

Programmation Système Répartie et Concurrente

Master 1 Informatique – MU4IN400

Cours 7 :

IPC Inter Process Communication

Mémoire Partagée, Sémaphores, Fichiers, Tubes

Yann Thierry-Mieg
Yann.Thierry-Mieg@lip6.fr

Plan

On a vu au cours précédent

- Création de processus, wait, signaux

Aujourd'hui : IPC POSIX pour la communication

- Entrées/sorties, fichier et filedescriptor
- Mémoire partagée
- Sémaphore
- Tubes et Tubes nommés
- (Files de messages)

Références :

Cours de P.Sens, L.Arantes (PR \leq 2017)

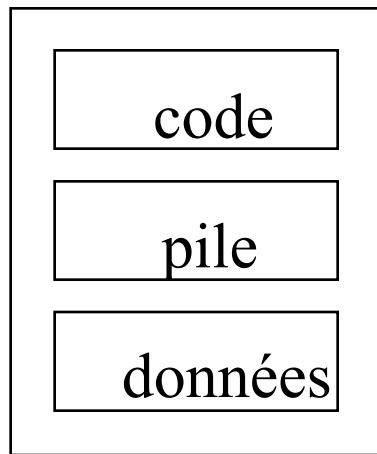
Cppreference

Le man, section 2 et 3

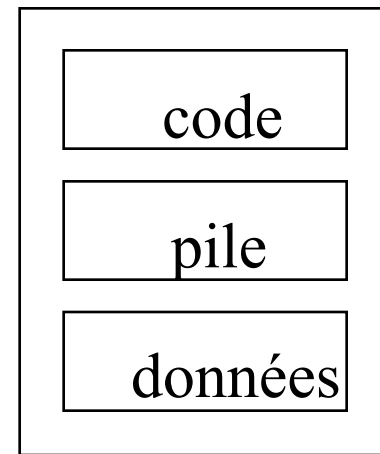
Inter Process Communication

Communication entre processus

Comment les processus peuvent communiquer, synchroniser ou partager des données?



Processus P1



Processus P2

Processus ne partagent pas leur segments de données

Communiquer *entre* processus

- Plus difficile qu'en thread :
 - Pas d'espace d'adressage commun
- Différents mécanismes utilisés (POSIX) :
 - Fichiers
 - Signaux
 - Tubes et tubes nommés (pipe)
 - Segment de mémoire partagée ou mmap
 - Files de messages
 - Sockets
- Et aussi des primitives de synchronisation inter process
 - Semaphore

Entrées Sorties POSIX

inode et fichier

Entrée Sorties

Primitives d'entrées-sorties POSIX

unistd.h, sys/stat.h, sys/types.h, fcntl.h

- ✓ Constituent l'interface avec le noyau Unix (**appels systèmes**) donc permettent l'utilisation complète des services offerts par le noyau.
- ✓ Portabilité des programmes sur Unix (Posix).

Notion d'inode

- Un objet apte à être partagé entre plusieurs processus
- Qui dispose d'une adresse unique
 - E.g. /usr/bin/gcc, /myshm
- Qui supporte des opérations simples pour lire et écrire par blocs

Inode

Un nœud d'index ou inode (contraction de l'anglais *index* et *node*) est une structure de données contenant des informations à propos d'un fichier ou répertoire.

- ✓ Chaque fichier a un seul inode, même s'il peut avoir plusieurs noms (lien physique).
- ✓ Sauvegarder dans le disque : mettre à jour la table des inodes
- ✓ Les inodes peuvent désigner d'autres objets (tubes nommés, sockets,...)

Inode

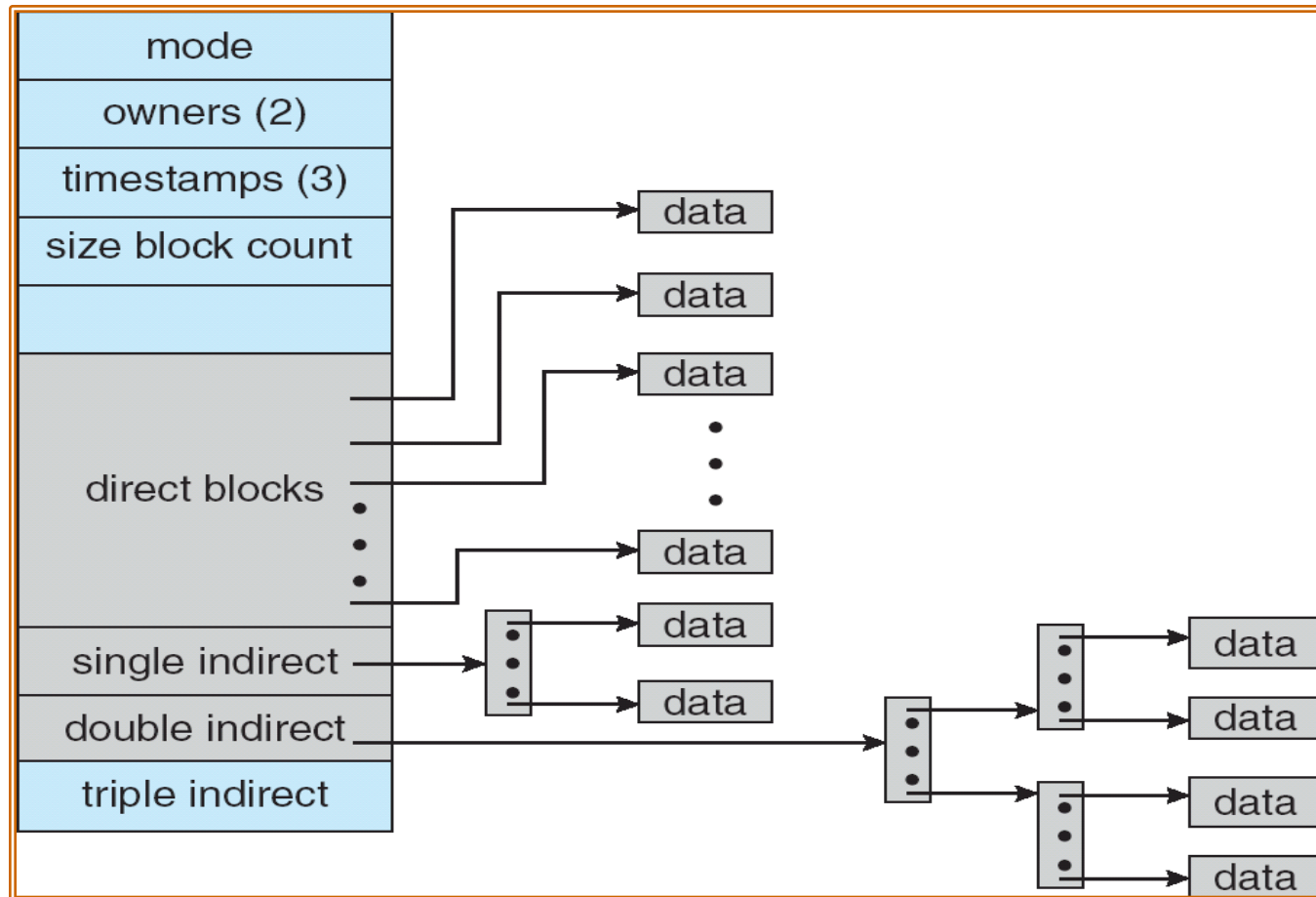
Les informations stockées dans un inode disque :

- utilisateur propriétaire,
- groupe propriétaire,
- type de fichier,
- droits d'accès,
- date de dernier accès,
- date de dernière modification,
- date de dernière modification de l'inode,
- nombre de liens,
- taille du fichier,
- adresses des blocs-disque contenant le fichier (13).

Cf. man 2 stat

Inode

NB : la structure interne précise de l'inode est à la charge du filesystem



Consultation de l'i-node (stat)

Standard POSIX : Structure stat

<sys/stat.h>

man 7 inode

```
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*/  
    off_t          st_size;         /* total size of file, in bytes */  
    unsigned long   st_blksize;      /* blocksize - file system I/O*/  
    unsigned long   st_blocks;      /* number of blocks allocated */  
    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 */  
}
```

Le couple st_dev+st_ino est unique

Type de fichier

Champ *st_mode* de *struct stat* contient deux infos :

- Les bits de droits d'accès
- Le type de l'inode : dossier, fichiers, pipe...

Type : on teste avec des macros

Fichiers réguliers de données : S_ISREG (t)

Répertoires : S_ISDIR (t)

Tubes FIFO : S_ISFIFO (t)

Fichiers spéciaux : périphériques bloc S_ISBLK (t) et caractère S_ISCHR (t)

Liens symboliques : S_ISLNK (t)

Sockets : S_ISSOCK (t)

Tous ces objets se comportent de la même manière.

Droits d'accès

Propriétaire, groupe et autres (Champ *st_mode* de *struct stat*)

✓ lecture, écriture et exécution

	Propriétaire	Groupe	Autres
Lecture	S_IRUSR	S_IRGRP	S_IROTH
Écriture	S_IWUSR	S_IWGRP	S_IWOTH
Exécution	S_IXUSR	S_IXGRP	S_IXOTH
Les trois	S_IRWXU	S_IRWXG	S_IRWXO

> ls -l

rwxr-xr--

S_IRWXU | *S_IRGRP* | *S_IXGRP* | *S_IROTH*

Exemple : stat et st_mode

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

```
int main(int argc, char **argv)
{
    if(argc != 2)
        return 1;
```

```
    struct stat fileStat;
    if(stat(argv[1], &fileStat) < 0)
        return 1;
```

```
    printf("Information for %s\n", argv[1]); }
    printf("-----\n");
    printf("File Size: \t\t%d bytes\n", fileStat.st_size);
    printf("Number of Links: \t%d\n", fileStat.st_nlink);
    printf("File inode: \t\t%d\n", fileStat.st_ino);
```

```
    printf("File Permissions: \t");
    printf( (S_ISDIR(fileStat.st_mode)) ? "d" : "-");
    printf( (fileStat.st_mode & S_IRUSR) ? "r" : "-");
    printf( (fileStat.st_mode & S_IWUSR) ? "w" : "-");
    printf( (fileStat.st_mode & S_IXUSR) ? "x" : "-");
    printf( (fileStat.st_mode & S_IRGRP) ? "r" : "-");
    printf( (fileStat.st_mode & S_IWGRP) ? "w" : "-");
    printf( (fileStat.st_mode & S_IXGRP) ? "x" : "-");
    printf( (fileStat.st_mode & S_IROTH) ? "r" : "-");
    printf( (fileStat.st_mode & S_IWOTH) ? "w" : "-");
    printf( (fileStat.st_mode & S_IXOTH) ? "x" : "-");
    printf("\n\n");
```

```
    printf("The file %s a symbolic link\n",
(S_ISLNK(fileStat.st_mode)) ? "is" : "is not");
```

```
    return 0;
```

Fonctions de consultation de l'i-node

Obtention des caractéristiques d'un fichier

- ✓ **int stat(const char *file_name, struct stat *buf);**
- ✓ **int fstat(int fdes, struct stat *buf);**
 - Résultats récupérés dans une *struct stat*

Test des droits d'accès d'un processus sur un fichier

- ✓ **int access (const char* pathname, int mode);**
 - mode : **R_OK, W_OK, X_OK, F_OK**
(droit de lecture, écriture, exécution, existence).

Exemple - stat

```
#define _POSIX_SOURCE 1
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>

int main (int argc, char* argv []) {
    struct stat stat_info;

    if ( stat (argv[1], &stat_info) == -1)
        { perror ("erreur stat");
          return EXIT_FAILURE;
        }

    if (S_ISDIR (stat_info.st_mode) )
        printf ("fichier répertoire\n");

    printf ("Taille fichier : %ld\n", (long)stat_info.st_size);

    if (stat_info.st_mode & S_IRGRP)
        printf ("les usagers du même groupe peuvent lire le fichier\n");

    return EXIT_SUCCESS;
}
```


Liens symboliques

`int symlink (const char* reference, const char* lien);`

- ✓ créer un lien symbolique sur le fichier *reference*

`int lstat (const char* reference, struct stat* pStat);`

`ssize_t readlink (const char* ref, char* tampon, size_t taille);`

- ✓ récupère à l'adresse *tampon* la valeur du lien symbolique (son contenu)

`lchmod (const char* reference, mode_t mode);`

`lchown (const char* reference, uid_t uid, gid_t gid);`

API globalement similaire, préfixée par « l » pour que les invocations suivent le lien.

