Les tubes

Cyril Rabat
cyril.rabat@univ-reims.fr

Licence 3 Informatique - Info0601 - Systèmes d'exploitation - concepts avancés

2021-2022

Cours n°6

Les tubes nommés et anonymes

Redirection, modification des propriétés des descripteurs de fichier

Version 9 janvier 2022

Table des matières

Communications inter-processus

- Processus :
  - Mémoire individuelle
  - Indépendants les uns des autres
- Comment faire communiquer des processus entre eux ?
  - Signaux standards ou signaux temps réel :
    ↪ Données limitées, pertes possibles (positionnement, etc.)
  - Les fichiers :
    ↪ Grosses quantités de données mais ralentissements dus aux accès disques
  - Autres mécanismes mis à disposition par le système :
    ↪ Mémoire partagée, files de messages (voir les cours suivants)
  - Les *sockets* :
    ↪ Processus distants (ou entre processus locaux)

Les tubes

- Un tube = un pipeline :
  ↪ Communication unidirectionnelle
  ↪ Sens choisi une bonne fois pour toute
- De chaque côté, un processus :
  ↪ L'un en lecture, l'autre en écriture
- Gestion au niveau du système de fichiers :
  - Un tube = 2 descripteurs de fichier
  - Un pour la lecture, l'autre pour l'écriture
- Deux types de tubes :
  - Tubes anonymes
  - Tubes nommés

Il est possible d'utiliser un seul tube pour plusieurs processus : attention aux synchronisations !

Les données

- Flots d'octets :
  ↪ Comme pour les fichiers avec `read` et `write`
- Les tubes sont FIFO (*First In First Out*) :
  - Données plus anciennes lues en premier
  - Supprimées au fur-et-à-mesure de la lecture
- Pas possible de mettre une quantité de données trop grande :
  - Limitée par la taille du tampon associé au tube :
    ↪ 65535o généralement sous *Linux*
  - Écriture bloquante si limite atteinte

Impossible d'utiliser `lseek` dans les tubes !

Lecture dans un tube

- Utilisation de la primitive `read`
- Si des données sont présentes :
  - Placées dans le tampon spécifié et supprimées du tube
  - Possiblement moins de données lues qu'attendues :
    ↪ L'appel peut être interrompu
- Si aucune donnée :
  ↪ Processus bloqué !
- Si plus d'écrivain :
  ↪ Fin de fichier détectée (et `read` retourne 0)

Écriture dans un tube

- Utilisation de la primitive `write`
- Si aucun lecteur n'est présent :
  - Signal `SIG_PIPE` envoyé à l'écrivain
  - Message `Broken pipe` affiché sur le terminal
  - Possibilité d'ignorer le signal :
    ↪ Erreur `EPIPE` retournée par `write`
- Le tampon associé au tube est rempli :
  - En une seule fois ssi la taille des données à écrire est inférieure à `PIPE_BUF`
  - Sinon, en plusieurs fois (découpage arbitraire) :
    ↪ Mélange possible de données envoyées par plusieurs processus ! ! !
  - Sous *Linux*, `PIPE_BUF` vaut généralement 4096
- Comme la lecture, l'écriture est bloquante :
  ↪ Tant que toutes les données ne sont pas écrites

