Université de ***Batna 2***

Communication de processus : Signaux & Tubes

Département d'Informatique

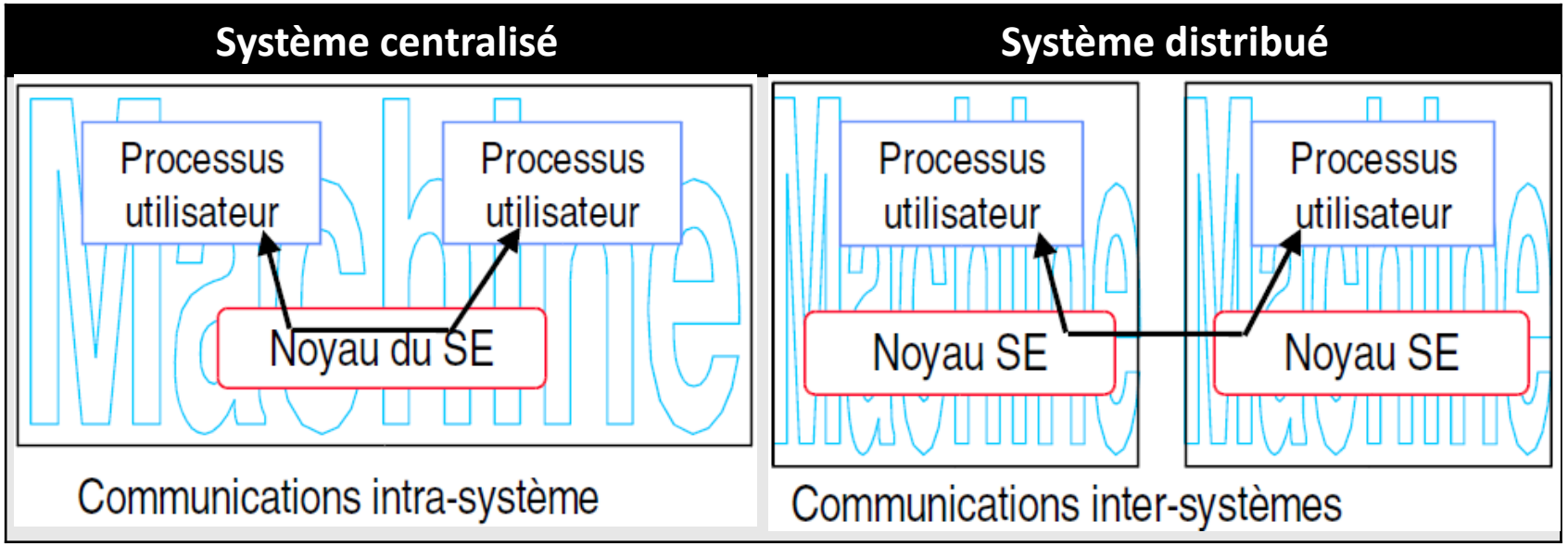
Master RSD/ISI

Plan

Introduction

Introduction

- **Schéma de communication de processus**
  - Communication intra-système
  - Communication inter-systèmes



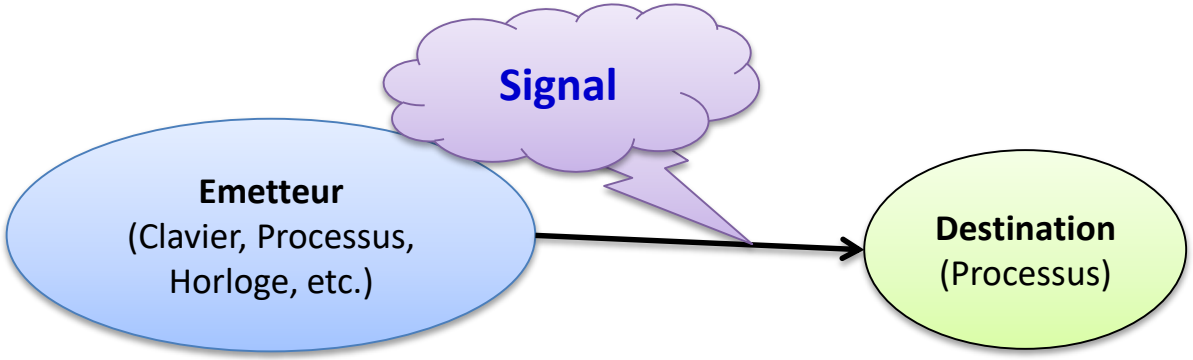
Introduction

- **Schéma de communication de processus**
  - Plusieurs mécanismes intra-système Unix peuvent être exploités afin d'assurer la communication entre les processus à savoir:
    - Signaux & Tubes
    - IPC (InterProcess Communication)
      - Files de Messages
      - Segment de données partagées
      - Sémaphores
- **Dans ce chapitre, nous nous intéresserons uniquement à la communication par Signaux & Tubes**

