
# Système d'exploitation et programmation système

## Cours 2

EISTI

2009/2010

Département informatique

# Echange de données entre processus

Utilisation de tubes

# Pipe “commande shell”

■La commande "**ps −a | wc −l**" entraîne la création de deux processus concurrents (allocation du processeur). Un tube est créé dans lequel les résultats du premier processus ("**ps −a**") sont écrits. Le second processus lit dans le tube.

■Un tube de communication (|) permet de mémoriser des informations. Il se comporte comme une file FIFO, d’où son aspect unidirectionnel.

■Lorsque le processus écrivain se termine et que le processus lecteur dans le tube a fini d’y lire (le tube est donc vide et sans lecteur), ce processus détecte une fin de fichier sur son entrée standard et se termine.

# Synchronisation

- Le système assure la synchronisation de l'ensemble dans le sens où :
  - il bloque le processus lecteur du tube lorsque le tube est vide en attendant qu'il se remplisse (s'il y a encore des processus écrivains);
  - il bloque (éventuellement) le processus écrivain lorsque le tube est plein (si le lecteur est plus lent que l'écrivain et que le volume des résultats à écrire dans le tube est important).
- Le système assure l'implémentation des tubes. Il est chargé de leur création et de leur destruction.

# Communication

- Par communication inter-processus on entend :
  - Existence de plusieurs processus sur la même machine travaillant "simultanément"
  - Echange d'information entre ces processus

# Problèmes classiques

- Quels moyens de communications sont disponibles ?
- Comment choisir le moyen de communication approprié ?
- Comment être sûr que le bon processus accède à la donnée qu'il attend ?
- Comment hiérarchiser l'exécution des processus pour que l'information demandée soit disponible ?

# Un air de déjà vu...

- Echange d'information par fichier : un conteneur (fichier) stocke l'information la rendant disponible pour tout autre processus ayant accès à ce fichier.
- En script : l'utilisation de pipe permet d'enchaîner des commandes (donc processus différents) en leur transmettant des informations.

# Les tubes

- Un tube est presque identique à un fichier ordinaire. Il est caractérisé par :
- Taille limitée
- Deux extrémités, permettant chacune soit de lire dans le tube, soit d'y écrire
- Au plus deux entrées dans la table des fichiers ouverts (une pour la lecture et une pour l'écriture)
- L'opération de lecture dans un tube est destructrice : une information ne peut être lue qu'une seule fois dans un tube
- Il existe deux sortes de tubes :
  - Ordinaire ou non nommé
  - Nommé (FIFO)

# Tubes anonymes (non nommés)

- Fichier logique
- Deux descripteurs (lecture/écriture)
- Aucune référence dans le système de fichier (fichier anonyme)
- Norme SVr4 :
  - Unidirectionnel ( SUN OS bidirectionnels)
  - Un descripteur pour la lecture, un descripteur pour l'écriture
- Lien de parenté obligatoire

# Tubes anonymes : comportement par défaut

- Tube vide :
  - Lecture bloquante
- Tube non vide :
  - On lit uniquement les caractères disponibles, même si on en attend plus
- Tube plein :
  - Ecriture bloquante

# Tubes anonymes : comportement sans lecteur ou écrivain

- Ecriture sans lecteur :
  - SIGPIPE envoyé par le système, comportement par défaut : fin du processus
- Lecture sans écrivain :
  - La lecture devient non bloquante