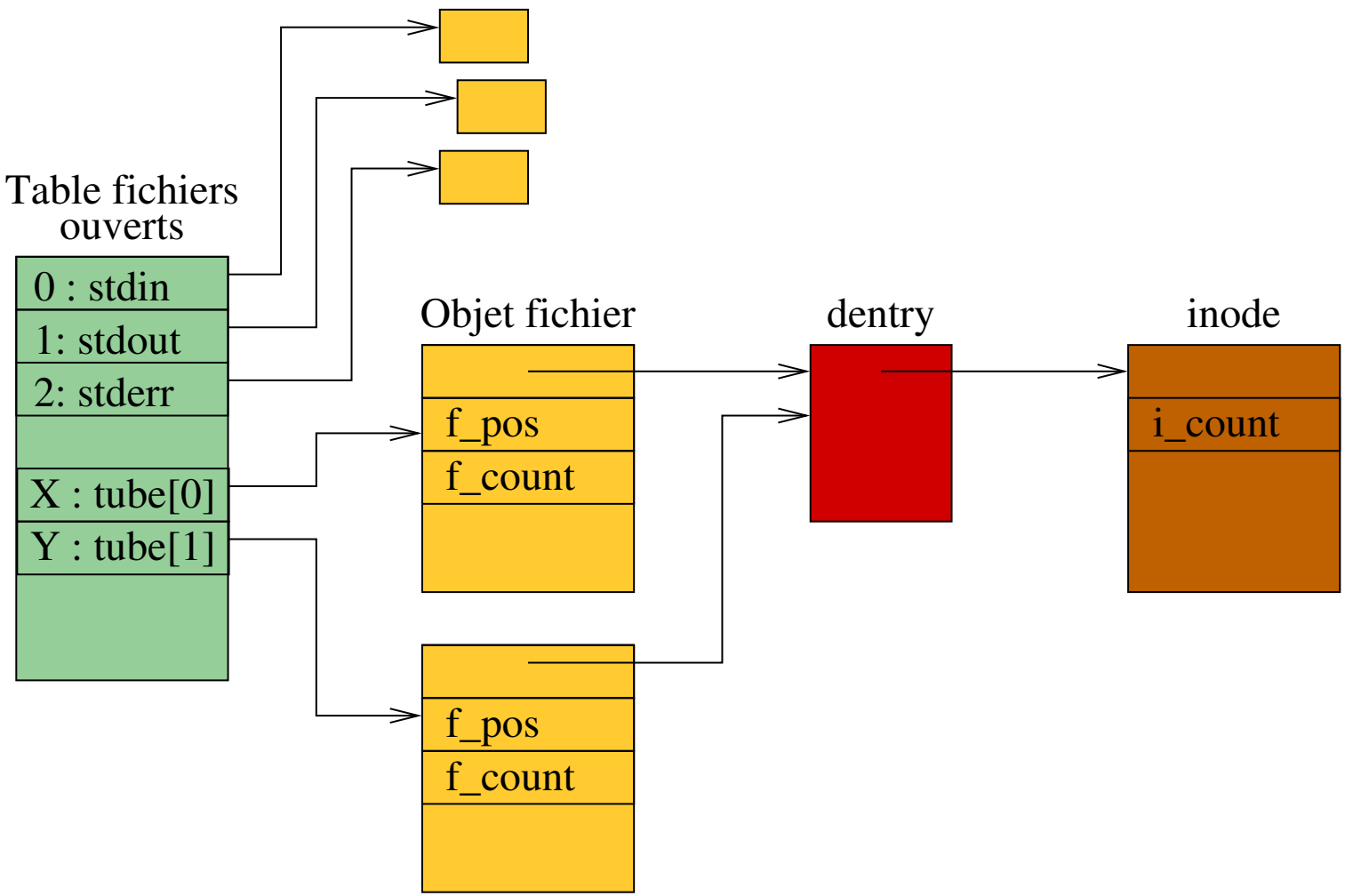
Lectures et écritures non bloquantes

- Possible de rendre les opérations de lecture/écriture non bloquantes :
  - Utilisation de l'appel système `fcntl`n
  - Option `O_NONBLOCK` à ajouter au descripteur
- Suivant la taille des données à écrire (notée n) :
  - Si `O_NONBLOCK` non présent (par défaut) :
    - Si $n \leq$ `PIPE_BUF` : données écrites en une fois
      $\hookrightarrow$ `write` peut bloquer s'il n'y a pas de place
    - Si $n >$ `PIPE_BUF` : écriture non atomique et bloquante
  - Si `O_NONBLOCK` est présent :
    - Si $n \leq$ `PIPE_BUF` : données écrites en une fois
      $\hookrightarrow$ `write` échoue s'il n'y a pas de place (erreur `EAGAIN`)
    - Si $n >$ `PIPE_BUF` : écriture non atomique
      $\hookrightarrow$ `write` échoue s'il n'y a pas de place (erreur `EAGAIN`)
      $\hookrightarrow$ Possiblement toutes les données écrites (à vérifier ! ! !)
      $\hookrightarrow$ Entrelacement possible avec d'autres écritures

Les tubes anonymes

- Pas de nom de fichier spécifié :
  - Gestion réalisée quand même par le VFS
  - Création d'un *inode* "virtuel"
  - Pas de bloc de données pointé par l'*inode*
- Nécessité de connaitre l'"objet" `fichier` associé aux descripteurs :
  $\hookrightarrow$ Possible uniquement si les processus sont affiliés
- Principe :
  1. Le processus crée un tube et connait les deux descripteurs associés
  2. Le processus crée un processus fils
  3. Le père et le fils partagent les descripteurs du tube
  4. Les descripteurs inutiles sont fermés

Représentation mémoire



Création d'un tube anonyme

- Utilisation de l'appel système `pipe` :

```
int pipe(int tube[2])
```

- En cas de réussite, deux descripteurs créés :
  ↪ `tube[0]` : utilisé pour la lecture
  ↪ `tube[1]` : utilisé pour l'écriture
- Tous les processus qui connaissent `tube[0]` peuvent lire dans le tube
- Tous les processus qui connaissent `tube[1]` peuvent écrire dans le tube

Il est fortement déconseillé de laisser des processus différents accéder simultanément au même descripteur !

Fonction `pipe`

En-tête de la fonction (S2)

- `int pipe(int tube[2])`
- *Inclusion* : `unistd.h`

Paramètre(s)

- `tube` : un tableau qui contiendra les deux descripteurs de fichier

Valeurs retournées et erreurs générées

- 0 en cas de réussite, -1 en cas d'échec
- Erreurs possibles :
  - `EMFILE` : trop de descripteurs utilisés par le processus
  - `ENFILE` : table système des tubes pleine
  - `EFAULT` : `fd` est invalide

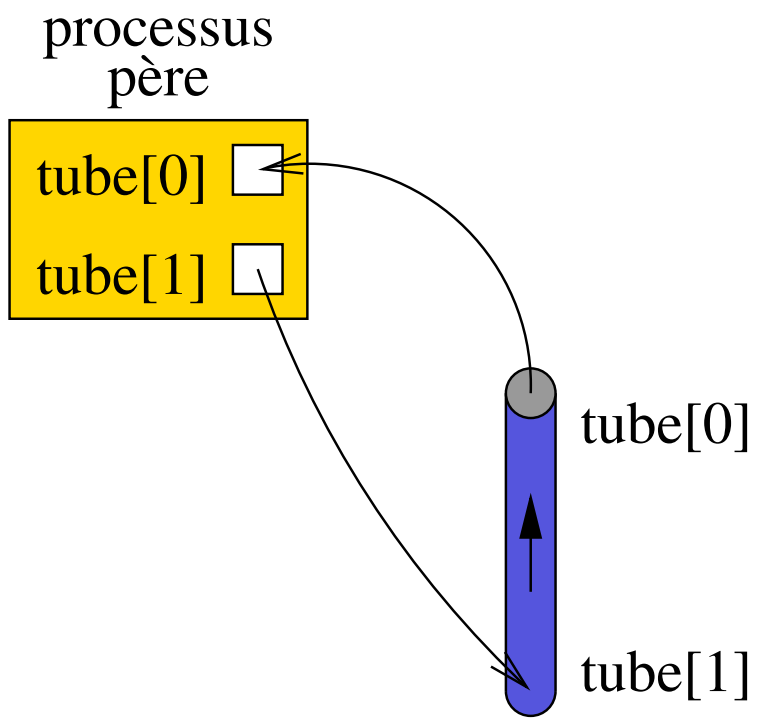
Fermeture d'un tube

- Les descripteurs sont partagés entre plusieurs processus
- Un tube est fermé quand :
  $\hookrightarrow$ TOUS les descripteurs en lecture ET en écriture sont fermés
- Utilisation de l'appel système `close`
- Attention :
  - Un descripteur fermé ne permet plus d'accéder au tube !
  - Il ne peut être régénéré !