Changement d'attributs d'un i-node

Droits d'accès

- ✓ **int chmod (const char* reference, mode_t mode);**
- ✓ **int fchmod (int descripteur, mode_t mode);**

attribution des droits d'accès *mode* au fichier :

- de nom *reference*
- associé à *descripteur*

Propriétaire

- ✓ **int chown (const char* reference, uid_t uid, gid_t gid);**
- ✓ **int fchown (int descripteur, uid_t uid, gid_t gid);**

modification du propriétaire *uid* et du groupe *gid* d'un fichier

code renvoi : 0 (succès) ; -1 (erreur) + utiliser perror pour interroger errno

Exemple - chmod

```
#define _POSIX_SOURCE 1
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>
```

test-chmod.c

```
int main (int argc, char* argv []) {
    if (chmod (argv[1],
        S_IRUSR | S_IWUSR |
        S_IRGRP | S_IWGRP |
        S_IROTH | S_IWOTH) == 0)
        printf ("fichier %s en lecture-ecriture pour tous les usagers \n ",
            argv[1]);
    else { perror ("chmod"); return EXIT_FAILURE; }
    return EXIT_SUCCESS;
}
```

```
>ls -l fich1
```

```
-rw----- ....
```

```
>test-chmod fich1
```

```
-rw-rw-rw- .....
```

Création et Manipulation d'inodes du système de fichiers

Primitives de base (1)

Ouverture d'un fichier : open

- ✓ **int open (const char* reference, int flags);**
- ✓ **int open (const char* reference, int flags, mode_t droits);**
 - renvoie un numéro de descripteur
 - **flags:**
 - **O_RDONLY** : ouverture en lecture
 - **O_WRONLY** : ouverture en écriture
 - **O_RDWR** : ouverture en lecture-écriture
 - **O_CREAT**: création d'un fichier s'il n'existe pas
 - **O_TRUNC**: vider le fichier s'il existe
 - **O_APPEND** : écriture en fin de fichier
 - **O_SYNC** : écriture immédiate sur disque
 - **O_NONBLOCK** : ouverture non bloquante
 - **droits:** lecture, écriture, exécution

code renvoi : descripteur int(succès) -1 (erreur)

Primitives de base (2)

Fermeture de fichier : close

✓ **int close (int descripteur);**

- Ferme le descripteur correspondant à un fichier en désallouant son entrée de la table des descripteurs du processus.
- Si nécessaire, mise à jour table des fichiers et table des i-nodes.

Création d'un fichier

✓ **int creat (const char* reference, mode_t droits);**

correspond à l'appel suivant:

`open (O_WRONLY | O_CREAT | O_TRUNC, droits);`

Organisation des Tables

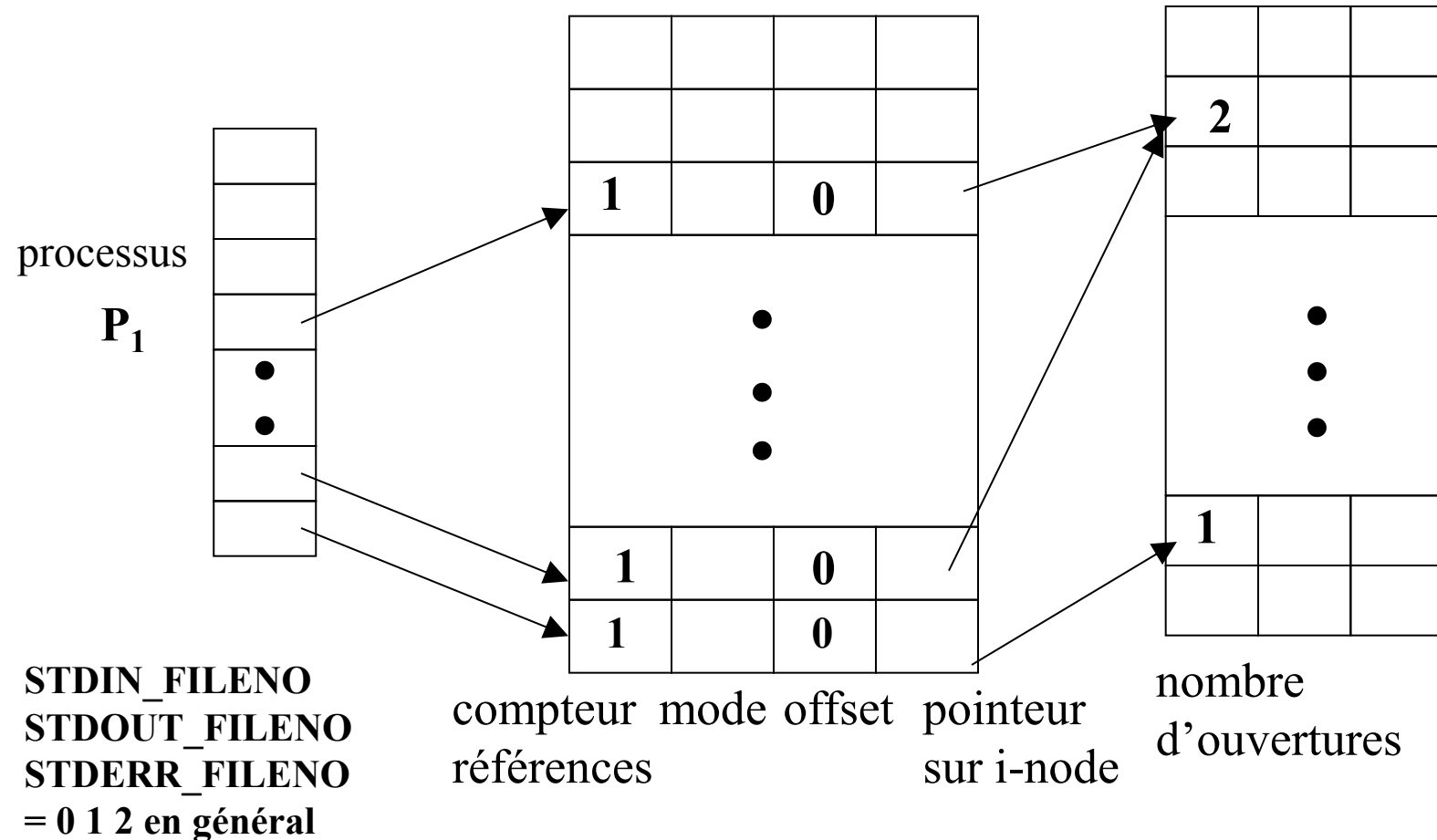
Le système maintient une table des descripteurs ouverts :

- File descriptor = index dans cette table

Table de descripteurs

Table de fichiers ouverts

Table d'i-nodes



Primitives de base (3)

Lecture dans un file descriptor : `read`, `readv`, `pread`

✓ **`ssize_t read (int desc, void* tampon, size_t nbr);`**

- Demande de lecture d'au + *nbr* caractères du fichier correspondant à *desc*.
- Les caractères lus sont écrits dans *tampon*.
- Renvoie le nombre de caractères lus ou -1 en cas d'erreur.
- La lecture se fait à partir de la **position courante**
offset de la *Table des Fichiers Ouverts* ; mise à jour après la lecture.

✓ **`ssize_t readv (int desc, const struct iovec* vet, int n);`**

- Données récupérées dans une *struct iovec* de taille *n*.

```
struct iovec {  
    void *iov_base;  
    size_t iov_len; }
```

✓ **`ssize_t pread (int desc, void* tampon, size_t nbr, off_t pos);`**

- Lecture à partir de la position *pos* ; *offset* n'est pas modifié.

Attention aux lectures partielles

- Read rend le nombre d'octets lus

```
int fullread (char *buff, size_t len, int fd) {  
    size_t lu=0, alire=len;  
    while (alire !=0) {  
        int n = read (fd, buff+lu,alire);  
        if (n < 0) return -1;  
        alire -= n;  
        lu +=n;  
    }  
    return 0;  
}
```

Primitives de base (4)

Écriture dans un fichier : `write`, `writew`, `pwrite`

- ✓ **`ssize_t write (int desc, void* tampon, size_t nbr);`**
 - Demande d'écriture de *nbr* caractères contenus à partir de l'adresse *tampon* dans le fichier correspondant à *desc*.
 - Renvoie le nombre de caractères écrits ou -1 en cas d'erreur.
 - L'écriture se fait à partir de la fin du fichier (`O_APPEND`) ou de la position courante.
 - Modifie le champ *offset* de la *Table des Fichiers Ouverts*.
- ✓ **`ssize_t writew (int desc, const struct iovec* vet, int n);`**
- ✓ **`ssize_t pwrite (int desc, void* tampon, size_t nbr, off_t pos);`**

Exemple – open, read et write

test-rw.c

```
#define _POSIX_SOURCE 1
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>

#define SIZE_TAMPON 100
char tampon [SIZE_TAMPON];
int main (int argc, char* argv []) {
    int fd1, fd2;    int n,i;

    fd1 = open (argv[1],
                O_WRONLY|O_CREAT|O_SYNC,0600);
    fd2 = open (argv[1], O_RDWR);
    if ( (fd1== -1) || (fd2 == -1)) {
        perror("open");
        return EXIT_FAILURE;
    }
}
```

```
    if (write (fd1,"abcdef", strlen ("abcdef"))
    == -1) { /* error */ }
    if (write (fd2,"123", strlen ("123") ) == -1)
    { /*error*/}
    if ((n= read (fd2,tampon, SIZE_TAMPON))
    <=0) { /*error*/ }

    for (i=0 ; i<n; i++)
        printf ("%c",tampon [i]);

    return EXIT_SUCCESS;
}
```

```
>test-rw fich2
def
>cat fich2
123def
```

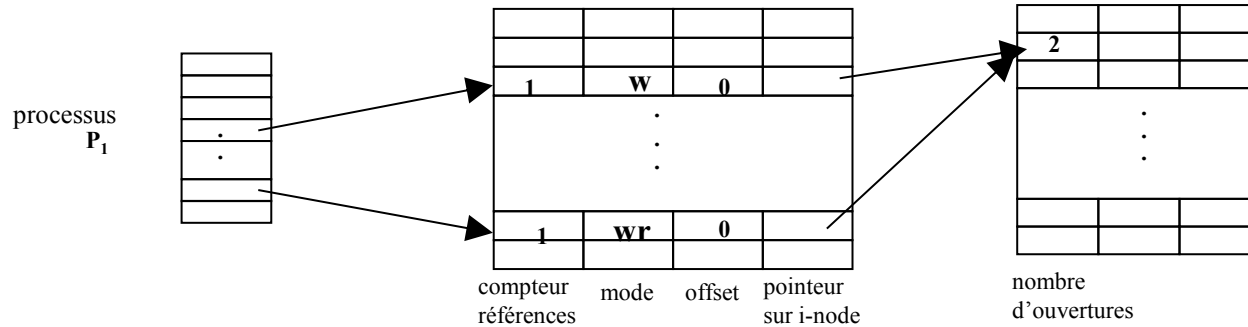
Organisation des Tables

Table de descripteurs

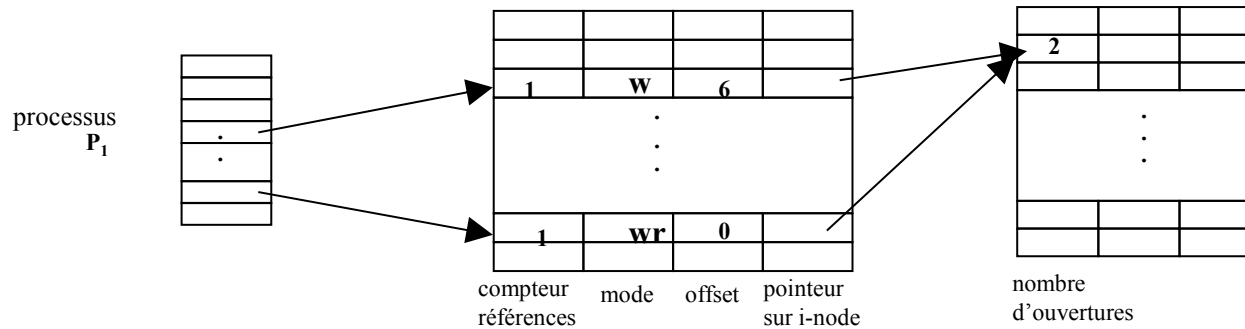
Table de fichiers ouverts

Table d'i-nodes

Après les 2 « opens »



Après le premier « write »

