est délivré au processus écrivain,
2. Le nombre de lecteurs est non nul
   a. si **l'écriture** est **bloquante** et `N <= PIPE_BUF`
      - Retour de la primitive une fois les N octets écrits de façon **atomique**,
        Le processus peut, éventuellement, être mis en sommeil dans l'attente que le tube se vide suffisamment pour contenir les N octets.

61

Algorithme d'écriture dans un tube



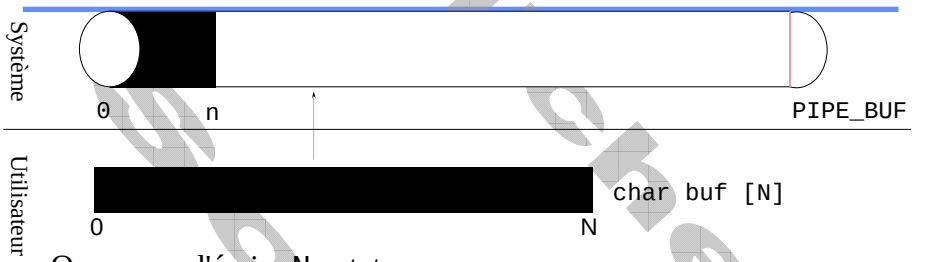
On essaye d'écrire N octets :

2. Le nombre de lecteurs est non nul
   b. si **l'écriture** est **bloquante** et `N > PIPE_BUF`
      - Retour de la primitive une fois les N octets écrits de façon **NON atomique**,

        Les données peuvent être entrelacées avec les données d'autres processus écrivains

62

Algorithme d'écriture dans un tube

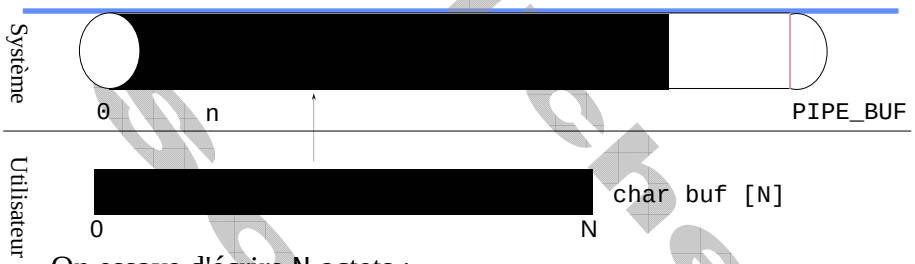


On essaye d'écrire N octets :

2. Le nombre de lecteurs est non nul
   c. si **l'écriture** est **non bloquante**
      - Si $N \leq$ `PIPE_BUF`
      - s'il y a au moins N octets libres dans le tube, une écriture atomique est réalisée

63

Algorithme d'écriture dans un tube



On essaye d'écrire N octets :

2. Le nombre de lecteurs est non nul
   c. si **l'écriture** est **non bloquante**
      - Si N <= PIPE_BUF
      - s'il y a moins de N octets libres dans le tube, aucune écriture n'est réalisée et la primitive retourne -1.

64

Les tubes nommés

- Référence dans le système de fichiers
- Suppression lorsque aucun lien physique et aucun lien interne
- Ouverture du tube avant accès (bloquante par défaut → synchronisation)
- Communication entre processus sans lien de parenté
- Libération des ressources système à la disparition du descripteur

65

Création d'un tube nommé

- Les primitives de création d'un tube nommé

```
#include <sys/types.h>
#include <sys/stat.h>
   int mkfifo(const char *pathname, mode_t mode);
   int mknod(const char *pathname, mode_t mode |
             S_IFIFO, dev_t dev);

   mkfifo [p] [-m mode] pathname
```

- Manipulation des tubes ordinaires
  - `open, close`
  - `read, write`
  - `unlink`

66

Ouverture d'un tube nommé

Demande d'ouverture d'un tube par un processus ayant les droits correspondants.

1. Si l'ouverture est bloquante → synchronisation (prise de rendez-vous)
   a. Une demande d'ouverture en lecture est bloquante s'il n'y a aucun écrivain.
   b. Une demande d'ouverture en écriture est bloquante s'il n'y a aucun lecteur.
2. Si l'ouverture est non bloquante
   a. Une demande d'ouverture en lecture réussit toujours.

      Les opérations de lecture ultérieures sont non bloquantes jusqu'à demande explicite du contraire.
   b. Une demande d'ouverture en écriture échoue s'il n'y a aucun lecteur.
   c. Une demande d'ouverture en écriture réussit s'il y a au moins un lecteur.

      Les opérations d'écriture ultérieures sont non bloquantes jusqu'à demande explicite du contraire.

67

Exemple d'utilisation d'un tube nommé

```
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <stdlib.h>

int main()
{
   int pid, h1, nread;
   char msg[16];

   mkfifo("echange.txt", 0666);
   pid=fork();
   if(pid == 0) {
       h1=open("echange.txt", O_WRONLY);
       write(h1,"Bonjour !\0",9);
       close(h1);
       exit(0);
   }
   h1=open("echange.txt", O_RDWR);

   nread=read(h1,&msg,9);
   printf("Msg lu : %s\n",msg);

   close(h1);
   wait(NULL);
   exit(0);
}
```


68