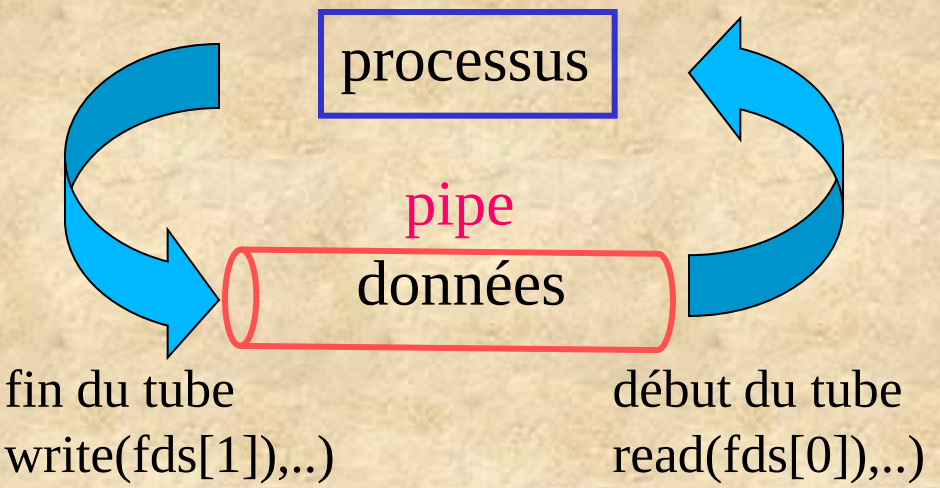
# Tubes anonymes : primitives

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

- pipe() crée un tampon que l'on peut écrire d'un coté et lire de l'autre de manière asynchrone
- int pipe(int filedesc[2]) renvoie 2 filedescriptors pour lire et écrire
- Demande la création d'un tube. Les descripteurs sont stockés dans le tableau d'entiers.
  - p[0] : descripteur en lecture
  - p[1] : descripteur en écriture
- par héritage : un processus fils hérite à sa création des descripteurs que possède son père, en particulier des descripteurs de tubes.

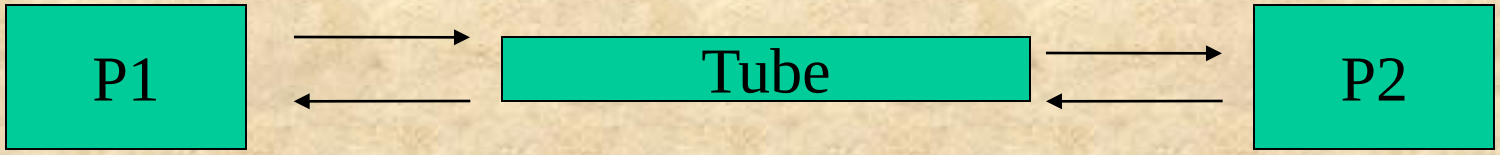
# Exemple de tube anonyme

```
void main()
{
  int fds[2];
  char buf[1024];
  strcpy(buf,"coucou!");
  pipe(fds);
  for( ; ; ){
    write(fds[1], buf, strlen("coucou"));
    read(fds[0], buf, 1024);
  }
}
```



# Lecteurs et Ecrivains multiples

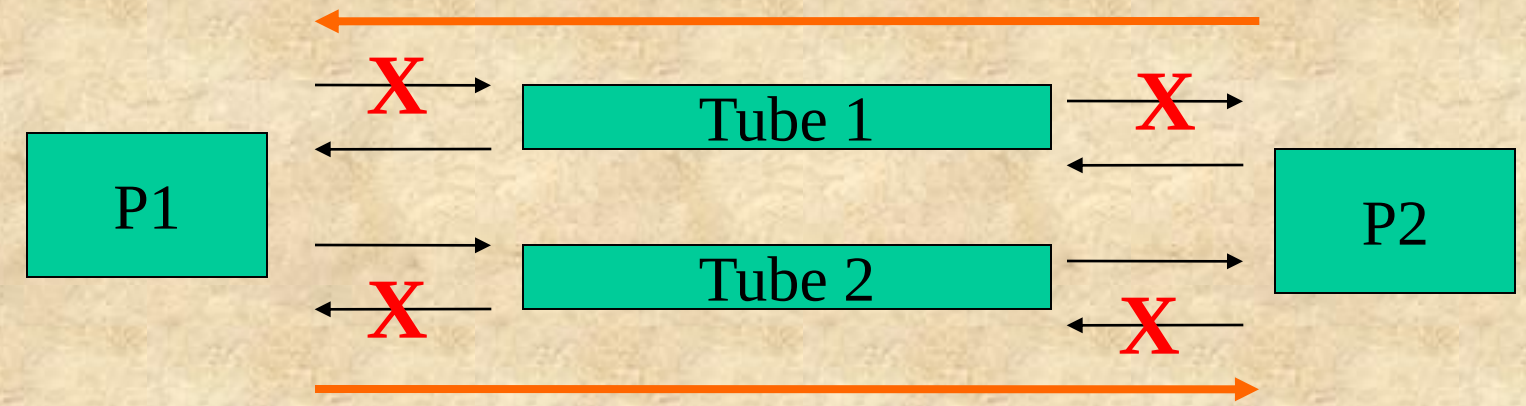
- Deux processus P1 et P2 utilisent le même tube dans les deux sens




- Situations à problème :
  - P1 écrit puis lit, donc P2 est bloqué
  - P1 écrit, P2 écrit puis lit, donc P2 lit ce qu'il a écrit.