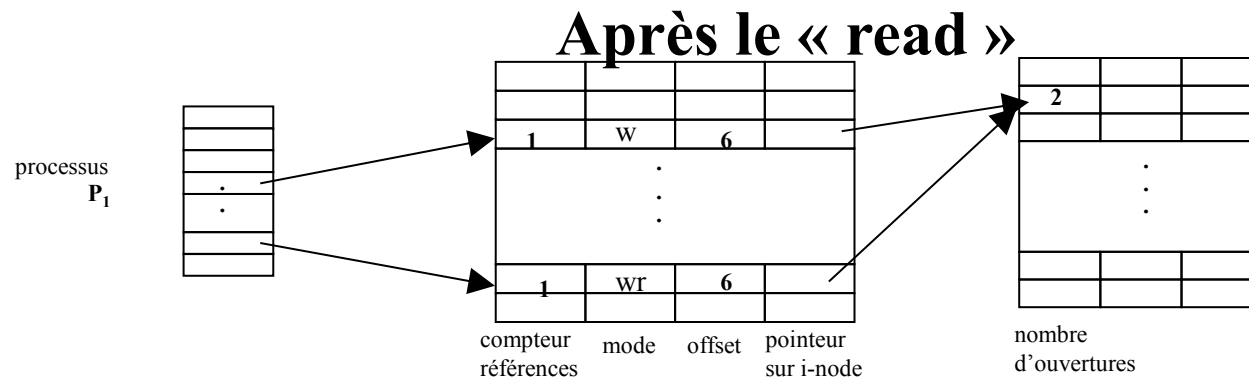
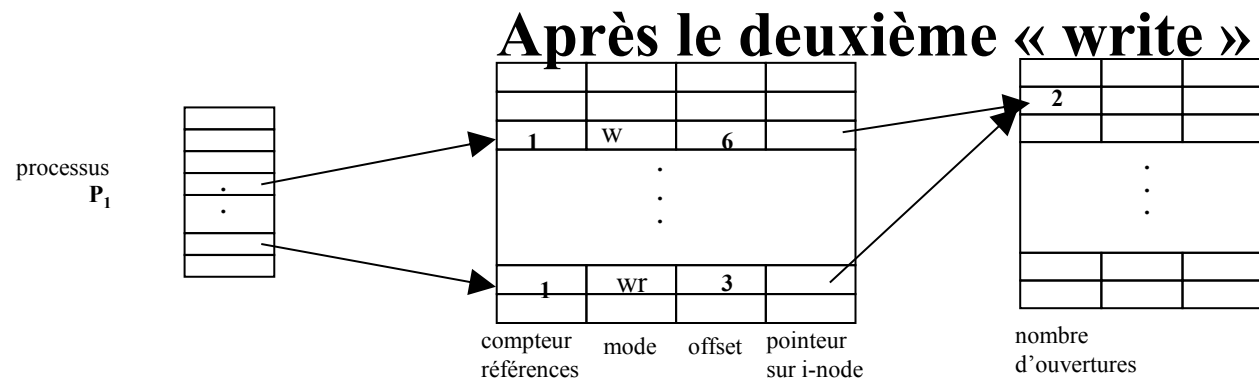


Organisation des Tables

Table de descripteurs

Table de fichiers ouverts

Table d'i-nodes



Primitives de base (5)

Manipulation de l'offset: `lseek`

- ✓ **`off_t lseek (int desc, off_t position, int origine);`**
 - Permet de modifier la position courante (*offset*) de l'entrée de la *Table de Fichiers Ouverts* associée à *desc*.
 - La position courante prend comme nouvelle valeur : *position* + *origine*.
 - **origine:**
 - **SEEK_SET**: 0 (début du fichier)
 - **SEEK_CUR** : Position courante
 - **SEEK_END** : Taille du fichier
 - Renvoie la nouvelle position courante ou -1 en cas d'erreur.

```
#define _POSIX_SOURCE 1
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>

#define SIZE_TAMPON 100
char tampon [SIZE_TAMPON];
int main (int argc, char* argv []) {
    int fd1, fd2;    int n,i;

    fd1 = open (argv[1],
O_WRONLY|O_CREAT|O_SYNC
        ,0600);
    fd2 = open (argv[1], O_RDWR);

    if ( (fd1== -1) || (fd2 == -1)) {
        printf ("open %s" ,argv[1]);
        return EXIT_FAILURE;
    }
```

```
        if (write (fd1,"abcdef", strlen ("abcdef"))
            == -1) { /*error*/}
        if (write (fd2,"123", strlen ("123")) )
            == -1) { /*error*/ }
        /* déplacement au début du fichier */
        if (lseek(fd2,0,SEEK_SET)
            == -1) { /*error*/}
        if ((n= read (fd2,tampon, SIZE_TAMPON))
            <=0) { /*error*/}
        for (i=0 ; i<n; i++)
            printf ("%c",tampon [i]);
        return EXIT_SUCCESS;
    }
```

```
>test-lseek fich3
123def
>cat fich3
123def
```

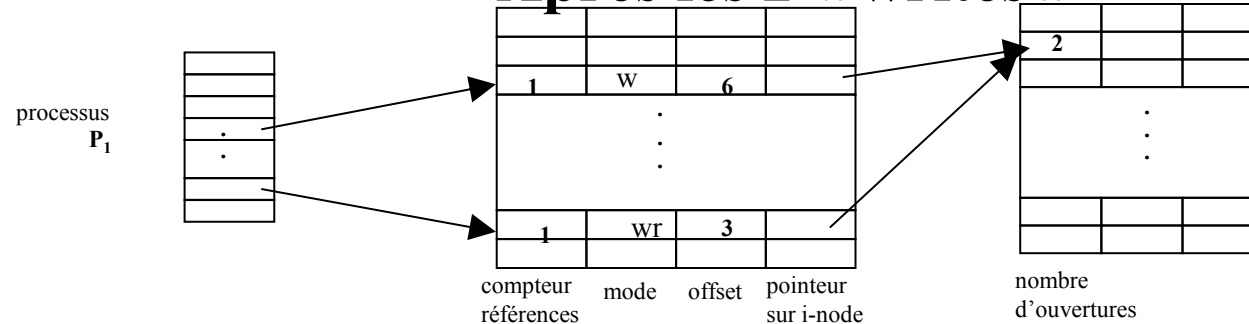
Organisation des Tables

Table de descripteurs

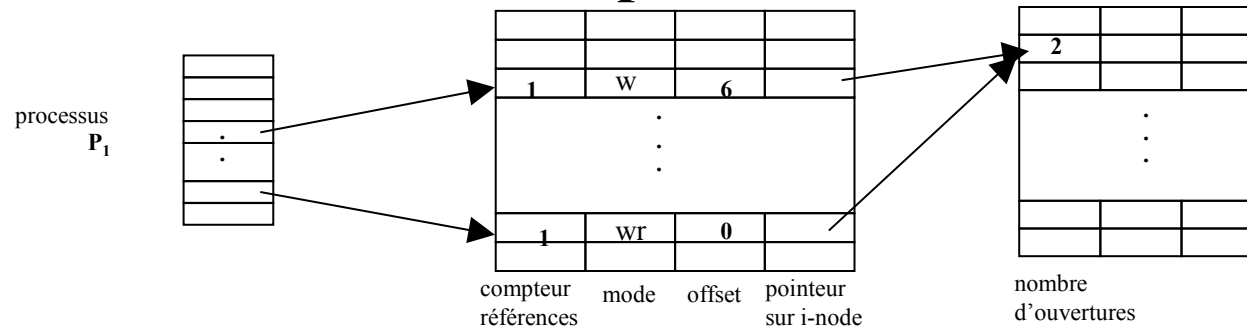
Table de fichiers ouverts

Table d'i-nodes

Après les 2 « writes »

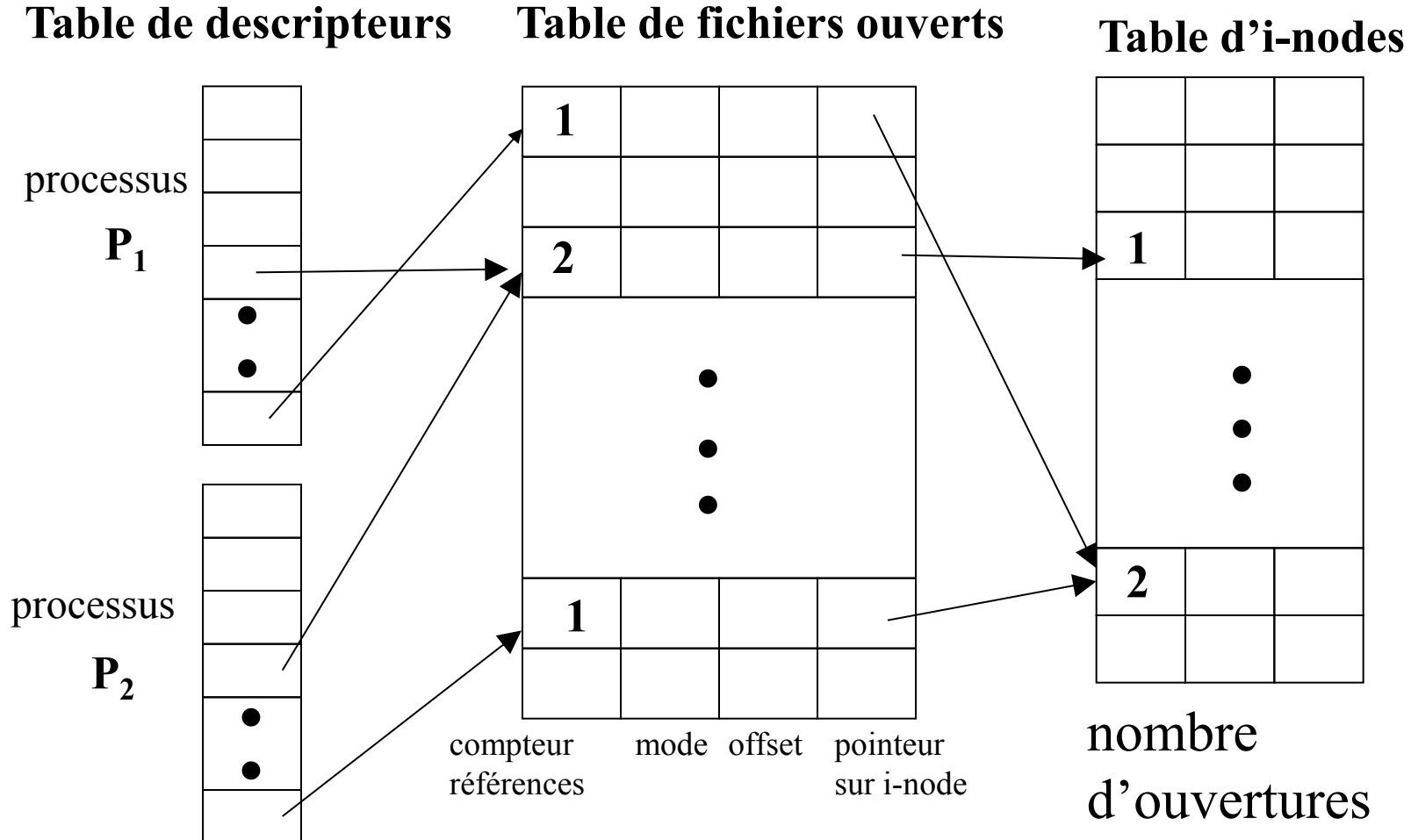


Après le lseek



Fork - organisation des Tables

En particulier, on partage les stdin/out/err et les offsets dedans



Exemple – fork

```
#define _POSIX_SOURCE 1
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <sys/wait.h>

#define SIZE_TAMPON 100
char tampon [SIZE_TAMPON];

int main (int argc, char* argv []) {
    int fd1, fd2;  int n,i;
    if ((
fd1 = open (argv[1],
            O_RDWR| O_CREAT|O_SYNC
            ,0600)) == -1) {}
    if (write (fd1,"abcdef", strlen ("abcdef"))
== -1) {}
```

```
    if (fork () == 0) {
        /* fils */
        if ((fd2 = open (argv[1], O_RDWR)) == -1) {}
        if (write (fd1,"123", strlen ("123")) == -1) {}
        if ((n= read (fd2,tampon, SIZE_TAMPON)) <=0)
            for (i=0 ; i<n; i++)
                printf ("%c",tampon [i]);
        exit (0);
    }
    else /* père */
        wait (NULL);
    return EXIT_SUCCESS;
}
```