Exemple d'utilisation

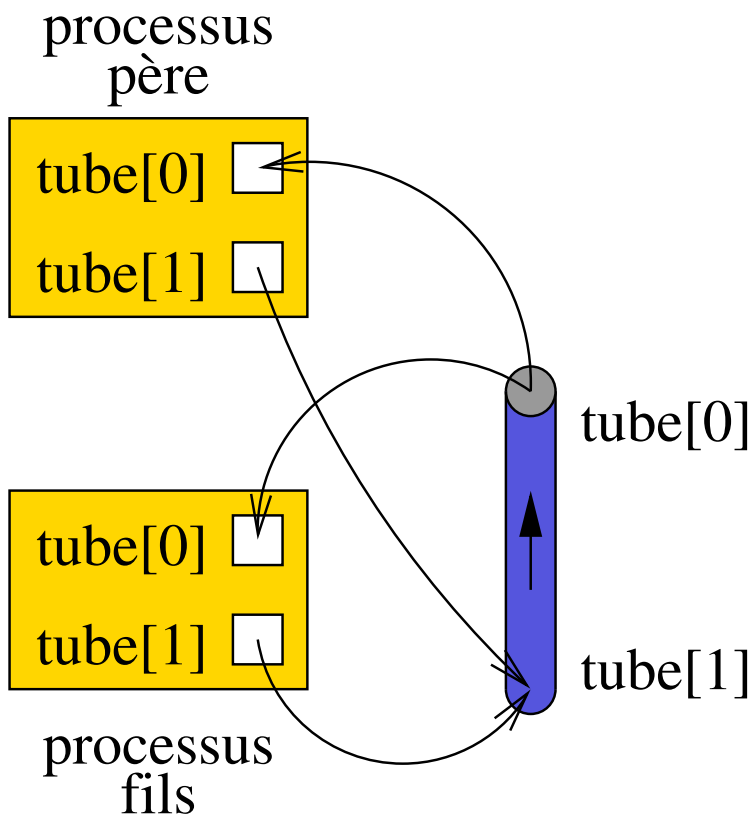
- Création d'un canal de communication d'un processus fils vers son père :
  $\hookrightarrow$ Canal *simplex* ($\neq$ *half-duplex* ou *full-duplex*)
- Principe :
  - Le père crée un tube (avec `pipe`) :
    $\hookrightarrow$ Tableau de deux descripteurs (l'un pour la lecture, l'autre pour l'écriture)
  - Création d'un fils (avec `fork`) :
    $\hookrightarrow$ Descripteurs connus par les deux processus
  - Du côté du père : fermeture du tube en écriture
  - Du côté du fils : fermeture du tube en lecture
  - Le fils peut maintenant écrire et le père lire dans le tube
- Dans notre exemple : le fils envoie 5 entiers à son père

Fonctionnement général



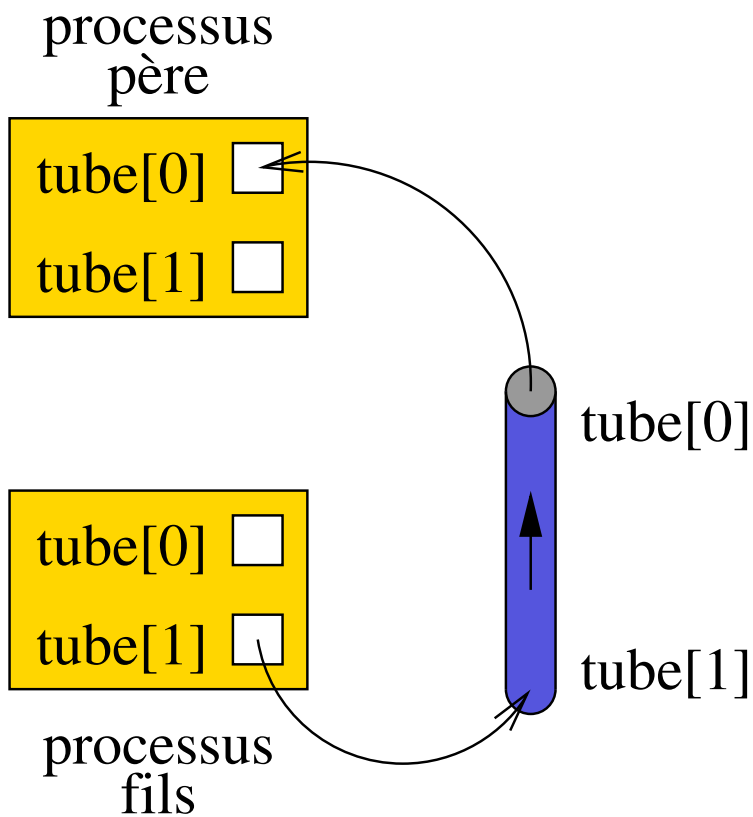
Création du tube avec la fonction "`pipe`"

Fonctionnement général



Duplication du processus père avec la fonction "`fork`"

Fonctionnement général



Fermeture des descripteurs inutiles avec la fonction "`close`"

Exemple d'utilisation : le code (1/2)

```
int main() {
  int tube[2], i, tmp;

  pipe(tube);
  if(fork() == 0)
    fils(tube);

  close(tube[1]);
  for(i = 0; i < 5; i++) {
    read(tube[0], &tmp, sizeof(int));
    printf("Pere␣:␣entier␣lu␣:␣%d\n", tmp);
    sleep(1);
  }
  close(tube[0]);

  wait(NULL);

  return EXIT_SUCCESS;
}
```