SIGNAUX

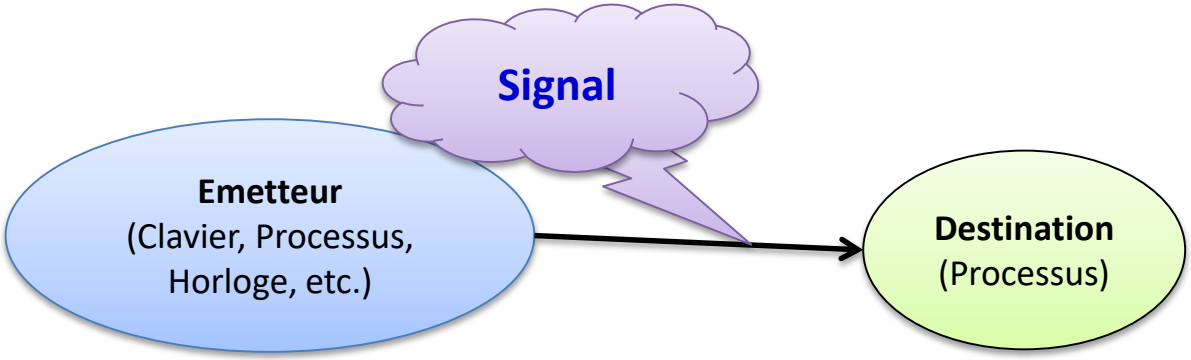
Signaux

- **Principe**
  - Un signal est « un message très court » qu'un processus peut envoyer à un autre processus, pour lui dire qu'un « événement particulier est arrivé »
    - Interruption : événement extérieur au processus (Ex: Frappe au clavier, Signal envoyé par processus, etc.)
    - Déroutement :événement intérieur au processus généré par le hard (Ex: Division par zéro, violation mémoire, etc.)
  - Un signal émis par un processus en direction d'un autre transite toujours par le système d'exploitation. C'est lui qui délivre le signal au processus destinataire.



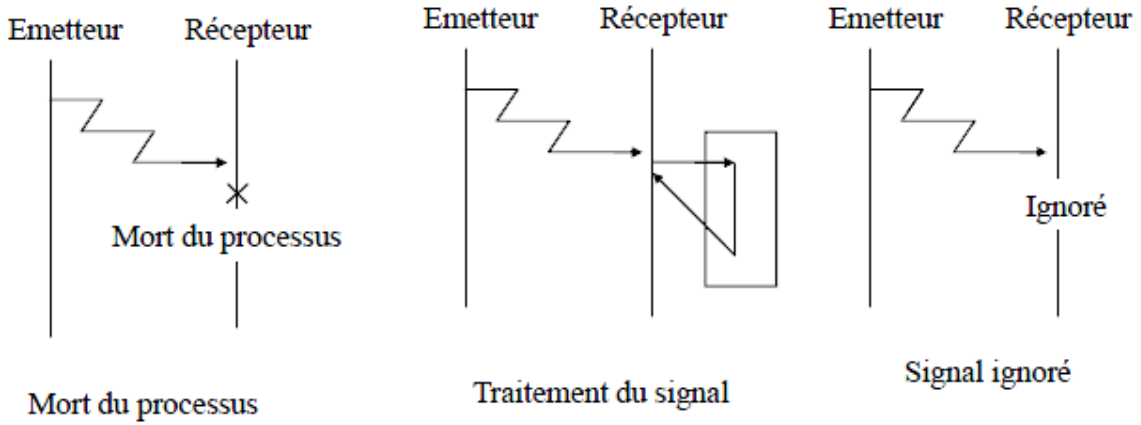
Signaux

- **Principe**
  - L'émission du signal est volontaire et donc parfaitement prévisible dans le déroulement chronologique du processus émetteur : c'est pour lui un événement synchrone.
  - la réception de ce signal par le processus récepteur peut arriver à n'importe quel moment, c'est donc un événement asynchrone.