Blocs vides {} = perror + exit

```
>test-fork fich4
  abcdef123
>cat fich4
  abcdef123
```

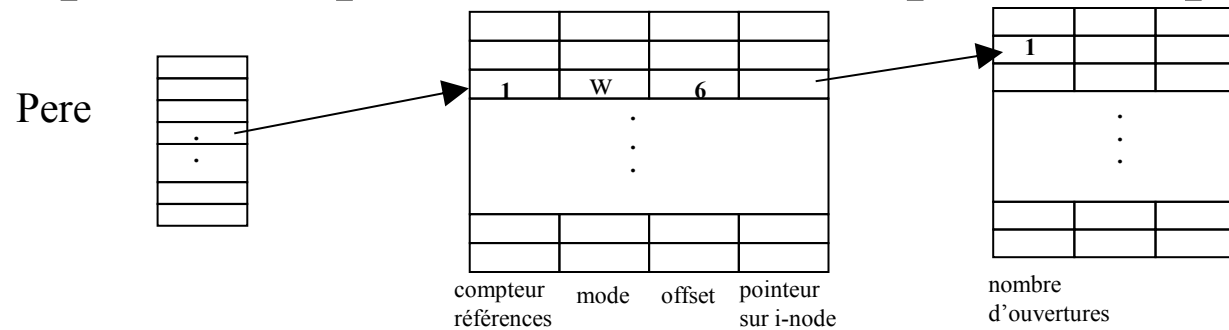
Organisation des Tables

Table de descripteurs

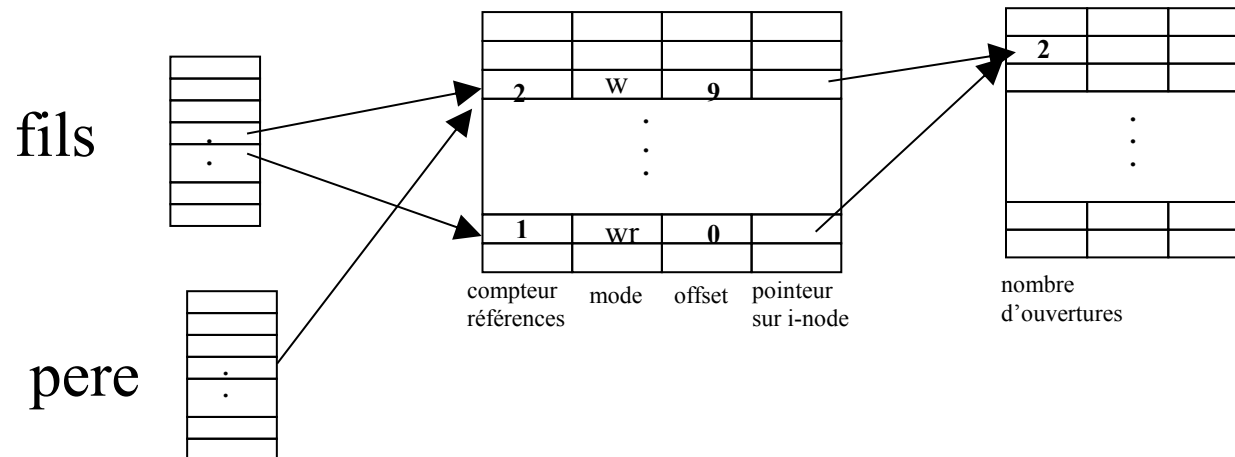
Table de fichiers ouverts

Table d'i-nodes

Après le « open » et « write » du processus principal



Après le fork , « open » et « write » (fils)



Duplication de descripteur

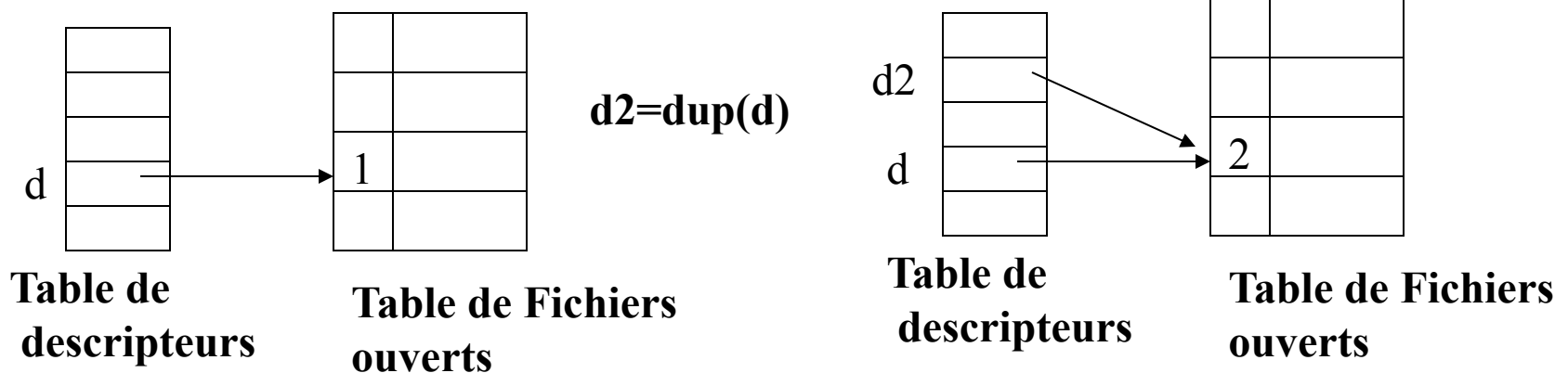
La primitive dup

✓ **int dup (int desc);**

- Recherche le + petit descripteur disponible dans la table des descripteurs du processus et en fait un synonyme de *desc*.

✓ **int dup2 (int desc, int desc2);**

- Force le descripteur *desc2* à devenir synonyme de *desc*.
- Utile en combinaison avec pipe par exemple



Exemple – dup2

```
#define _POSIX_SOURCE 1
```

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include <unistd.h>
```

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

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

```
int fd1;
```

```
int main (int argc, char* argv []) {
```

```
    if ((fd1 = open (argv[1], O_WRONLY| O_CREAT,  
                    0600)) == -1) {}
```

```
    printf ("avant le dup2: descripteur %d \n", fd1);
```

```
    dup2 (fd1, STDOUT_FILENO);
```

```
    printf ("après le dup2 \n");
```

```
    return EXIT_SUCCESS;
```

```
}
```

Redirection de stdout

> **test-dup2 fich5**

avant le dup2 : descripteur 3

> **cat fich5**

après le dup2

Manipulation de liens physiques

Création d'un lien physique sur un répertoire

✓ **int link (const char *origine, const char *cible)**

- permet de créer un nouveau lien physique
- contraintes
 - *origine* ne peut pas être un répertoire
 - *cible* ne doit pas exister

> ln Fic1 Fic2

> ls -la

Suppression d'un lien physique

✓ **int unlink (const char *ref)**

- supprime le lien associé à *ref*
- fichier supprimé si:
 - nombre de liens physiques sur le fichier est nul
 - nombre d'ouvertures du fichier est nul

24	.
43	..
78	Fic1
78	Fic2

Changement de nom de lien physique

✓ **int rename (const char *ancien, const char *nouveau)**

- *nouveau* ne doit pas exister
- impossible de renommer . et ..

code renvoi : 0 (succès) ; -1 (erreur)
--

Quelques erreurs associées aux E/S

Attention il FAUT tester et rattraper les valeurs -1/négatives

```
#include <errno.h>
```

```
extern int errno;
```

- ✓ **EACCESS** : accès interdit.
- ✓ **EBADF**: descripteur de fichier non valide.
- ✓ **EEXIST** : fichier déjà existant.
- ✓ **EIO**: erreur E/S.
- ✓ **EISDIR**: opération impossible sur un répertoire.
- ✓ **EMFILE**: trop de fichiers ouverts pour le processus (> OPEN_MAX).
- ✓ **EMLINK** : trop de liens physiques sur un fichier (> LINK_MAX).
- ✓ **ENAMETOOLONG** : nom fichier trop long (>PATH_MAX)
- ✓ **ENOENT** : fichier ou répertoire inexistant.
- ✓ **EPERM** : droits d'accès incompatible avec l'opération.

Tubes, Tubes Nommés

Tubes anonymes et nommés

Mécanisme de communications du système de fichiers

- ✓ I-node associé.
- ✓ Type de fichier: S_IFIFO.
- ✓ Accès au travers des primitives *read* et *write*.

Mécanisme de communication inter process

Sémantique bloquante / capacité

Gestion en mémoire efficace

Les tubes sont unidirectionnels

- ✓ Une extrémité est accessible en *lecture* et l'autre l'est en *écriture*.
- ✓ Dans le cas des tubes anonymes, si l'une ou l'autre extrémité devient inaccessible, cela est irréversible.

Tubes anonymes et nommés

Mode FIFO

- ✓ Première information écrite sera la première à être consommée en lecture.

Communication d'un flot continu de caractères (stream)

- ✓ Possibilité de réaliser des opérations de lecture dans un tube sans relation avec les opérations d'écriture.

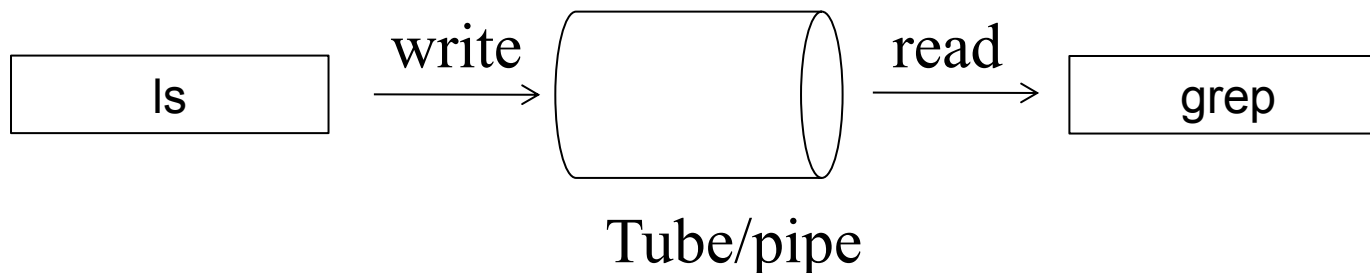
Opération de lecture est destructive :

- ✓ Une information lue est extraite du tube.

Les tubes (pipe)

Exemple : `ls | grep « toto »`

- La sortie STDOUT de « `ls` » est connectée à l'entrée STDIN de « `grep` »
 - `ls` peut tourner jusqu'à remplir le tube, puis il est bloqué
 - `grep` consomme les données dans le tube, se bloque si vide
 - Quand `ls` se termine, et que le tube est vide, `grep` se termine
- Gestion efficace :
 - au max taille du tube en mémoire, même si « `ls` » produit beaucoup de données.



Tubes anonymes et nommés

Capacité limitée

- ✓ Notion de tube plein (taille : PIPE_BUF).
- ✓ Écriture éventuellement bloquante.

Possibilité de plusieurs lecteurs et écrivains

- ✓ Nombre de lecteurs :
 - L'absence de lecteur interdit toute écriture sur le tube.
 - Signal SIGPIPE.
- ✓ Nombre d'écrivains :
 - L'absence d'écrivain détermine le comportement du *read*: lorsque le tube est vide, la notion de fin de fichier est considérée.

Primitive de **synchronisation et de communication** simple₄₄

Les tubes anonymes

Primitive pipe

Pas de nom

- ✓ Impossible pour un processus d'ouvrir un pipe anonyme en utilisant *open*.

Acquisition d'un tube:

- ✓ Création : primitive *pipe*.
- ✓ Héritage : *fork*, *dup*
 - Communication entre père et fils.
 - Un processus qui a perdu un accès à un tube n'a plus aucun moyen d'acquérir de nouveau un tel accès.

Les tubes anonymes

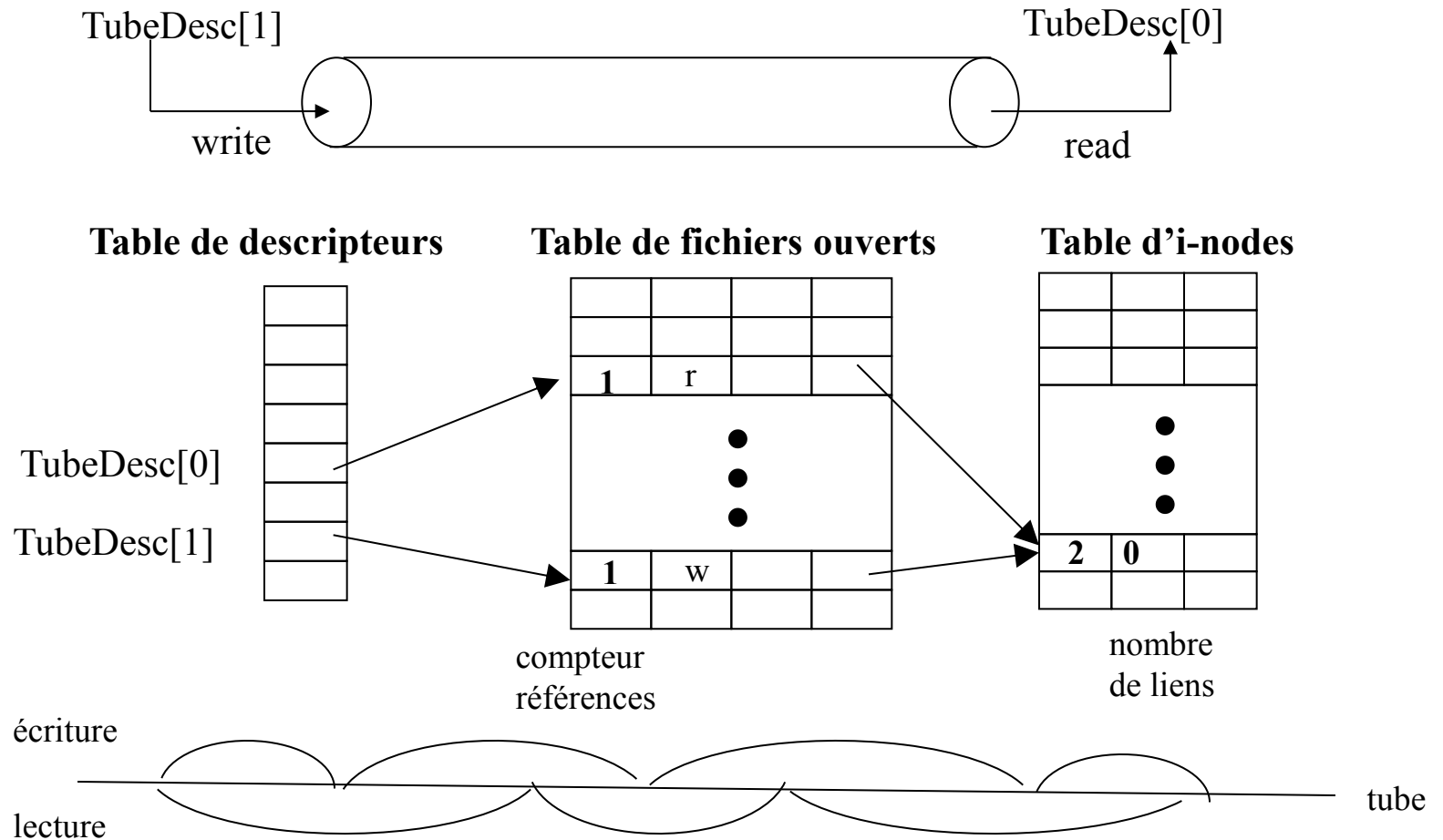
```
#include <unistd.h>
```