Exemple d'utilisation : le code (2/2)

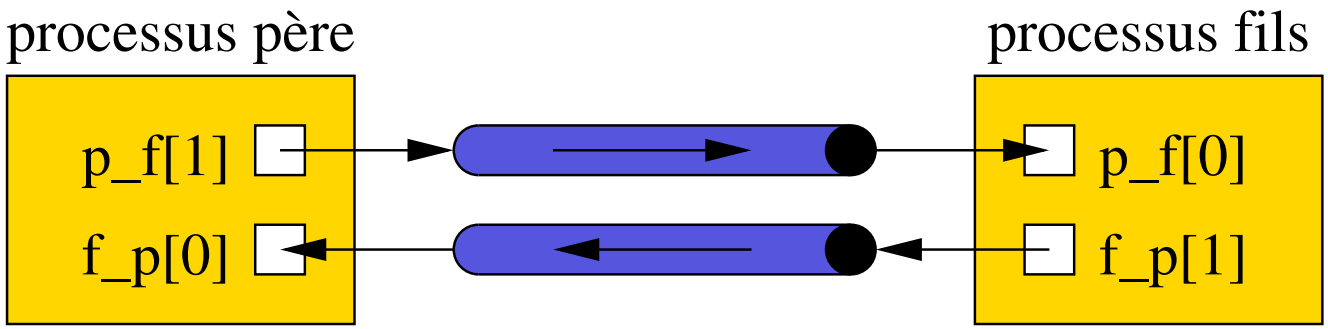
```
void fils(int tube[2]) {
  int i;

  close(tube[0]);
  for(i = 0; i < 5; i++) {
    write(tube[1], &i, sizeof(int));
    printf("Fils␣:␣entier␣envoyé␣:␣%d\n", i);
  }
  close(tube[1]);

  exit(EXIT_SUCCESS);
}
```

Possibilité pour éviter les erreurs

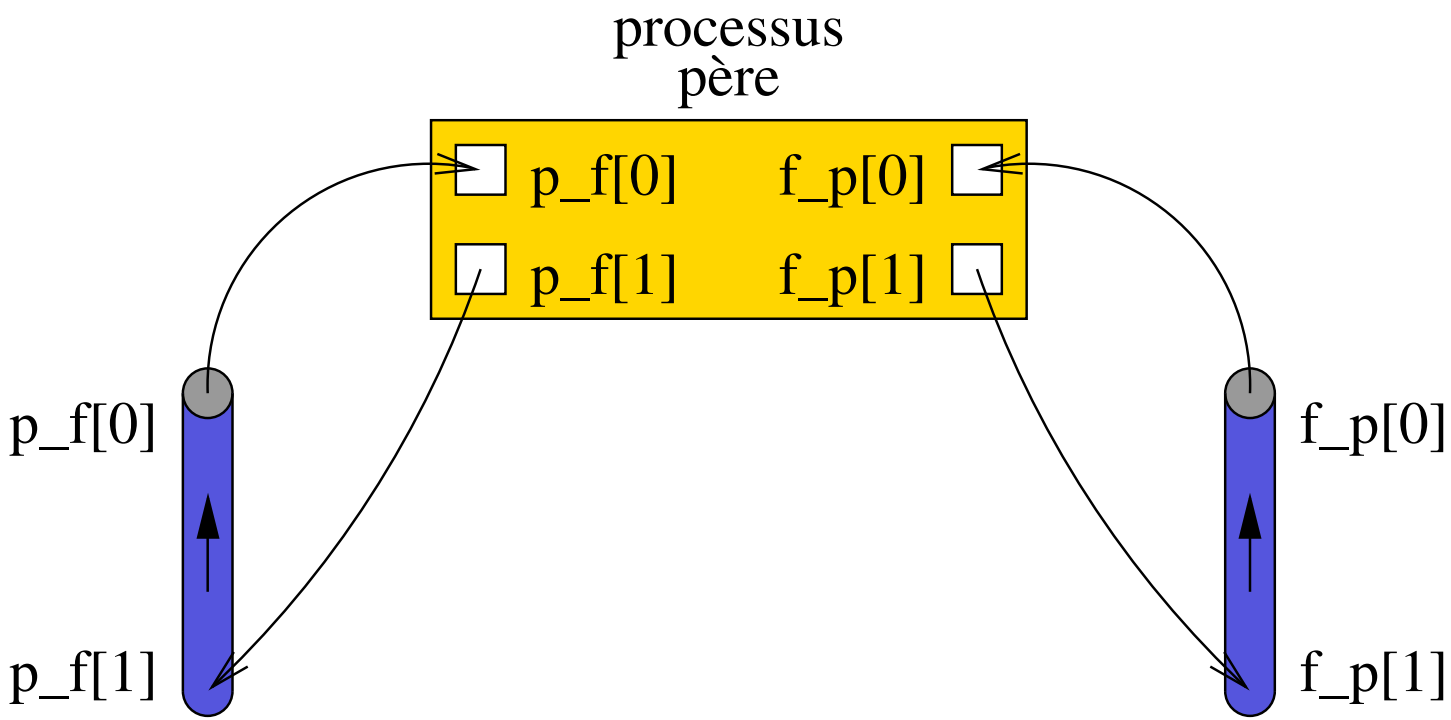
```
#define TUBE_LECTURE   0
#define TUBE_ECRITURE  1
```

Autre exemple : communication bidirectionnelle

- Un tube = communication unidirectionnelle
- Comment envoyer des données dans les deux sens ?
- Solution :
  ↪ Utilisation de deux tubes !