Signaux

- **Effets d'un signal**
  - Trois types de comportements peuvent être associés à un signal :
    - Par défaut (SIG_DFL): acceptation du comportement prédéfinis par le système d'exploitation (active une routine système);
    - Déroutement (Traitement Spécifique) : activation d'un Handler décrit par le programmeur (active une routine définis par le programmeur).
    - Ignorance volontaire (SIG_IGN) : aucun comportement n'est absolument associé au signal;



Signaux

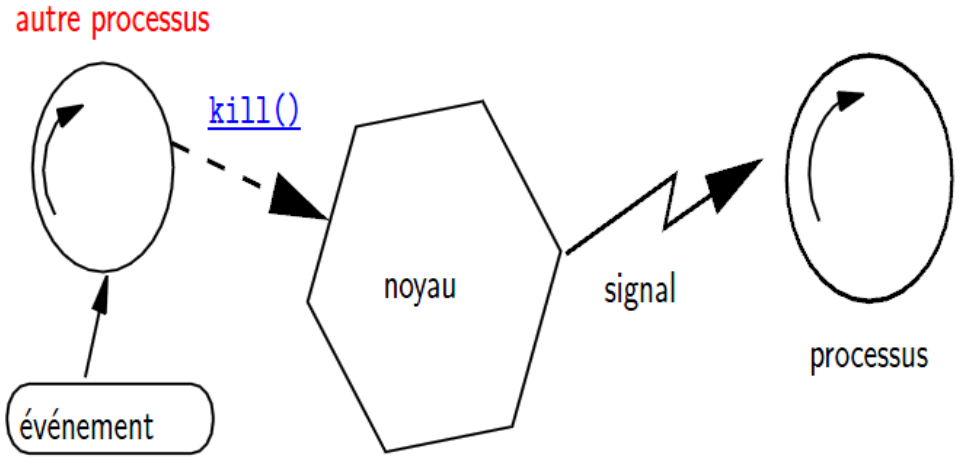
- **Types de signaux :** Sous Unix, Il y a 64 types de signal

Signal		Traitement par défaut
SIGHUP	Terminaison du processus leader de session	(1)
SIGINT	Frappe du car **intr** sur terminal de contrôle	(1)
SIGQUIT	Frappe du car **quit** sur terminal de contrôle	(2)
SIGILL	Détection d'une instruction illégale	(2)
SIGABRT	Terminaison provoquée en exécutant **abort**	(1)
SIGFPE	Erreur arithmétique (division par zéro, …)	(1)
SIGKILL	Signal de terminaison	(1) * non modifiable
SIGSEGV	Violation mémoire	(2)
SIGPIPE	Écriture dans un tube sans lecteur	(1)
SIGALARM	Fin de temporisation (fonction alarm)	(1)
SIGTERM	Signal de terminaison	(1)
SIGUSR1	Signal émis par un processus utilisateur	(1)
SIGUSR2	Signal émis par un processus utilisateur	(1)
SIGCHLD	Terminaison d'un fils	(3)
SIGSTOP	Signal de suspension	(4) *
SIGSUSP	Frappe du car susp sur terminal de contrôle	(4)
SIGCONT	Signal de continuation d'un signal d'un processus stoppé	(5) *

- La commande kill –l donne la liste des signaux du système
- **Traitements par défaut**: (1) terminaison de processus (2) terminaison de processus avec image mémoire (fichier core) (3) signal ignoré (sans effets) (4) suspension du processus, (5) continuation (reprise d'un processus stoppé).

Signaux

- **Envoie d'un signal**
  - C'est le noyau qui achemine le signal aux processus (identifiés par leurs **pids**).
    - Un autre processus, du même propriétaire (super-utilisateur), au moyen de l'appel systeme kill(), peut envoyer un signal à un processus ou un groupe de processus.



    - L'envoie d'un signal à un processus zombi est sans effet sur lui;
    - L'envoie d'un signal à un processus à l'état bloqué, il le réveille et le passe à l'état prêt;

Signaux

- **Envoie d'un signal**

  #include <signal.h>

  int kill(pid_t pid, int sig);

  – Sig: est le nom d'un signal ou un entier correspondant au numéro du signal.