```
// déclarer int tubeDesc[2];
```

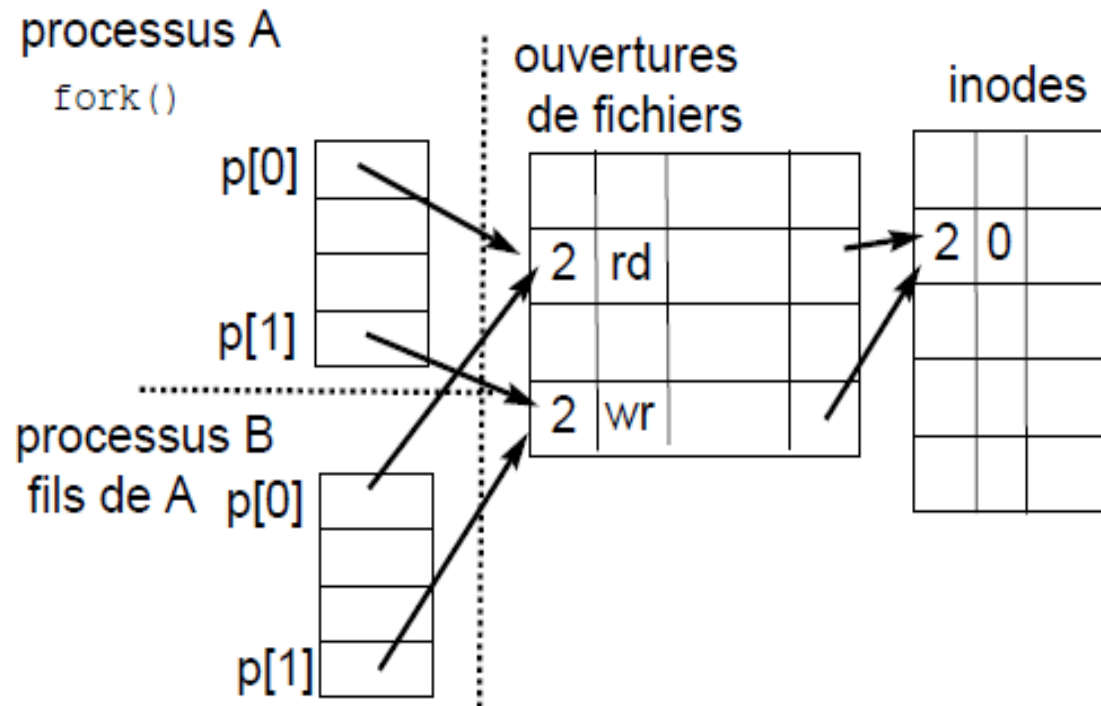
```
int pipe (int *TubeDesc);
```

- ✓ en cas de succès: appel renvoie 0
 - TubeDesc[0] : descripteur de lecture
 - TubeDesc[1] : descripteur d'écriture
- ✓ en cas d'échec : appel renvoie -1
 - errno = EMFILE (table de descripteurs de processus pleine).
 - errno = ENFILE (table de fichiers ouverts du système pleine).

Les tubes anonymes



Les tubes anonymes



Les tubes anonymes

Opérations autorisées

- ✓ *read, write* : lecture et écriture
 - Opérations bloquantes par défaut
- ✓ *close*: fermer des descripteurs qui ne sont pas utilisés
- ✓ *dup, dup2* : duplication de description; redirection
- ✓ *fstat, fcntl*: accès/modification des caractéristiques

Opérations non autorisées

- ✓ *open, stat, access, link, chmod, chown, rename*

Les tubes anonymes (fstat)

Accès aux caractéristiques d'un tube

```
struct stat stat;

int main (int argc, char ** argv) {
    int tubeDesc[2];

    if (pipe (tubeDesc) == -1) {
        perror ("pipe"); exit (1);
    }

    if ( fstat (tubeDesc[0],&stat) == -1) {
        perror ("fstat"); exit (2);
    }

    if (S_ISFIFO (stat.st_mode)) {
        printf ("il s'agit d'un tube \n");
        printf ("num. inode %d \n", (int)stat.st_ino);
        printf ("nbr. de liens %d \n", (int)stat.st_nlink);
        printf ("Taille : %d \n", (int) stat.st_size);
    }

    return EXIT_SUCCESS;
}
```

Les tubes anonymes (lecture)

Lecture dans un tube d'au plus `TAILLE_BUF` caractères

✓ `read (tube[0], buff, TAILLE_BUF);`

- **si le tube n'est pas vide** et contient *taille* caractères, lire dans *buff* min (*taille*, `TAILLE_BUF`). Ces caractères sont extraits du tube.
- **si le tube est vide**
 - **si le nombre d'écrivains est nul**
 - » fin de fichier; *read* renvoie 0
 - **sinon**
 - » **si la lecture est bloquante** (par défaut),
le processus est mis en sommeil jusqu'à ce que le tube ne soit plus vide ou qu'il n'y ait plus d'écrivains;
 - » **sinon**
retour immédiat; renvoie -1 et `errno = EAGAIN`.

Les tubes anonymes (écriture)

Ecriture de `TAILLE_BUF` caractères dans un tube:

- ✓ `write (tube[1], buff, TAILLE_BUF);`
 - Ecriture sera **atomique** si `TAILLE_BUF < PIPE_BUF`.

- **si le nombre de lecteurs dans le tube est nul**
 - signal **SIGPIPE** est délivré à l'écrivain (terminaison du processus par défaut); si **SIGPIPE** capté, fonction *write* renvoie -1 et **errno = EPIPE**.
- **sinon**
 - **si l'écriture est bloquante**
 - » le retour du *write* n'a lieu que lorsque `TAILLE_BUF` caractères ont été écrits.
 - **sinon**
 - » **si (`TAILLE_BUF <= PIPE_BUF`)**
s'il y a au moins `TAILLE_BUF` emplacements libres dans le tube
écriture atomique est réalisée;
 - sinon**
renvoie -1, **errno = EAGAIN**.
 - » **sinon**
le retour est un nombre inférieur à `TAILLE_BUF`.

Attention à boucler et à l'atomicité si grosse écriture

Les tubes anonymes (exemple fork)

Communication entre processus père et fils

```
#define _POSIX_SOURCE 1
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/wait.h>
#define S_BUF 100

int main (int argc, char ** argv) {
    int tubeDesc[2];
    char buffer[S_BUF];
    int n; pid_t pid_fils;

    if (pipe (tubeDesc) == -1) {
        perror ("pipe"); exit (1);
    }
    if ( (pid_fils = fork ( )) == -1 ) {
        perror ("fork"); exit (2);
    }
}
```

```
if (pid_fils == 0) { /*fils */
    if (( n = read (tubeDesc[0],buffer, S_BUF)) == -1) {
        perror ("read"); exit (3);
    }
    else {
        buffer[n] = '\0'; printf ("%s\n", buffer);
    }
    exit (0);
}
else { /*père */
    if ( write (tubeDesc[1],"Bonjour", 7)== -1) {
        perror ("write"); exit (4);
    }
    wait (NULL);
}
return (EXIT_SUCCESS); }
```

Affichage fils:
Bonjour

Les tubes anonymes (exemple 2 fork)

Communication entre père et fils : blocage

```
#define _POSIX_SOURCE 1
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/wait.h>
#define S_BUF 100

int main (int argc, char ** argv) {
    int tubeDesc[2];
    char buffer[S_BUF];
    int n;
    pid_t pid_fils;
    if (pipe (tubeDesc) == -1)
    { perror ("pipe"); exit (1); }
    if ( (pid_fils = fork ()) == -1 ) {
        perror ("fork"); exit (2); }
    if (pid_fils == 0) { /* fils */
        for (int i=0; i<2; i++) {
            if (( n= read (tubeDesc[0], buffer, S_BUF)) == -1)
                { perror ("read"); exit (3); }
            else
                { buffer[n] = '\0'; printf ("%s\n",buffer); }
        }
        exit (0);
    } else { /* père */
        for (int i=0; i<2; i++) {
            if ( write (tubeDesc[1], "Bonjour", 7) == -1 )
                { perror ("write"); exit (4); }
        }
        wait (NULL);
    }
    return (EXIT_SUCCESS);
}
```

Affichage fils:

BonjourBonjour

- Processus père et fils bloqués si double write avant read

Les tubes anonymes (exemple)

Ecriture dans un tube sans lecteur

```
#define _POSIX_SOURCE 1
#include <sys/types.h>
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

void sig_handler (int sig) {
    if (sig == SIGPIPE)
        printf ("ecriture dans un tube sans lecteurs \n");
}

int main (int argc, char ** argv) {
    int tubeDesc[2]; struct sigaction action;
    action.sa_handler= sig_handler;
    sigaction (SIGPIPE, &action, NULL);
```

test-sigpipe.c

```
if (pipe (tubeDesc) == - 1) {
    perror ("pipe");
    exit (1);
}
close (tubeDesc[0]); /* sans lecteur */

if ( write (tubeDesc[1], "x", 1) == -1)
    perror ("write");

return EXIT_SUCCESS;
}
```

> test-sigpipe

ecriture dans un pipe sans lecteurs
write: Broken pipe

Les tubes anonymes

read et *write* sont des opérations bloquantes par défaut

fonction *fcntl*:

- ✓ permet de les rendre non bloquantes

```
int tube[2], attributs;
```

```
..... pipe (tube); .....
```

```
/* rendre l'écriture non bloquante */
```

```
attributs = fcntl (tube[1], F_GETFL);
```

```
attributs |= O_NONBLOCK;
```

```
fcntl (tube[1], F_SETFL,attributs);
```

```
.....
```


Les tubes anonymes (dup et close)

close

- ✓ Fermeture des descripteurs qui ne sont pas utilisés.

dup, dup2

- ✓ Duplication des descripteurs.
- ✓ Rediriger les entrées-sorties standard d'un processus sur un tube.