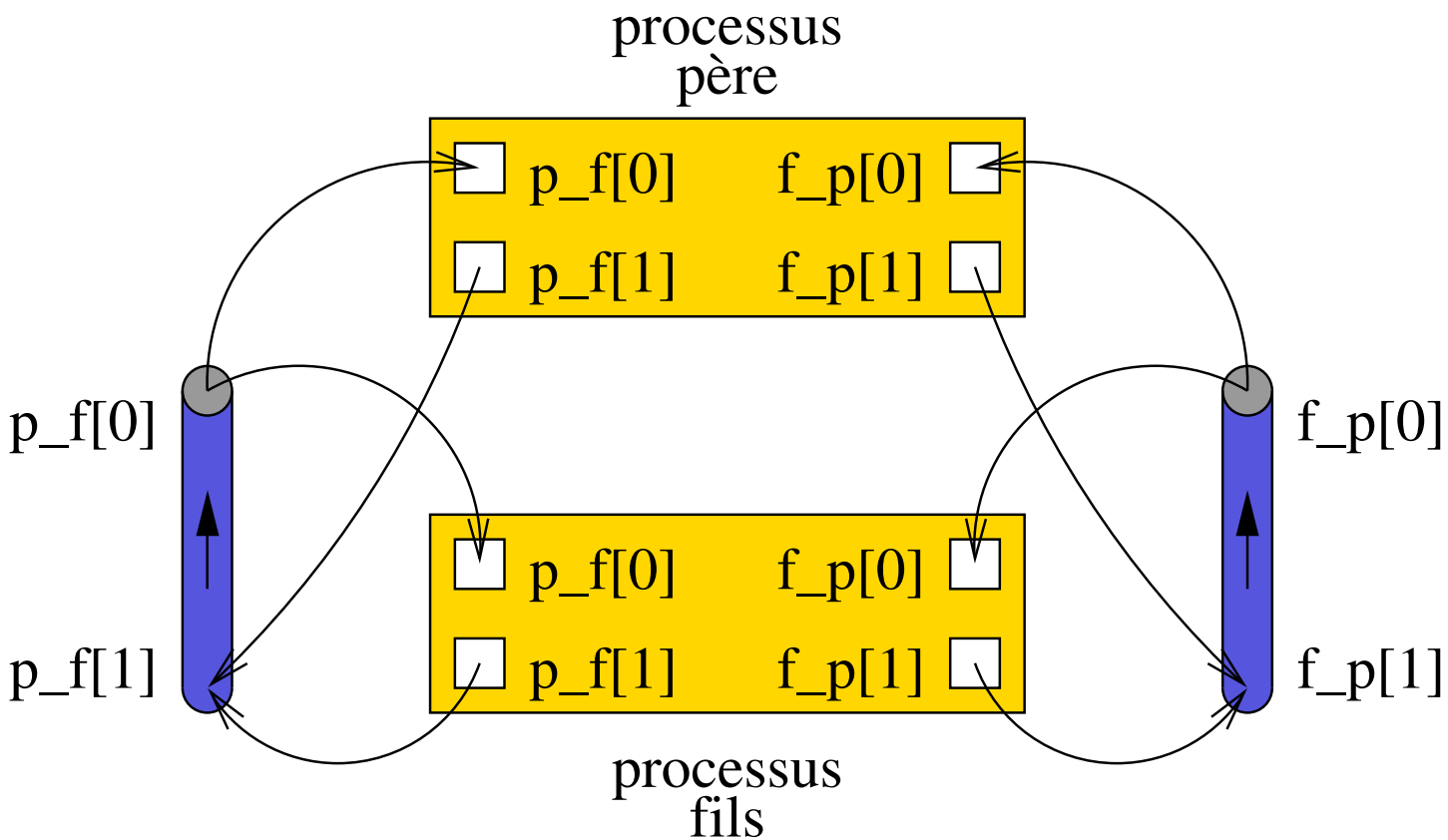


Fonctionnement général



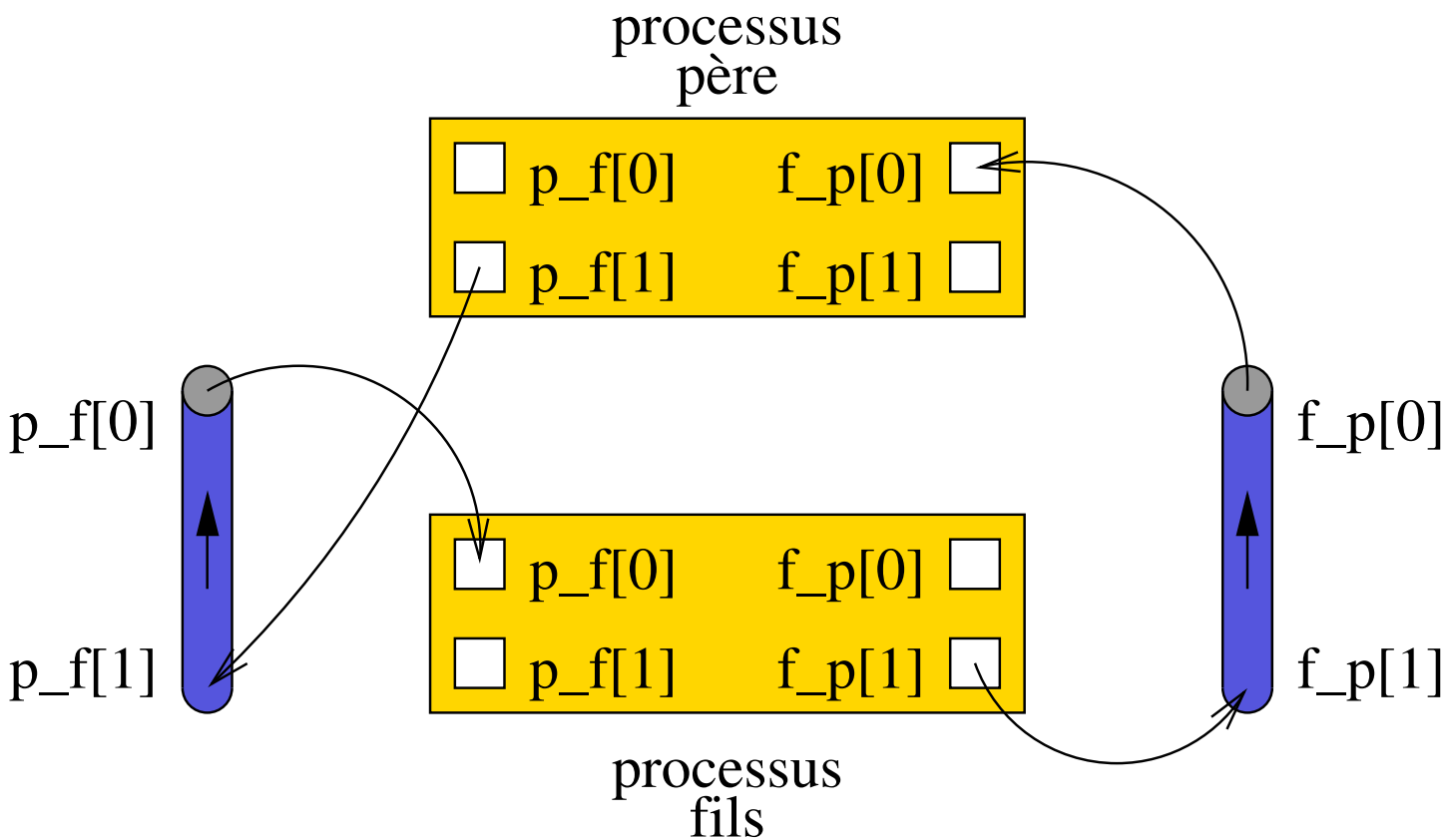
Création des tubes avec la fonction "`pipe`"