# Fonctionnement normal

- L'utilisation unidirectionnelle permet d'être certain du comportement :



# Tubes anonymes : utilisation

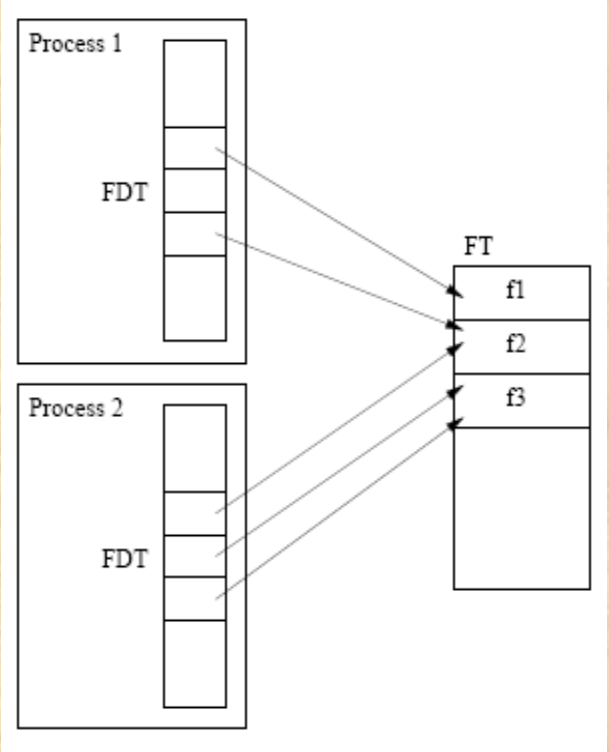
```
#include <stdio.h>
#include<unistd.h>
void main() {
   int p[2];
   char buf[20];
   pipe(p);
   switch (fork())
   {
     case -1 : exit(1);
     case 0 : close(p[1]);
              read(p[0], buf, sizeof(buf));
              printf("%s bien recu \n",buf);
              break;
     default : close(p[0]);
              write(p[1], "Bonjour", strlen("Bonjour"));
   }
}
```

# La table des descripteurs

On constate que :
deux files descriptors de deux processus différents
peuvent pointer vers le même fichier (c'est le cas de f2)

Ceci peut se produire dans le cas de fork.
En effet, le fork duplique les processus et donc
également la file descriptor table.

Il se peut également que deux files descriptors
du même processus pointent vers le même fichier,
et ce grâce aux appels système dup et dup2



# La duplication des descripteurs

- La duplication de descripteur permet à un processus d'acquérir un descripteur supplémentaire associé à une entrée existante dans la table des fichiers ouverts.

```
#include <unistd.h>
int dup (int oldfd);
int dup2(int oldfd, int newfd);
```

- **dup** et **dup2** créent une copie du descripteur de fichier *oldfd*.
- **dup** utilise le plus petit numéro inutilisé pour le nouveau descripteur (*en pratique, il vaut 3 puisque le noyau attribue le plus petit descripteur inutilisé, et que 0, 1 et 2 sont utilisés pour l'entrée standard*).
- **dup2** transforme *newfd* en une copie de *oldfd*, fermant auparavant *newfd* si besoin est.
- **dup** et **dup2** renvoient le nouveau descripteur, ou -1 s'ils échouent, auquel cas *errno* contient le code d'erreur :
  - **EBADF** : *oldfd* n'est pas un descripteur valide, ou *newfd* n'est pas dans les valeurs autorisées pour un descripteur.
  - **EMFILE** : Le processus dispose déjà du nombre maximum de descripteurs de fichiers autorisés simultanément, et tente d'en ouvrir un nouveau.

# Redirection de la sortie d'un processus vers un fichier

- La commande "**ps −a > nom_fichier** " redirige la sortie standard de la commande **ps** vers le fichier de nom **nom_fichier**
- Si un processus veut rediriger sa sortie standard (descripteur "**1**") vers un fichier dont il a accès par descripteur, il suffit :
  - il ouvre le fichier qui doit servir de nouvelle sortie, par exemple avec *open (*Cela lui donne un nouveau descripteur de fichier, en pratique, il vaut 3)
  - il ferme ensuite le descripteur 1 (la sortie),
  - il duplique avec *dup* le descripteur obtenu par *open* et le système lui attribue le plus petit numéro disponible (ici "**1** ")
  - il ferme ensuite le descripteur obtenu avec open
- Le processus a donc de nouveau trois descripteurs de fichiers 0, 1 et 2, mais le descripteur 1 désigne maintenant le fichier ouvert avec open
- Enfin les écritures standard (**fwrite**, **printf**, ...) iront écrire directement dans le fichier.