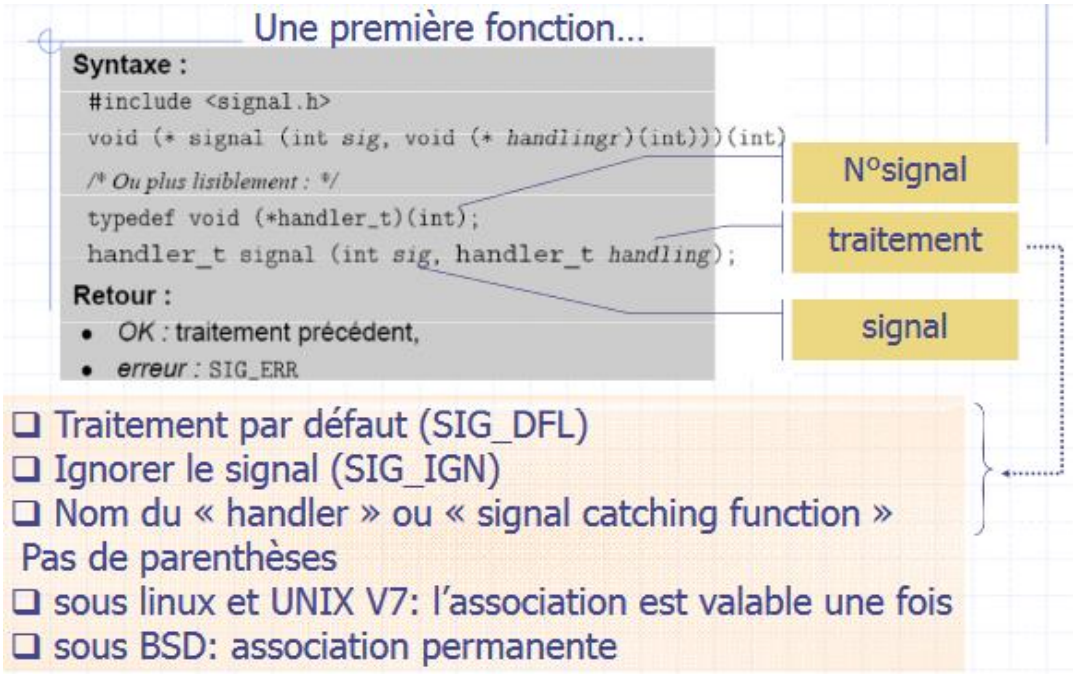
  – Les valeurs de pid peuvent être :
    - pid >0 : envoyer un signal au processus de l'identité pid
    - pid=0 : envoyer le signal à tous les processus dans le même groupe que le processus émetteur.
    - pid<-1 envoyer le signal à tous les processus du groupe ayant le numéro |pid|.
    - pid==-1, le signal est envoyé à tous les processus sauf init.

  – La valeur de retour de la fonction kill peut être :
    - -1 en cas de déroutement anormal (cas d'échec)
    - 0 si le signal a pu être envoyé

Signaux

- **Traitement spécifique d'un signal**
  - En C via ***signal*(..,..),** les signaux (autres que SIGKILL, SIGCONT et SIGSTOP) peuvent avoir un Handler spécifique installé par un processus.

```
#include<signal.h>
void signal (int sig, void *p_handler);
```