```
int tube[2], attributs;
```

```
.....
```

```
    pipe (tube); ....
```

```
    dup2(tube[0], STDIN_FILENO);
```

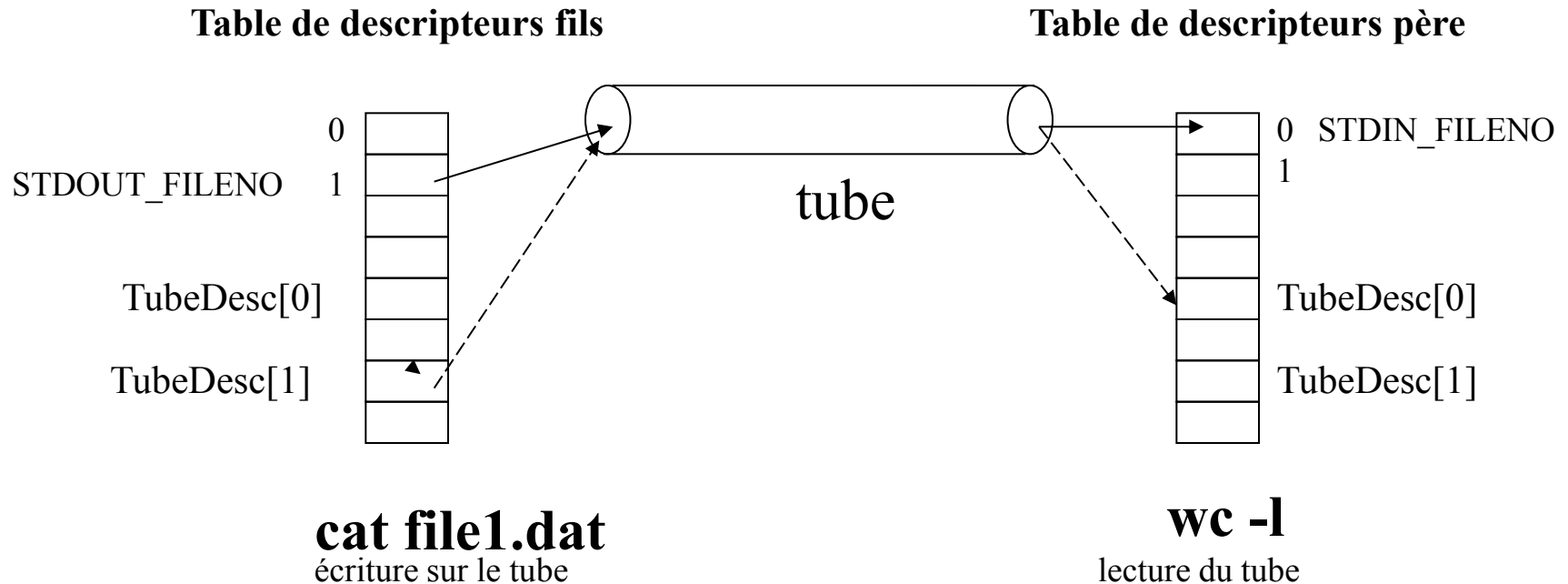
```
    close (tube[0]);
```

```
...
```

Les tubes anonymes (exemple de redirection)

/* nombre de lignes d'un fichier */

✓ `cat file1.dat | wc -l`



Les tubes anonymes (exemple de redirection)

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

int main (int argc, char ** argv) {
    int tubeDesc[2]; pid_t pid_fils;

    if (pipe (tubeDesc) == -1) {
        perror ("pipe");
        exit (1);
    }

    if ( (pid_fils = fork ( )) == -1 ) {
        perror ("fork");
        exit (2);
    }
```

```
    if (pid_fils == 0) { /* fils */
        dup2(tubeDesc[1],STDOUT_FILENO);
        close (tubeDesc[1]); close (tubeDesc[0]);
        if (execl ("/bin/cat", "cat", "file1.dat",NULL) == -1) {
            perror ("execl"); exit (3);
        }
    }
    else { /* père */
        dup2(tubeDesc[0],STDIN_FILENO);
        close (tubeDesc[0]);
        close (tubeDesc[1]);
        if (execl ("/bin/wc", "wc", "-l", NULL) == -1) {
            perror ("execl"); exit (3);
        }
    }
    return (EXIT_SUCCESS);
}
```

Tubes Nommés

Tubes nommés

Permettent à des processus sans lien de parenté de communiquer en mode flot (stream).

- ✓ Toutes les caractéristiques des tubes anonymes.
- ✓ Sont référencés dans le système de gestion de fichiers.
- ✓ Utilisation de la fonction *open* pour obtenir un descripteur en lecture ou écriture.
- ✓ `ls -l`

```
prw-rw-r--  1 arantes  src          0 Nov  9 2004  tube1
```

Tubes nommés (mkfifo)

Création d'un tube nommé

✓ **mkfifo [-p] [-m mode] *référence***

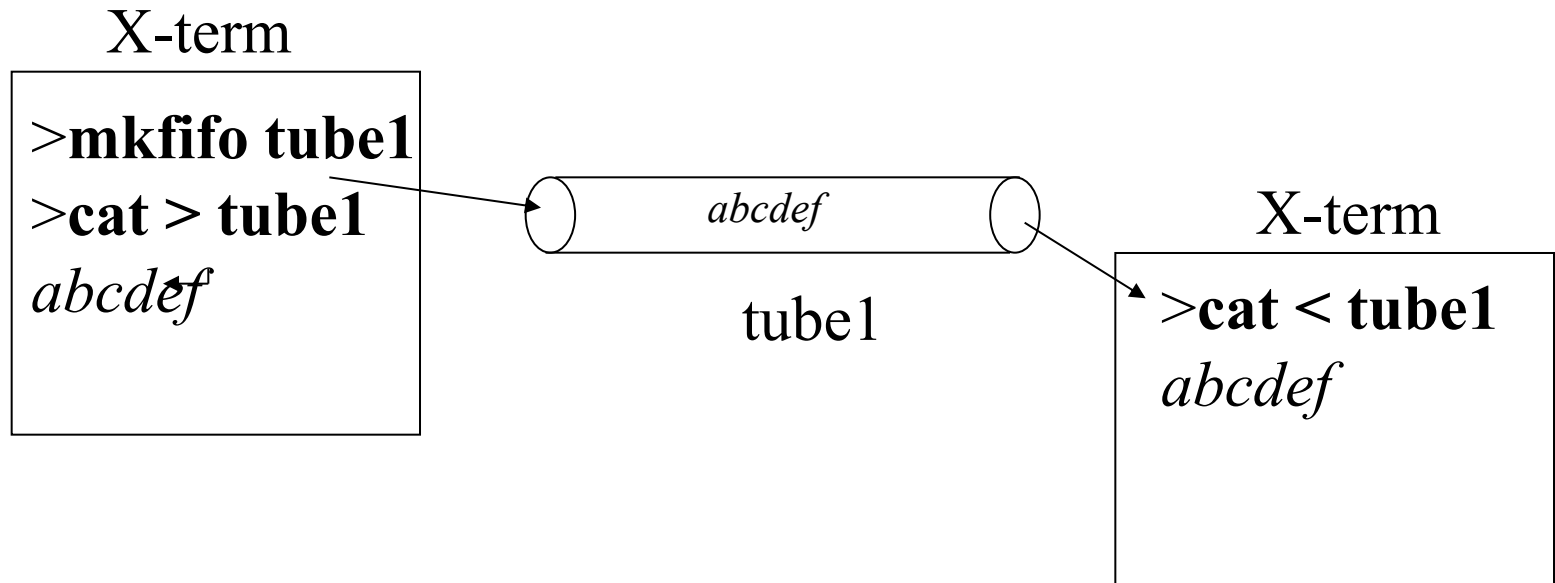
- -m mode : droits d'accès (les mêmes qu'avec chmod).
- -p : création automatique de tous les répertoires intermédiaires dans le chemin *référence*.

Fonction

✓ **int mkfifo (const char *ref, mode_t droits);**

- *ref* définit le chemin d'accès au tube nommé et *droits* spécifie les droits d'accès.
- Renvoie 0 en cas de succès; -1 en cas d'erreur.
 - errno = EEXIST, si fichier déjà créé.

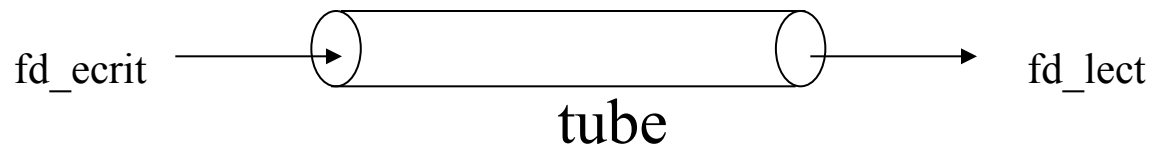
Tubes nommés (mkfifo)



Tubes nommés (open)

Par défaut bloquante (rendez-vous):

- ✓ Une demande d'ouverture en lecture est bloquante s'il n'y a aucun écrivain sur le tube.
- ✓ Une demande d'ouverture en écriture est bloquante s'il n'y a aucun lecteur sur le tube.
- ✓ Attention à initialiser de façon asymétrique si bi-directionnel : P0: read puis write, P1 write puis read



```
int fd_lect, fd_ecrit;  
fd_lect = open ("tube", O_RDONLY);  
fd_ecrit = open ("tube", O_WRONLY);
```


Tubes nommés (*open*)

Ouverture non bloquante

- ✓ Option `O_NONBLOCK` lors de l'appel à la fonction *open*
 - **Ouverture en lecture :**
 - Réussit même s'il n'y a aucun écrivain dans le tube.
 - Opérations de lectures qui se suivent sont non bloquantes.
 - **Ouverture en écriture :**
 - Sur un tube sans lecteur, l'ouverture échoue: valeur -1 renvoyée.
 - Si le tube possède des lecteurs, l'ouverture réussit et les écritures dans les tubes sont non bloquantes.

Tube nommé (suppression du nœud)

Un nœud est supprimé quand:

- ✓ Le nombre de liens physiques est nul.
 - Fonction *unlink* ou commande *rm*.
- ✓ Le nombre de liens internes est nul.
 - Nombres de lecteurs et écrivains sont nuls.

Si nombre de liens physiques est nul, mais le nombre de lecteurs et/ou écrivains est non nul

- ✓ Tube nommé devient un tube anonyme.