Fonctionnement général



Duplication du processus père avec la fonction "`fork`"

Fonctionnement général



Fermeture des descripteurs inutiles avec la fonction "`close`"

Exemple de communication bidirectionnelle (1/3)

```
int main() {
  int p_f[2], f_p[2], i, tmp = 1;

  pipe(p_f);
  pipe(f_p);

  if(fork() == 0)
    fils(p_f, f_p);

  close(p_f[TUBE_LECTURE]);
  close(f_p[TUBE_ECRITURE]);
  ...
```

Exemple de communication bidirectionnelle (2/3)

```
  ...
  for(i = 0; i < 5; i++) {
    write(p_f[TUBE_ECRITURE], &tmp, sizeof(int));
    printf("Pere␣:␣entier␣envoye␣:␣%d\n", tmp);

    read(f_p[TUBE_LECTURE], &tmp, sizeof(int));
    printf("Pere␣:␣entier␣lu␣:␣%d\n", tmp);
  }

  close(p_f[TUBE_ECRITURE]);
  close(f_p[TUBE_LECTURE]);

  wait(NULL);

  return EXIT_SUCCESS;
}
```

Exemple de communication bidirectionnelle (3/3)

```
void fils(int p_f[2], int f_p[2]) {
  int i, tmp;

  close(p_f[TUBE_ECRITURE]);close(f_p[TUBE_LECTURE]);

  for(i = 0; i < 5; i++) {
    read(p_f[TUBE_LECTURE], &tmp, sizeof(int));
    printf("Fils␣:␣entier␣lu␣:␣%d\n", tmp);
    tmp *= 2;
    write(f_p[TUBE_ECRITURE], &tmp, sizeof(int));
    printf("Fils␣:␣entier␣envoye␣:␣%d\n", tmp);
  }

  close(p_f[TUBE_LECTURE]);close(f_p[TUBE_ECRITURE]);

  exit(EXIT_SUCCESS);
}
```

Lectures et écritures non bloquantes

- *Rappel* : lecture par défaut bloquante !
  ↪ L'écriture aussi dans certains cas
- Solution : modifier les attributs des descripteurs créés par `pipe`
- Fonction *fcntl* :
  - Permet de récupérer les attributs actuels de descripteurs
  - Permet de les modifier, etc.
- Principe d'utilisation :
  1. Récupérer les attributs actuels du descripteur
  2. Ajouter l'attribut `O_NONBLOCK`

`fcntl` permet de faire beaucoup d'autres opérations.
Nous nous limitons ici aux tubes.

Fonction `fcntl`

En-têtes de la fonction (S2)

- `int fcntl(int fd, int cmd)`
- `int fcntl(int fd, int cmd, long arg)`
- *Inclusions* : `unistd.h` et `fcntl.h`