Signaux

- **Exemple**

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
int nb_req=0; int pid_fils, pid_pere;
void Hand_Pere( int sig ){
 nb_req ++;
 printf("\t le pere traite la requête
 numero %d du fils \n", nb_req);
}
void main() {
if ((pid_fils=fork()) == 0) {
/* FILS */
pid_pere = getppid();
sleep(2); /* laisser le temps au pere
de se mettre en pause */
for (i=0; i<10; i++) {
printf(« Fils envoie un signal au
pere\n");
/* demande de service */
kill (pid_pere, SIGUSR1);
}
exit(0);
}
else { /* PERE */
signal(SIGUSER1,Hand_Pere);
while(1) {
pause(); /*attend demande service */
sleep(5); /* realise le service */
}
}
}
```

TUBES

Tubes

- **Principe**
  - mécanisme de communication unidirectionnel.
  - Possède deux extrémités, une pour lire et l'autre pour y écrire
  - Les tubes permettent la communication d'un flot continu de caractères (mode Stream)
  - Un tube a une capacité finie
  - La lecture dans un tube est destructrice: l'info lue est supprimée du tube
  - La gestion des tubes se fait en mode FIFO
  - Deux types de tubes :
    - ordinaires
    - nommés

Tubes

- **Tubes Ordinaire**
  - Un tube ordinaire n’a pas de **nom**.
  - A un tube est associé un nœud du système de gestion de fichiers (son compteur de liens est = 0 car aucun répertoire ne le référence).
  - Le tube sera supprimé et le nœud correspondant libéré lorsque plus aucun processus ne l'utilise.
  - L'existence d'un tube correspond à la possession d'un descripteur acquis de deux manières:
    - un appel à la primitive de création de tube *pipe;*
    - par héritage: un processus fils hérite de son père des descripteurs de tubes, entre autres.

Tubes

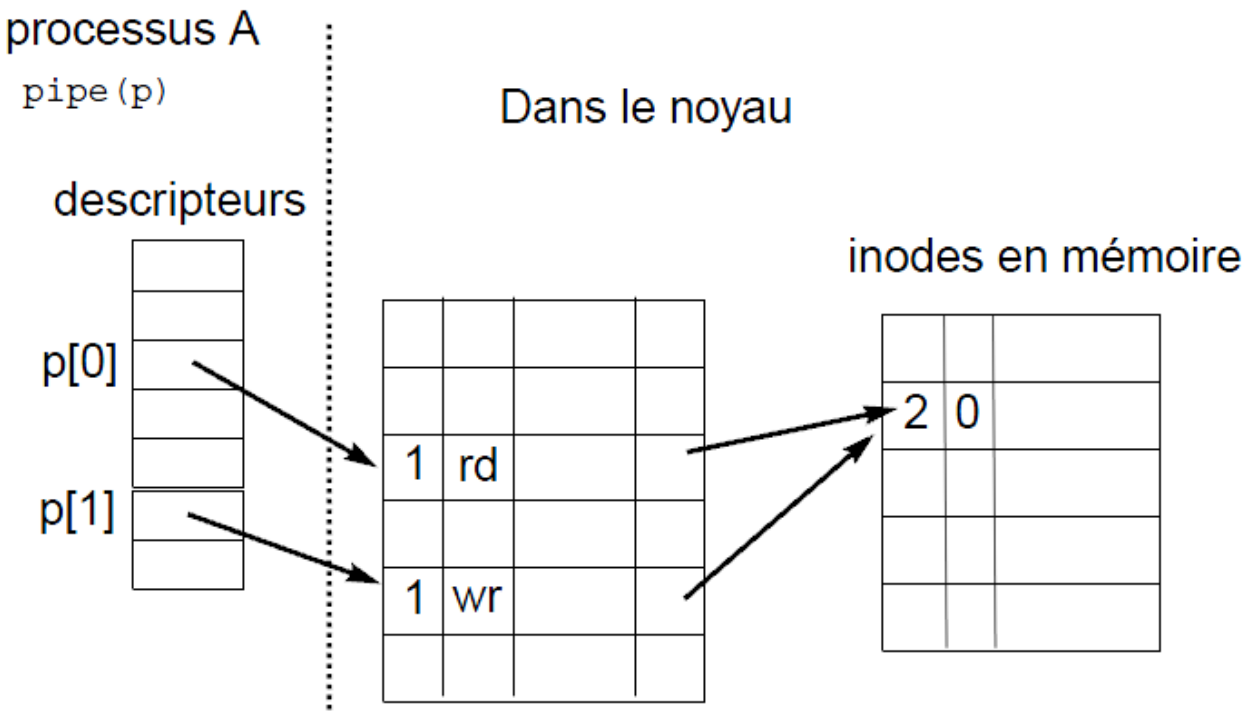
- **Tubes Ordinaire**
  - Récupération des descripteurs d'un tube
    1. **Par appel à la primitive de création pipe**
       - l'os fourni une api pour utiliser ce type de média

         #include <unistd.h>

         int pipe(int p[2]);
       - Si **création est bien passée** la valeur de retour est **0**, et pipe retourne dans **p** respectivement les descripteurs de lecture et d'écriture:
         - » p[0] est le descripteur en lecture
         - » p[1] celui en écriture.
       - Sinon la valeur de retour est **-1**

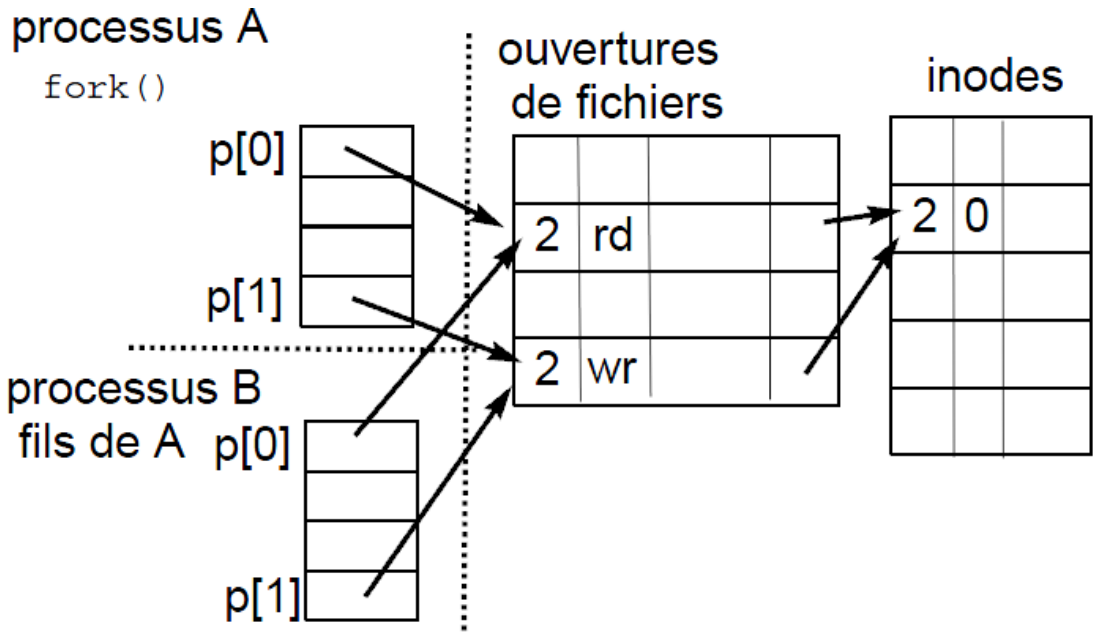
Tubes

- **Tubes Ordinaire**
  - Un nœud sur le disque des tubes,
  - Deux entrées dans la table des fichiers ouverts (1 en lecture, 1 en écriture), c.-à-d. deux descripteurs dans la table des processus appelant.



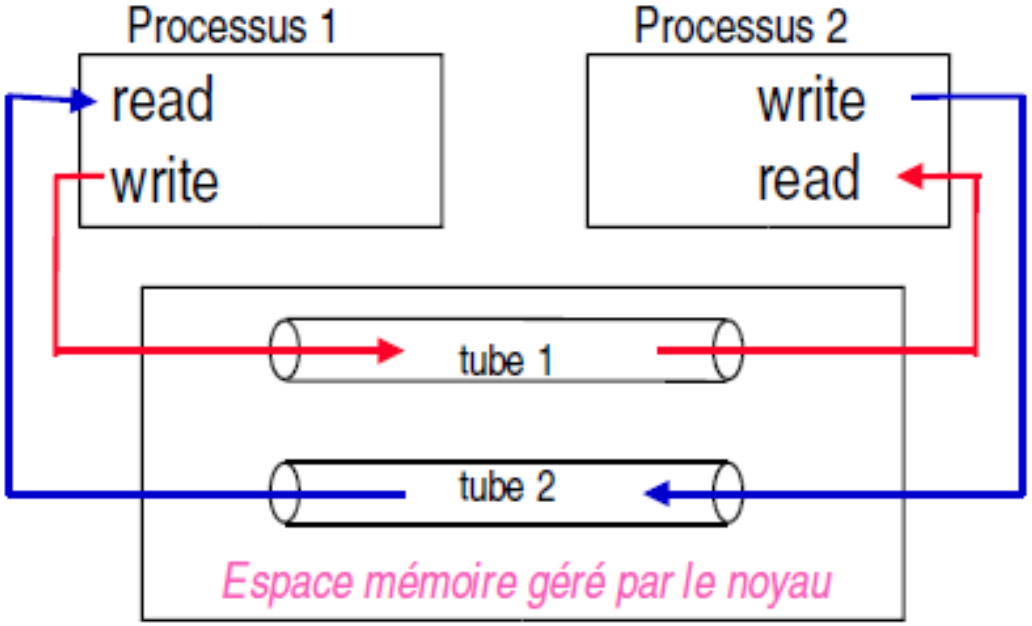
Tubes

- **Tubes Ordinaire**
  - Récupération des descripteurs d'un tube
    - **2. Par héritage**
      - Chaque processus fils héritera tous les descripteurs ouverts par son père (l'appelant de **fork()**).



Tubes

- **Tubes Ordinaire**
  - Lecture et écriture dans un tube
    - Uniquement entre processus père et fils ou bien de même père
    - Lectures destructives
    - Taille mémoire limitée



Tubes

- **Tubes Ordinaire**
  - Lecture depuis un tube
    - La lecture se fait grâce à la primitive *read():*
    - Exemple