Tubes nommés (écrivain)

```
#define _POSIX_SOURCE 1
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/stat.h>
#include <fcntl.h>

#define S_BUF 100
int n ;
char buffer[S_BUF];
int main (int argc, char ** argv) {
    int fd_write;

    if ( mkfifo(argv[1],
        S_IRUSR|S_IWUSR) == -1) {
        perror ("mkfifo");
        exit (1);
    }
```

```
    if (( fd_write = open (argv[1],
        O_WRONLY)) == -1) {
        perror ("open");
        exit (2);
    }

    if (( n= write(fd_write,"Bonjour", 7)) == -1) {
        perror ("write");
        exit (3);
    }

    close (fd_write);
    return EXIT_SUCCESS;
}
```

Tubes nommés (lecteur)

```
#define _POSIX_SOURCE 1
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/stat.h>
#include <fcntl.h>
#define S_BUF 100
int n ; char buffer[S_BUF];

int main (int argc, char ** argv) {
    int fd_read;
    if (( fd_read = open (argv[1],
        O_RDONLY)) == -1) {
        perror ("open"); exit (1)
    }
}
```

```
if (( n= read (fd_read, buffer, S_BUF)) == -1){
    perror ("read");
    exit (2);
}
else {
    buffer[n] = '\0';
    printf ("%s\n",buffer);
}
close (fd_read);
return EXIT_SUCCESS;
}
```

Tubes nommés: interblocage (ouverture bloquante)

PROCESSUS 1 :

```
int main (int argc, char ** argv) {
    int fd_write, fd_read;

    if ( (mkfifo("tube1",S_IRUSR|S_IWUSR) == -1) ||
        (mkfifo("tube2",S_IRUSR|S_IWUSR) == -1)) {
        perror ("mkfifo"); exit (1);
    }

    if (( fd_write = open ("tube1", O_WRONLY)) == -1) {
        perror ("open"); exit (2);
    }

    if (( fd_read = open ("tube2", O_RDONLY)) == -1) {
        perror ("open"); exit (3);
    }
    .....
    return EXIT_SUCCESS;
}
```

PROCESSUS 2 :

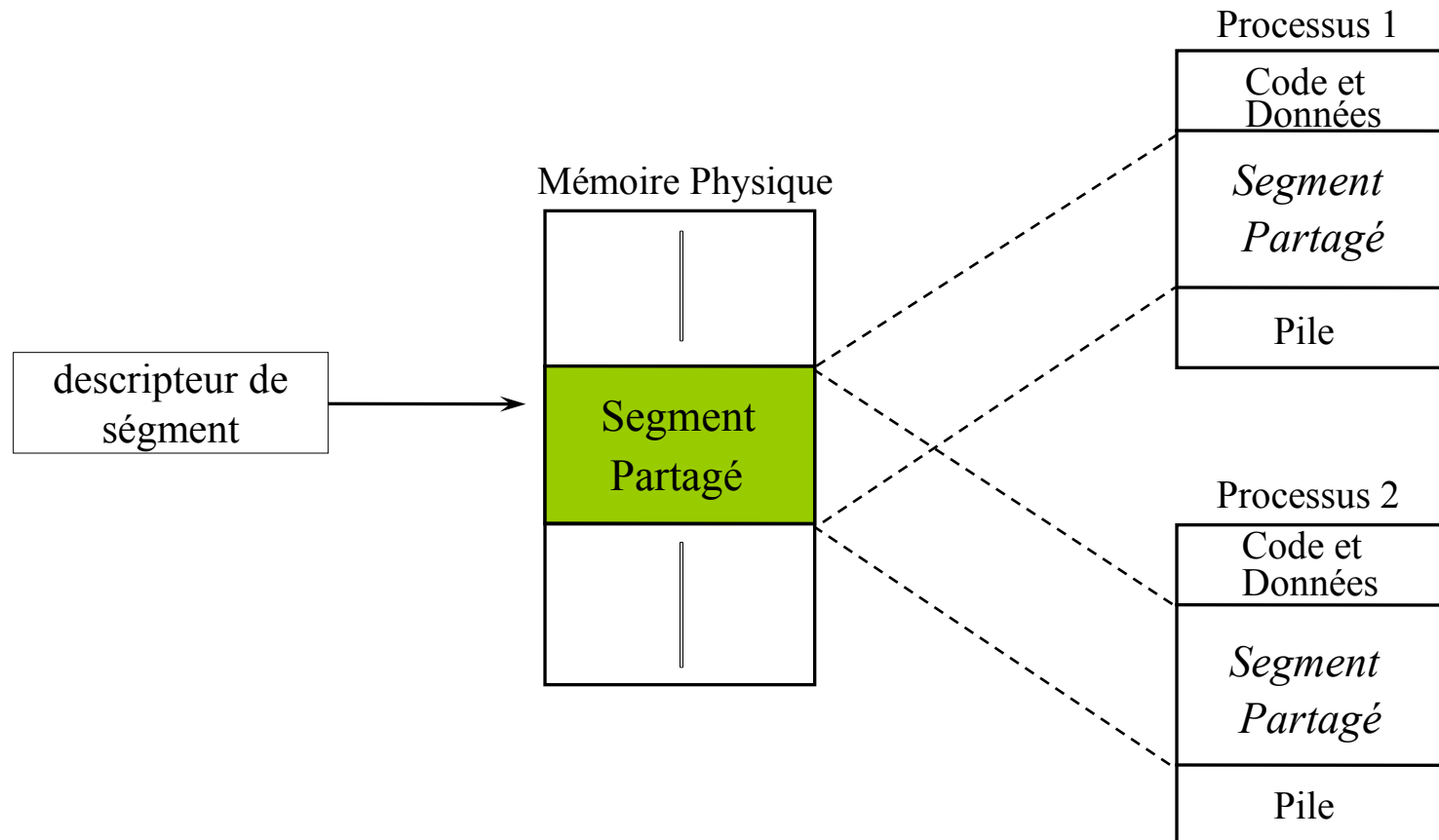
```
int main (int argc, char ** argv) {
    int fd_write, fd_read;

    if (( fd_write = open ("tube2", O_WRONLY)
        == -1) {
        perror ("open"); exit (2);
    }

    if (( fd_read = open ("tube1", O_RDONLY))
        == -1) {
        perror ("open"); exit (3);
    }
    .....
    return EXIT_SUCCESS;
}
```

Mémoire partagée

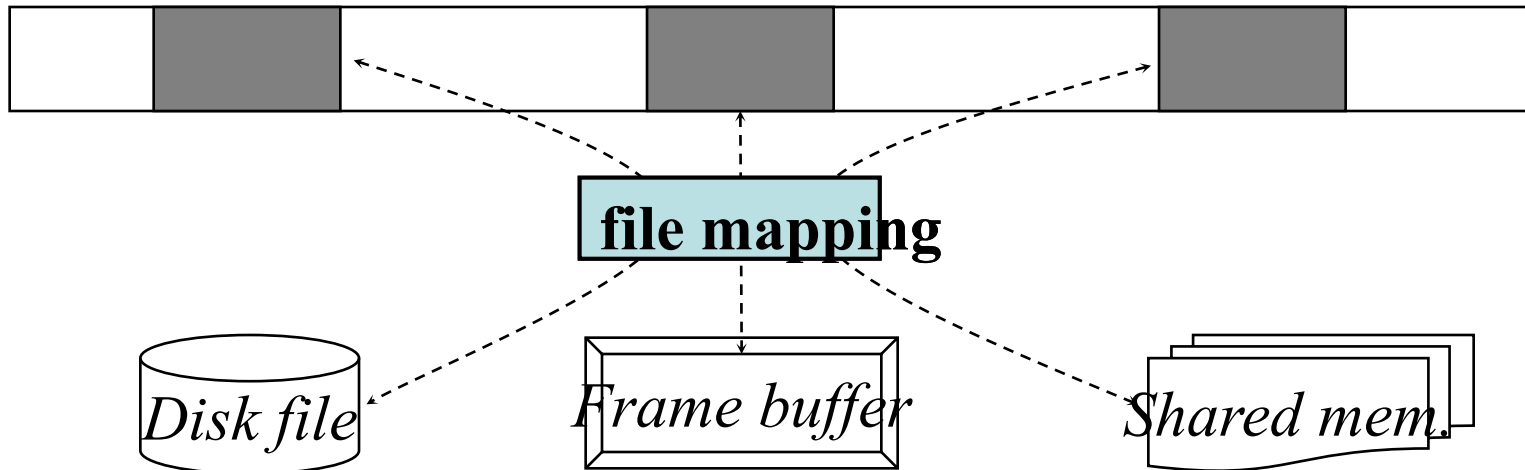
Segments de mémoire partagée



Segments de mémoire partagée

Principe

- ✓ Zone mémoire attachée à un processus mais accessible pour d'autres processus
- ✓ Liée à un autre service : file mapping
 - Etablissement d'une correspondance (attachement) entre :
 - Un fichier (ou un segment de mémoire)
 - Une partie de l'espace d'adressage d'un processus réservée à cet effet



Segments de mémoire partagée

Avantages

- ✓ Accès totalement libre
 - Chaque processus détermine à quelle partie de la structure de données il accède
- ✓ Efficacité
 - Pas de recopie mémoire : tous les processus accèdent **directement** au même segment

Désavantages

- ✓ Accès totalement libre
 - Pas de synchro implicite comme pour les tubes et les files de msgs
 - \Rightarrow Synchro doit être explicitée (sémaphores ou signaux)
- ✓ Pas de gestion de l'adressage
 - Validité d'un pointeur limitée à son esp. d'adressage
 - \Rightarrow Impossible de partager des pointeurs entre processus

Mémoire partagée POSIX

Fichier <sys/mman.h>

Fonctions contenues dans la bibliothèque librt (real-time)

```
$ gcc -Wall -o monprog monprog.c -lrt
```

Accès

- ✓ `shm_open` \Rightarrow créer / ouvrir un segment en mémoire
- ✓ `close` \Rightarrow fermer un segment
- ✓ `mmap` \Rightarrow attacher un segment dans l'espace du processus
- ✓ `munmap` \Rightarrow détacher un segment de l'espace du processus
- ✓ `shm_unlink` \Rightarrow détruire un segment

Opérations sur un segment

- ✓ `mprotect` \Rightarrow changer le mode de protection d'un segment
- ✓ `ftruncate` \Rightarrow allouer une taille à un segment

Ouverture / Destruction d'un segment de mémoire partagée

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

```
int shm_open(const char *name, int flags,  
             mode_t mode);
```

- ✓ Crée un nouveau segment de taille 0
ou recherche le descr. d'un segment déjà existant
- ✓ Retourne un descripteur positif en cas de succès, -1 sinon
- ✓ *flags* idem open
- ✓ *mode* idem chmod

```
int shm_unlink(const char *name);
```

- ✓ Idem mq_unlink

Attachement / Détachement d'un segment de mémoire partagée

```
void * mmap(void *addr, size_t len, int prot, int flags,  
            int fd, off_t offset);
```

- ✓ Retourne NULL en cas d'échec,
l'@ d'un attachement de taille *len* à partir d'*offset* ds le segment de descr *fd* sinon
 - ✓ *Addr* : adresse où attacher le segment en mémoire ; 0 \Rightarrow choix du système
 - ✓ *prot*: protection associée (PROT_READ, PROT_WRITE, PROT_EXEC, PROT_NONE)
 - ✓ *flags*: mode de partage
 - MAP_SHARED : modifs visibles par tous les pcs ayant accès (partage)
 - MAP_PRIVATE : modifs visibles par le pcs appelant uniquement (shadow copy)
 - MAP_FIXED : force l'utilisation d'*addr*
- `mmap(0, sizeof(data), PROT_READ|PROT_WRITE, MAP_ANONYMOUS | MAP_SHARED, -1, 0)`
: sous linux, MAP_ANONYMOUS + passer -1 comme fd = proche malloc

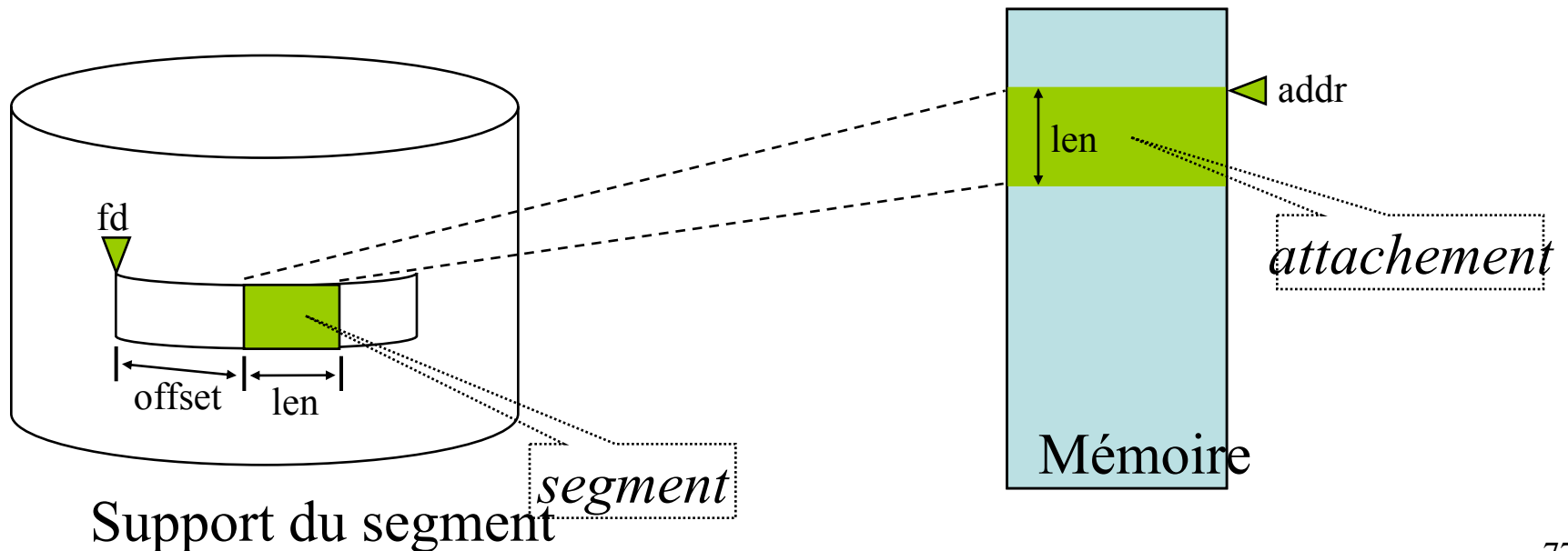
```
int munmap(caddr_t addr, size_t len);
```