# Tubes nommés (FIFO)

- Fichier physique de type 'p' (existence d'un nœud)
- Ils permettent de transmettre des données entre des processus qui ne sont pas attachés par des liens de parenté.
- Le shell utilise la commande **mknod** et plus généralement (X/Open) la commande **mkfifo**

**$> mkfifo nom fichier**

- En "C" on utilisera l'interface suivante (**mknod** existe aussi mais n'est pas conseillée pour la portabilité du code)

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

*nomfichier :* définit le chemin d'accès au tube nommé

*mode :* les droits d'accès des différents utilisateurs à cet objet (S_IRUSR, S_IWUSR, S_IXUSR).

La valeur renvoyée par *mkfifo* est **0** s'il réussit, ou -1 s'il échoue, auquel cas *errno* contient le code d'erreur

# Tubes nommés : comportement par défaut

- Tube vide :
  - Lecture bloquante (sauf si ouverture en O_NDELAY)
- Tube non vide :
  - Attente de suffisamment de données à lire.

# Tubes nommés : utilisation

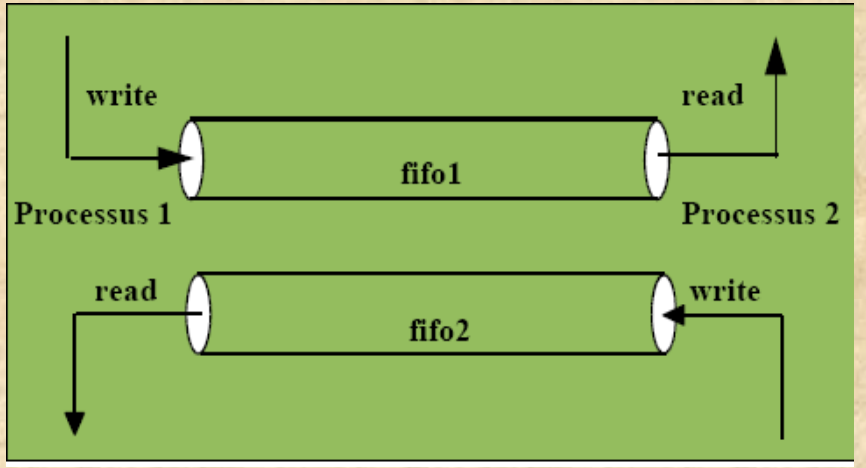
- Séquence classique :
  - Un processus en lecture : open(O_RDONLY)
  - Un processus en écriture : open(O_WRONLY)
  - L'ouverture d'un tube nommé est bloquante par défaut (on utilise O_NONBLOCK sinon).
  - …

# Tubes nommés : primitives

- Pour accéder à un tube nommé un processus devra faire un "open" sur le fichier correspondant.
- C'est un moyen pour deux processus de **faire un point de rendez–vous**.
- En effet il suffit que l'un demande l'ouverture en écriture et l'autre en lecture. Ils seront synchronisés par le système sur le deuxième "**open**".
- Il faudra néanmoins, lors de dialogues dans les deux sens entre deux processus, s'assurer que les ordres d'ouverture sont faits dans le bon sens, sinon c'est l'**interblocage** !

# Exemple de séquence d'ouverture correcte

```
/* pour le processus 1 */
int d_lecture, d_ecriture;
...
d_ecriture=open("fifo1", O_WRONLY);
d_lecture=open("fifo2", O_RDONLY);
```




```
/* pour le processus 2 */
int d_lecture, d_ecriture;
...
d_ecriture=open("fifo2", O_WRONLY);
d_lecture=open("fifo1", O_RDONLY);
```

# Manipulation des fichiers en C (open)

- Les fichiers seront toujours manipulés avec des primitives bas niveau (non bufferisées) :

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

int desc;
int open (const char * ref, int mode_ouverture, mode_t droits);
//demande une nouvelle entrée dans la table des des fichiers ouverts du système