```
char buf [10];
int nbr_car=10;
int nb_lu;
nb_lu= read (p[0], &buf, nbr_car);
```

correspond à la lecture d'au plus *nbr_car caractères qui seront rangés dans une* zone pointée par *buf.*

Tubes

- **Tubes Ordinaire**
  - Fonctionnement d'un read
    - **Si *tube non vide et contient nbr caractères* alors**
      - la primitive extrait du tube ***nb_lu=Min (nbr, nbr_car)*** *caractères et les place à* l'adresse ***buf;***
    - **Sinon**
      - **Si *nombre d'écrivains = 0,*** *la fin de fichier est atteinte* **alors**
        - » aucun caractère n'est lu et la primitive renvoie la valeur ***nb_lu=0;***
      - **Sinon (c-à-d.** *nombre d'écrivains != 0)*
        - » **Si *lecture bloquante* alors**
          - *le processus est bloqué jusqu'à tube non vide;*
        - » **Sinon (c-à-d.** lecture non bloquante)
          - le retour est immédiat et la valeur de retour est -1.
        - » **Fsi**
      - **Fsi**
    - **Fsi**

Tubes

- **Tubes Ordinaire**
  - Ecriture dans un tube :
    - L'écriture se fait à l'aide de la primitive *write*.
    - Exemple