- ✓ Détruit l'attachement de taille *len* à l'adresse *addr*
- ✓ Retourne -1 en cas d'échec, 0 sinon

Projection des fichiers

Permet de projeter dans l'espace d'adressage du processus un segment de longueur *len* le segment $[offset, offset+len]$ du fichier associé à *fd*.

```
#include<sys/mman.h>
#include<sys/types.h>
void * mmap(void *addr, size_t len, int prot, int flags,
            int fd, off_t offset);
```



Opérations sur un segment de mémoire partagée

```
void * mprotect(caddr_t addr, size_t len, int prot);
```

- ✓ Modifie la protection associée au segment : PROT_READ, PROT_WRITE, PROT_EXEC, PROT_NONE
- ✓ Retourne -1 en cas d'échec, 0 sinon

```
int ftruncate(int fd, off_t length);
```

- ✓ Définit la taille du segment de descr. *fd*
 - nouvelle taille = length
 - si (ancienne taille > nouvelle taille), alors les données en excédent sont perdues
- ✓ Retourne -1 en cas d'échec, 0 sinon

Exemple mémoire partagée

```
int *sp;
int main() {
    int fd;
    /* Créer le segment monshm, ouverture en R/W */
    if ((fd = shm_open("monshm", O_RDWR | O_CREAT,
        0600)) == -1) {
        perror("shm_open");
        exit(1);}

    /* Allouer au segment une taille pour stocker un entier */
    if (ftruncate(fd, sizeof(int)) == -1) {
        perror("ftruncate");
        exit(1);}

    /* “mapper” le segment en R/W partagé */
    if ((sp = (int *) mmap(NULL, sizeof(int), PROT_READ |
        PROT_WRITE, MAP_SHARED, fd, 0))
        == MAP_FAILED) {
        perror("mmap");
        exit(1);}

}
```

```
/* Accès au segment */
*sp = 10;

/* “detacher” le segment */
munmap(sp, sizeof(int));

/* détruire le segment */
shm_unlink("monshm");
return 0;
```

Et aussi en version anonyme

```
void * ptr = mmap(0, // addr au choix système
    sizeof(data), // taille allocation
    PROT_READ|PROT_WRITE, // permissions
    MAP_ANONYMOUS | MAP_SHARED, // Anonyme ! Linux
    -1, // filedescriptor : non positionné
    0) // offset
```

: (*non POSIX*) sous linux,

MAP_ANONYMOUS

+ passer -1 comme fd = proche malloc

Sémaphores

Les sémaphores

Principe (Dijkstra)

- ✓ Mécanisme de synchronisation
 - accès concurrents à une ressource partagée (*eg. segment de mémoire*)
 - solution au problème de l'exclusion mutuelle

Structure sémaphore

- ✓ un compteur : nb d'accès disponibles avant blocage
- ✓ une file d'attente : processus bloqués en attente d'un accès

Opérations :

- Acquisition (noté P) : Décrémenter le compteur de une unité (bloque si non disponible)
- Libération (noté V) : Incrémenter le compteur de une unité (débloque le cas échéant)

Les sémaphores

Fonctionnement

- ✓ Demande d'accès (P - *proberen* ou "*puis-je ?*")
 - sem_get
 - Décrémentation du compteur
 - Si compteur < 0 , alors blocage du pcs et insertion ds la file

- ✓ Fin d'accès (V - *verhogen* ou "*vas-y*")
 - sem_put
 - Incrémentation du compteur
 - Si compteur ≥ 0 , alors déblocage d'un pcs de la file

Blocage, déblocage et insertion des processus dans la file sont des ops implicites

Les sémaphores

Exemple

init (sem, 1) ;

P1 {

....

P(sem)
région critique)
V(sem);

...

}

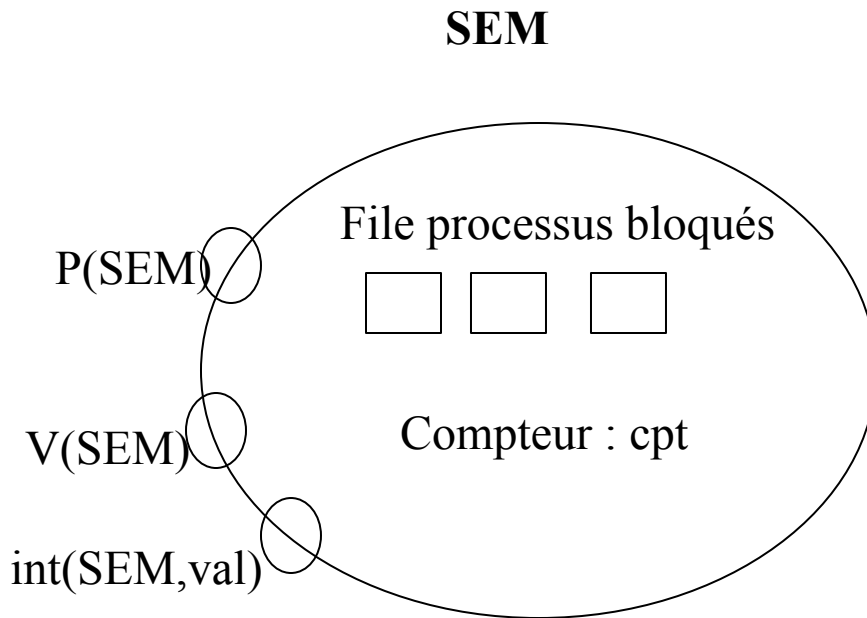
P2 {

....

P(sem)
région critique)
V(sem);

...

}



Les sémaphores

Dysfonctionnements possibles

- ✓ Liés à l'exécution non déterministe

Interblocage

- ✓ 2 processus P et Q sont bloqués en attente
 - P attend que Q signale sa fin d'accès et Q attend que P signale sa fin d'accès

Famine

Un processus est bloqué en attente d'une fin d'accès qui n'arrivera jamais

Sémaphore POSIX vs PTHREAD

- Le sémaphore joue plusieurs rôles simultanément
 - Il peut servir de mutex, si on l'initialise à 1, et que c'est le même processus qui P(1) et V(1)
 - Il peut servir de condition_variable, si on l'initialise à 0 et que un processus fait P(1) pour attendre, et un **autre** fait V(1) pour le notifier.
 - NB: c'est interdit avec les mutex.
 - (sémaphores System V, non POSIX) Il peut servir de verrou lecteur/écrivain, si on l'initialise à N le maximum de lecteurs, qu'un lecteur fait P(1)/V(1) et un écrivain fait P(N)/V(N) :

Sémaphores POSIX

Deux types de sémaphores :

- ✓ Sémaphores nommés

- Portée : tous les processus de la machine
- Primitives de base : **sem_open**, **sem_close**, **sem_unlink**, **sem_post**, **sem_wait**

- ✓ Sémaphores anonymes (*memory-based*)

- Portée : processus avec filiation, uniquement threads dans linux
- Primitives de base : **sem_init**, **sem_destroy**, **sem_post**, **sem_wait**

Opérations

- ✓ $P() = \text{sem_wait}$; $V = \text{sem_post}$

Inclus dans la bibliothèque des pthreads

```
$ gcc -Wall -o monprog monprog.c -lpthread
```

```
$ g++ -Wall -o monprog monprog.cpp -pthread
```

Création de sémaphore nommé

```
#include <semaphore.h>
```

```
sem_t *sem_open(const char *name, int oflag,  
                mode_t mode, int value);
```

- ✓ Crée ou ouvre le sémaphore de nom *name*
- ✓ *oflag, mode* idem open
- ✓ *value* valeur initiale du compteur

Retourne un pointeur sur le sémaphore, NULL en cas d'erreur

Ex : Creation d'un sémaphore initialisé à 10

```
sem_t *s;
```

```
s = sem_open (« monsem », O_CREAT | O_RDWR, 0600, 10);
```


Création de sémaphore anonyme

```
int sem_init(sem_t *sem, int pshared, unsigned val);
```

- ✓ Crée et initialise le sémaphore *sem*.
 - *sem* : doit être alloué dans l'espace d'adressage du processus
- ✓ *pshared* != 0
 - partageable entre processus (filiation)
 - Sem: référence à un objet partagé (mémoire partagée)
- ✓ *pshared* == 0
 - partageable entre threads
- ✓ *val*: valeur initiale du sémaphore

Retourne -1 en cas d'erreur, 0 sinon

Ex : création de sémaphore partagé, initialisé à 10

```
sem_t s;  
sem_init(&s, 1, 10);
```

Opérations sur sémaphore

- **Opération P**

int sem_wait (sem_t *sem);

- Attendre que le compteur soit supérieur à zéro et le décrémenter avant de revenir.

- **Opération V**

int sem_post (sem_t *sem);

- Compteur incrémenté; un processus/thread en attente est libérée.

- **Opération P non bloquant**

int sem_trywait (sem_t *sem);

- Fonctionnement égal à *sem_wait* mais non bloquante.

- **Consultation compteur sémaphore**

int sem_getvalue (sem_t *sem, int *valeur);

- Renvoie la valeur du compteur du sémaphore *sem*. dans *valeur.

Fermeture / Destruction

Sémaphore nommé :

- ✓ Fermer le sémaphore
- ✓ `int sem_close(sem_t *sem) ;`
- ✓ Détruire le sémaphore
- ✓ `int sem_unlink(const char *name) ;`

Sémaphore anonyme :

- ✓ `int sem_destroy(sem_t *sem) ;`

Exemple : sémaphores nommés

```
...
int main() {

    sem_t *smutex;

    /* creation d'un semaphore mutex initialisé à 1 */
    if ((smutex = sem_open("/monsem",
        O_CREAT | O_EXCL | O_RDWR, 0666, 1)) ==
        SEM_FAILED) {
        if (errno != EEXIST) {
            perror("sem_open"); exit(1);
        }

        -----

        /* Semaphore déjà créé, ouvrir sans O_CREAT */
        smutex = sem_open("/monsem", O_RDWR);
    }

    /* P sur smutex */
    sem_wait(smutex);

    region critique

    /* V sur smutex */
    sem_post(smutex);

    /* Fermer le semaphore */
    sem_close(smutex);

    /* Detruire le semaphore */
    sem_unlink("monsem");
    return 0;
}
```

Exemple : sémaphores anonymes

```
#include <pthread.h>
#include <semaphore.h>
sem_t mutex; //

void *my_thread(void *arg){

while(1){
sem_wait (&mutex); // P( )
//SC
sem_post(&mutex) // V ( )
}
}
```

```
int main(){
pthread_t thread1_id, thread2_id;

sem_init(&mutex, 0, 1); // initialize mutex

pthread_create(&thread1_id, NULL, &my_thread, NULL);
pthread_create(&thread2_id, NULL, &my_thread, NULL);

pthread_join(thread1_id, NULL);
pthread_join(thread2_id, NULL);

sem_destroy(&mutex);
}
```

sémaphores anonymes avec des processus

```
struct myshm {
    sem_t sem;
}

int shm_id;
struct myshm *shm;

/* Créé le segment en lecture écriture */
if ((shm_id = shm_open("/shm1",
O_RDWR |
    O_CREAT, 0666)) == -1) {
    perror("shm_open ");
    exit(EXIT_FAILURE);
}

/* Allouer au segment une taille*/
if(ftruncate(shm_id, sizeof(struct myshm))
== -1) {
    perror("ftruncate shm");
    exit(EXIT_FAILURE);
}
```

```
/* Mapper le segment en read-write partagée*/
if((shm= mmap(NULL, sizeof(struct myshm),
    PROT_READ | PROT_WRITE,
    MAP_SHARED,
    shm_id, 0)) == MAP_FAILED){
    perror("mmap shm_pere");
    exit(EXIT_FAILURE);
}

if (sem_init(&(shm->sem), 1, 1) == -1){
    perror("sem_init ");
    exit(EXIT_FAILURE);
}

....
sem_wait(&(shm->sem));

....

sem_post(&(shm->sem));
```