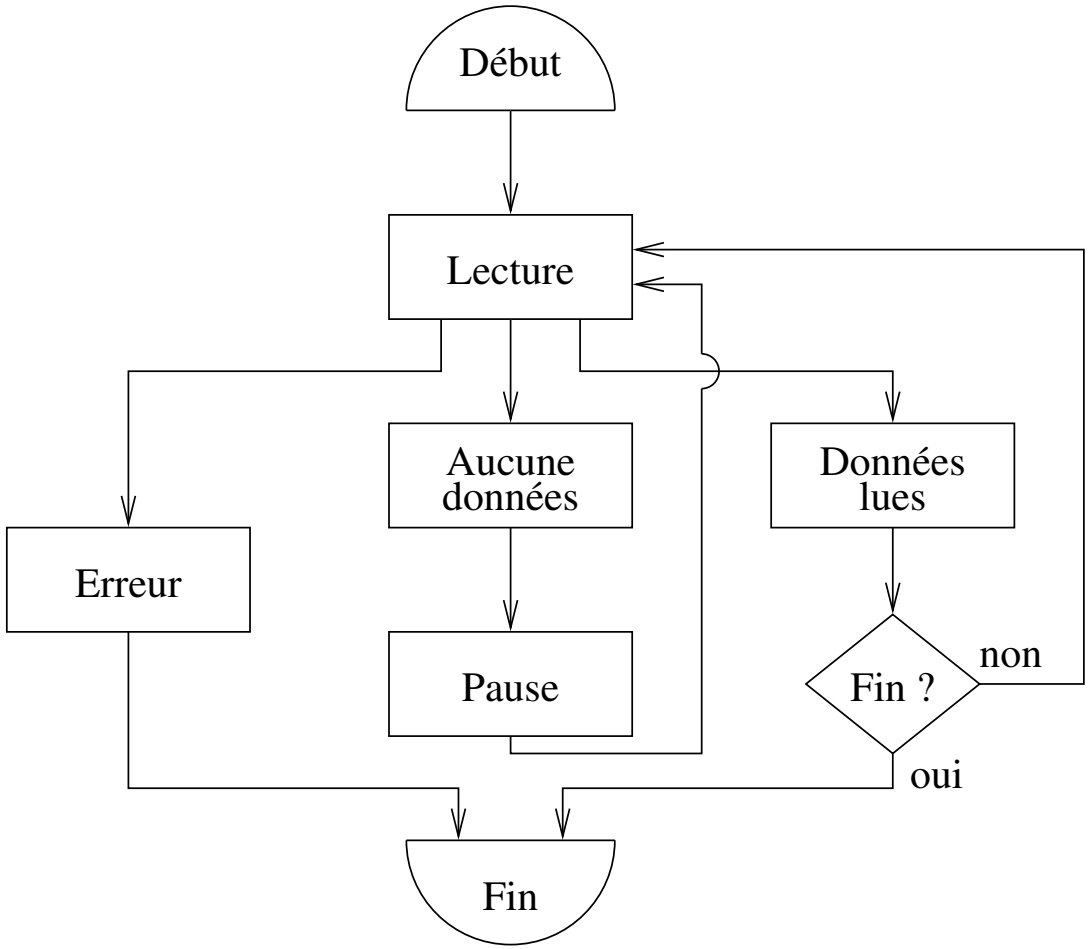
Paramètre(s)

- `fd` : le descripteur de fichier
- `cmd`, la commande parmi :
  - `F_GETFL` : récupère les attributs actuels
  - `F_SETFL` : fixe les nouveaux attributs spécifiés dans `arg`
- `arg` : les nouveaux attributs (si `cmd = F_SETFL`)

Valeurs retournées et erreurs générées

- Retourne 0 ou les paramètres ou -1 en cas d'échec
- Quelques erreurs possibles :
  - `EBADF` : `fd` n'est pas un descripteur valide

Lecture non bloquante avec attente active



Il est important de trouver une autre solution : la surveillance de descripteurs à l'aide de `select` (cf cours sur les sockets)

Exemple d'utilisation de `fcntl`

```
int attributs, res, tmp;

attributs = fcntl(tube[0], F_GETFL);
fcntl(tube[0], F_SETFL, attributs | O_NONBLOCK);

for(i = 0; i < 5; i++) {
    res = read(tube[0], &tmp, sizeof(int));
    while((res == -1) && (errno == EAGAIN)) {
        printf("Je n'ai rien reçu encore...\n");
        sleep(1);
        res = read(tube[0], &tmp, sizeof(int));
    }
    if((res == -1) && (errno != EAGAIN)) {
        perror("Erreur lors de la lecture ");
        exit(EXIT_FAILURE);
    }
    printf("Père : entier lu = %d\n", tmp);
}
```

Les tubes nommés

- Appelés aussi *fifo*
- Contrairement aux tubes anonymes :
  - Création d'une entrée dans le système de fichiers avec un nom
  - Mais toujours pas d'opération sur le périphérique sous-jacent !
- Accessibles depuis tout processus (qui possède les droits !) :
  $\hookrightarrow$ Pas de filiation nécessaire
- Pour afficher les tubes : `ls -l`
  $\hookrightarrow$ Attribut `p` pour les tubes nommés
  $\hookrightarrow$ Valeur après les droits = nombre de processus qui ont le tube ouvert

Création et manipulation d'un tube nommé

- Utilisation de la fonction :

```
int mkfifo(const char *nomfichier, mode_t mode)
```

- Pour communiquer dans le tube :
  - Ouverture comme pour un fichier régulier
  - Mais deux modes possibles : soit lecture, soit écriture
- Grosse différence = ouverture bloquante :
  - Tant que l'écrivain n'existe pas
  - Ou tant que le lecteur n'existe pas
- Lecture et écritures "classiques" via le descripteur
- La fermeture :
  - Avec `close`
  - Le fichier n'est pas détruit : utilisation de `unlink`
    ↪ Destruction non immédiate si un processus l'utilise actuellement

Fonction `mkfifo`

En-tête de la fonction (S3)

- `int mkfifo (const char *nomfichier, mode_t mode)`
- *Inclusions* : `sys/types.h` et `sys/stat.h`

Paramètre(s)

- `nomfichier` : le nom de fichier
- `mode` : droits d'accès (comme pour `open`)

Valeurs retournées et erreurs générées

- Retourne 0 ou -1 en cas d'échec
- Quelques erreurs possibles :
  - `EEXIST` : le fichier existe déjà
  - `ENOENT` : un répertoire dans le chemin n'existe pas

Fonction `unlink`

En-tête de la fonction (S2)

- `int unlink (const char *nomFichier)`
- *Inclusion* : `unistd.h`

Paramètre(s)

- `nomfichier` : le nom de fichier

Valeurs retournées et erreurs générées

- Retourne 0 ou -1 en cas d'échec
- Quelques erreurs possibles :
  - `EACCES` : problème d'accès
  - `EPERM` : `nomFichier` est un répertoire

Exemple d'utilisation des *fifos* (1/2) : le "serveur"

```c
#define NOM_TUBE "/tmp/montube"

int main(int argc, char *argv[]) {
  int fd, i;

  mkfifo(NOM_TUBE, S_IRUSR | S_IWUSR);

  fd = open(NOM_TUBE, O_WRONLY);

  for(i = 0; i < 5; i++)
    write(fd, &i, sizeof(int));

  close(fd);

  unlink(NOM_TUBE);

  return EXIT_SUCCESS;
}
```