```
int nb_ecrit;
char buf[8]="Bonjour\0"; int n=8;
nb_ecrit= write(p[1], buf, n);
```

demande l'écriture dans le tube de descripteur p[1] de n caractères accessibles à l'adresse buf.

Remarque: n doit être ≤ PIPE_BUF défini dans <limits.h>.

Tubes

- **Tubes Ordinaire**
  - Fonctionnement d'un write
    - **Si** *nombre de lecteurs dans le tube == 0* **alors**
      - le signal SIGPIPE est envoyé au processus, ayant pour handler de terminer ce processus;
    - **Sinon**
      - **si** *écriture bloquante* **alors**
        - » le retour de la primitive (avec la valeur n) n'a lieu que lorsque les n caractères ont été écrits (le processus passe à l'état bloqué dans l'attente que le tube se vide);
      - **sinon**
        - » **si** n > PIPE_BUF **alors** le retour est un nombre < n **fsi**
        - » **si** n ≤ PIPE_BUF et nbre emplacements libres dans le tube ≥ n **alors** une écriture atomique est réalisé; **fsi**
        - » **si** n ≤ PIPE_BUF et nbre emplacements libres dans le tube < n **alors** le retour est immédiat sans écriture avec la valeur de retour 0 ou -1
      - **fsi**
    - **Fsi**

Tubes

- **Tubes Ordinaire**
  - Exemple

```
#include <stdio.h>
#include <unistd.h>
int tube[2];
char buf[20];
main() {
  pipe(tube);
  if (fork()==0) { /* fils */
    close(tube[0]);
    write(tube[1], "bonjour", 7);
  } else { /* pere */
    close(tube[1]);
    read(tube[0], buf, 8);
    printf("%s  est bien reçu\n", buf);
  } }
```

```
$ ./tube_exo1
$ bonjour est bien reçu
$
```

Tubes

- **Tubes Nommés**
  - tube **unidirectionnel** (aussi appelé FIFO)
  - Ils permettent de transmettre des données entre des processus qui ne sont pas attachés par des liens de **parenté**.
  - Un tube nommé possède une référence dans le système de fichiers.
  - Pour accéder à un tube nommé un processus devra faire un "**open**" sur le fichier correspondant.
  - Si cette ouverture est faite en lecture et qu'il n'y ait aucun processus qui ait fait une ouverture en écriture, alors le processus courant est **endormi** et réciproquement avec l'ouverture en écriture.
  - 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 synchronisé par le système sur le deuxième "open".

Tubes

- **Tubes Nommés**
  - Création d'un tube nommé dans le Shell

    $ mknod nomtube p
  - Création d'un tube nommé dans un programme

    # include <sys/types.h>

    #include <sys/stat.h>

    int mkfifo (const char *ref, mode_t mode);

      - ref : indique l'emplacement du tube
      - mode : indique les permissions d'accès (droits d'accès)
    - La valeur renvoyée par **mkfifo**() est **0** si elle réussit, ou **-1** si elle échoue
    - Exemple : mkfifo ("/tmp/tube ",010666) ⇔ mknod ("/tmp/tube" ,010666|S_IFIFO,0);

Tubes

- **Tubes Nommés**
  - Ouverture d'un tube nommé
    - nous devons faire un appelle à la primitive **open**, qui permet d'acquérir un descripteur sur un tube nommé.
    - **open** est **bloquante** : le 1er processus demandant l'ouverture est bloqué jusqu'à ce que le second le demande aussi.

      NB. Si vous utilisez plusieurs tubes dans votre programme, il faut s'assurer que les ouvertures se font dans le même ordre afin ***d'éviter*** *les situations* ***d'inter-blocage***.

      - Syntaxe de open :

        int desc = open (nom_du_fichier, mode)

        - mode = O_RDONLY : ouverture en lecture
        - mode = O_WRONLY ouverture en écriture
    - Exemple

      int d_ecrit = ***open*** *(" fifo1" , O_WRONLY); /*ouverture en écriture*

      int d_lect = ***open*** *(" fifo2" , O_RONLY);/* ouverture en lecture*

Tubes

- **Tubes Nommés**
  - Lecture et écriture dans un tube nommé
    - **read** (desc, buf, nb) lecture dans le tube
      - si O_NDELAY à l'ouverture
        - » retour code erreur si lecture dans un tube vide
      - sinon
        - » processus bloqué si tube vide, attente tube suffisamment de données à lire
    - **write** (desc, buf, nb) écriture dans le tube
      - si O_NDELAY à l'ouverture
        - » retour code erreur si tube plein
      - sinon
        - » processus bloqué si tube plein, attente tube suffisamment vide pour écrire

Tubes

- **Tubes Nommés (exemple)**

Emetteur (fichier emet)