desc = open (ref, mode_ouverture, droits);
```

# Paramètre mode_ouverture

Le paramètre mode_ouverture est construit par disjonction bit à bit des constantes définies dans fcntl.h

| | |
|---|---|
| O_RDONLY | En lecture |
| O_WRONLY | En écriture |
| O_RDWR | En lecture est en écriture |
| O_NONBLOCK | Ouverture dans un mode non bloquant |
| O_APPEND | En ajout à la fin du fichier |
| O_CREAT | Crée fichier s'il n'existe pas |
| O_TRUNC | Remet le fichier à 0 s'il existe |
| O_EXCL | Echoue si le fichier existe |

# Manipulation des fichiers en C (close)

- Les fichiers seront toujours manipulés avec des primitives bas niveau (non bufferisées) :

#include <unistd.h>

Cette primitive permet de libérer un descripteur dans la tables des descripteurs

int **close** (int *desc*)

Ces primitives permettent la lecture et l'ecriture dans un fichier de descripteur *desc*

ssize_t **read** (int *desc*, void **ptr*, size_t *nb_octets*)

*nb_octets* : le nombre (int) d'octets que l'utilisateur voudrait lire

*ssize_t* : reçoit le nombre d'octets réellement lus

la fonction read détecte la fin de fichier et renvoie -1.

ssize_t **write** (int *desc*, void **ptr*, size_t *nb_octets*)

Modifier l'offset par rapport à une origine dans le fichier de descripteur *desc*

off_t **lseek** (int *desc*, off_t *offset*, int *origine*)

# Fonction **stat**

La fonction **stat** permet de récupérer des informations sur un fichier dont on a l'accès en lecture des répertoires intervenant dans la référence de celui-ci. Il faut passer à cette fonction, par adresse, une structure dans laquelle on récupère les données sur le fichier de référence Ref.

#include <sys/stat.h>

int **stat**(const char **Ref*, struct stat **ptr_structure*)

int **fstat**(const int *desc*, struct stat **ptr_structure*)

où la structure stat est définie de la façon suivante dans <sys/stat.h>

# Structure **stat**

**struct stat**

{

**dev_t** st_dev; identification du disque logique

**ino_t** st_ino; numéro d'inode du fichier

**mode_t** st_mode; type (sur 4 bits) et droits d'accès aux fichiers (sur 12 bits).

**nlink_t** st_nlink; nombre de liens physiques

**uid_t** st_uid; propriétaire

**gid_t** st_gid; groupe du propriétaire

**off_t** st_size; taille

**time_t** st_atime; date de dernier accès

**time_t** st_mtime; date de dernière modification

**time_t** st_ctime; date de dernière modification de l'inode

}

Le champ st_mode est exprimé sur 12 bits (dont 4 pour le type du fichier). Une valeur de ce champ est une combinaison logique par l'opérateur de disjonction | des constantes suivantes:

# Le type st_mode

| Nom symbolique du bit | Interprétation du bit |
|---|---|
| **S_ISUID, S_ISGID** | le set_uid et le set_gid<br>un autre bit pour le sticky bit |
| **S_IRUSR**<br>**S_IWUSR**<br>**S_IXUSR**<br>**S_IRWXU** | lecture par le propriétaire<br>écriture par le propriétaire<br>exécution par le propriétaire<br>lecture, écriture et exécution par le propriétaire |
| **S_IRGRP**<br>**S_IWGRP**<br>**S_IXGRP**<br>**S_IRWXG** | lecture par les membres du groupe propriétaire<br>écriture par les membres du groupe propriétaire<br>exécution par les membres du groupe propriétaire<br>lecture, écriture et exécution par le groupe |
| **S_IROTH**<br>**S_IWOTH**<br>**S_IXOTH**<br>**S_IRWXO** | lecture par les autres<br>écriture par les autres<br>exécution par les autres<br>lecture, écriture et exécution par les autres |

# Le type st_mode

- S_IRWXU = S_IRUSR | S_IWUSR | S_IXUSR
- S_IRWXG = S_IRGRP | S_IWGRP | S_IXGRP
- S_IRWXO = S_IROTH | S_IWOTH | S_IXOTH

```
#include <sys/types.h>
#include <sys/stat.h>
int main( void ){
        mode_t mode ;
        mode = S_IRUSR|S_IWUSR|S_IRGRP ; /* rw pour le proprietaire, r pour le groupe */
        if( mkfifo("`fifo1'", mode) == -1) perror( "mkfifo(fifo1)" ) ;
        else fprintf( stderr, "fifo1 creee\n'" ) ;
        if( mkfifo("`fifo2'", mode) == -1) perror( "mkfifo(fifo2)" ) ;
        else fprintf( stderr, "fifo2 creee\n'" ) ;
        return 0 ;
}
```

# Le type st_mode

Les 4 bits du types de fichiers sont :

- les fichiers réguliers sont les fichiers de caractères sans aucune organisation particulière. La taille du fichier permet au système de connaître la fin de fichier.
- les répertoires sont des fichiers munis d'une structure permettant d'associer aux noms internes des noms externes ;
- les tubes nommes sont des fichiers en structure fifo.
- les fichiers spéciaux sont les associes aux ressources du système. Ils ont des liens dans le répertoire  /dev.

# Exemple d’utilisation de **stat**

struct stat *variable*;

int *retour*;

*retour* = stat("myfic",&*variable*);

if (*retour* != -1) printf("L'utilisateur est : %d ",*variable*.st_uid);

else perror("stat:");

FIN