```
#include <stdio.h>
#include <sys/fcntl.h>
main( ) {
    int fd;
    fd = open(''toto'', O_WRONLY);
    write(fd, ''un message'', 10);
    printf(''%d]> fin ecriture'‘, getpid() );
}
```

Récepteur (fichier recoit)

```
#include <stdio.h>
#include <sys/fcntl.h>
main( ) {
    int fd; char buf[50];
    fd = open(''toto'', O_RDONLY);
    read(fd, buf, sizeof(buf));
    printf(''%d]> recu'', %s \n'‘,getpid(), buf);
}
```

Exécution :

$ /etc/mknod toto p (création d’un tube nommé à partir du shell)
$ recoit & (exécution du processus recoit) [1] 2190
$ emet (exécution du processus emet)
2191]> fin ecriture
[1] Done recoit
2190]> recu: un message

Fin

Merci pour votre attention

Droits d’accès

- **Deux solutions :**
  - La première, c'est de lire la documentation du fichier sys/stat.h. C'est en anglais voir la section « File mode bits ». Vous y trouverez des constantes
  - correspondant aux droits d'accès (S_IRUSR, S_IWUSR...). Vous pouvez combiner ces constantes avec le symbole « | ».
  - Deuxième solution, pour les allergiques à l'anglais et pour ceux qui savent faire des additions :
  - fabriquer des valeurs de droits tout seul.
  - Les valeurs des droits comprennent quatre chiffres, sont sous mode octal : elles commencent donc par un zéro.
  - Le premier chiffre correspond au propriétaire (vous en l'occurrence),
  - le deuxième correspond au groupe du propriétaire
  - et le troisième correspond à tous les autres.
  - Une valeur est attribuée à chaque permission : 1 pour l'exécution, 2 pour l'écriture et 4 pour la lecture.
  - On fait une somme quand on veut combiner plusieurs permissions.
  - Un petit exemple, Pour attribuer toutes les permissions à vous, seule la lecture pour le groupe, et aucune pour les autres,
  - la valeur correspondante est 0720 (Premier chiffre = 0 (obligatoire) ; Deuxième chiffre = 1 (Exécution) + 2 (Ecriture) + 4 (Lecture) ;
  - Troisième chiffre = 0 (aucune permission)).

Droits d’accès

- **File mode bits:**
  - S_IRWXU
  - read, write, execute/search by owner
  - S_IRUSR
  - read permission, owner
  - S_IWUSR
  - write permission, owner
  - S_IXUSR
  - execute/search permission, owner
  - S_IRWXG
  - read, write, execute/search by group
  - S_IRGRP
  - read permission, group
  - S_IWGRP
  - write permission, group
  - S_IXGRP
  - execute/search permission, group
  - S_IRWXO
  - read, write, execute/search by others
  - S_IROTH
  - read permission, others
  - S_IWOTH
  - write permission, others
  - S_IXOTH
  - execute/search permission, others
  - S_ISUID
  - set-user-ID on execution
  - S_ISGID
  - set-group-ID on execution
  - S_ISVTX