Exemple d'utilisation des *fifos* (2/2) : le "client"

```c
#define NOM_TUBE "/tmp/montube"

int main(int argc, char *argv[]) {
  int fd, tab[5], i;

  fd = open(NOM_TUBE, O_RDONLY));

  read(fd, tab, sizeof(int) * 5);

  printf("Lu : ");
  for(i = 0; i < 5; i++)
    printf("%d ", i);
  printf("\n");

  close(fd);

  return EXIT_SUCCESS;
}
```

Communication bidirectionnelle

- Par rapport aux tubes anonymes :
  $\hookrightarrow$ Utilisation aussi de deux tubes
- Mais ouverture du tube bloquante :
  $\hookrightarrow$ Attention à l'ordre d'ouverture entre les deux processus !
- Une solution :
  - Ouverture du premier tube en lecture puis du second en écriture dans le premier processus
  - Ouverture du premier tube en écriture puis du second en lecture dans le deuxième processus

Communication bidirectionnelle (1/2) : le “serveur”

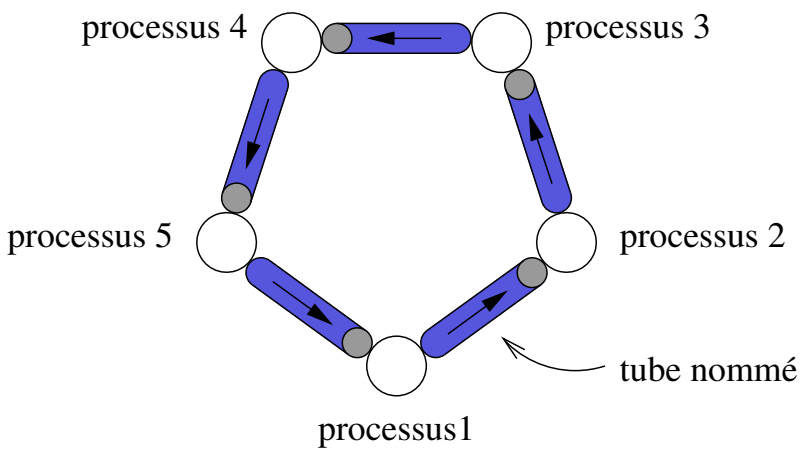
```
#define NOM_TUBE_1 "/tmp/montube1"
#define NOM_TUBE_2 "/tmp/montube2"

int main() {
  int i = 5, fd1, fd2;

  mkfifo(NOM_TUBE_1, S_IRUSR | S_IWUSR);
  mkfifo(NOM_TUBE_2, S_IRUSR | S_IWUSR);

  fd1 = open(NOM_TUBE_1, O_WRONLY);
  fd2 = open(NOM_TUBE_2, O_RDONLY);

  write(fd1, &i, sizeof(int));
  printf("Serveur : valeur envoyée = %d.\n", i);
  read(fd2, &i, sizeof(int));
  printf("Serveur : valeur reçue = %d.\n", i);

  close(fd1);close(fd2);
  unlink(NOM_TUBE_1);unlink(NOM_TUBE_2);

  return EXIT_SUCCESS;
}
```

Communication bidirectionnelle (2/2) : le "client"

```
#define NOM_TUBE_1 "/tmp/montube1"
#define NOM_TUBE_2 "/tmp/montube2"

int main() {
  int i, fd1, fd2;

  fd1 = open(NOM_TUBE_1, O_RDONLY);
  fd2 = open(NOM_TUBE_2, O_WRONLY);

  read(fd1, &i, sizeof(int));
  printf("Client : valeur reçue = %d.\n", i);
  i *= 2;
  write(fd2, &i, sizeof(int));
  printf("Client : valeur envoyée = %d.\n", i);

  close(fd1);
  close(fd2);

  return EXIT_SUCCESS;
}
```

Autre exemple : anneau unidirectionnel

- Idée : organiser les processus en anneau
  ↪ Chaque processus communique avec son suivant via un tube nommé
- Utilisation d'un programme unique
- Problèmes :
  - Attention aux interblocages !
  - Qui crée les tubes ?
  - Quels tubes choisir pour la lecture et l'écriture ?



Explications

- Chaque processus est identifié par un numéro unique :
  ↪ Numéro passé en paramètre à l'exécution du programme
- Le processus 0 crée les tubes nommés et les détruits :
  ↪ Attention : les autres processus doivent attendre la fin de la création
- Convention de nommage pour les tubes :
  ↪ *Exemple* : `tube_X` avec X$\in$ [0; 4]
- Le programme *n* :
  - Lit dans le tube `tube_n`
  - Écrit dans le tube `tube_m` avec m = X + 1 % 5
- Comment résoudre le problème de l'interblocage ?

Éviter les interblocages

- Description du problème :
  - Si chaque processus ouvre d'abord son tube en lecture :
    ↪ Tous seront bloqués en attente d'un écrivain
  - Idem si chacun ouvre d'abord son tube en écriture
- Première solution : rendre l'ouverture non bloquante
  ↪ Trop facile ! Et déconseillé...
- Deuxième solution : réfléchir !

Éviter les interblocages

- Description du problème :
  - Si chaque processus ouvre d'abord son tube en lecture :
    ↪ Tous seront bloqués en attente d'un écrivain
  - Idem si chacun ouvre d'abord son tube en écriture
- Première solution : rendre l'ouverture non bloquante
  ↪ Trop facile ! Et déconseillé...
- Deuxième solution :
  - Pour les processus de numéro impair :
    1. Ouverture de leur tube en écriture
    2. Ouverture de leur tube en lecture
  - Pour les processus de numéro pair :
    1. Ouverture de leur tube en lecture
    2. Ouverture de leur tube en écriture

Extrait de code (1/4)

```c
#define NOM_TUBE "/tmp/tube_"
#define NB_PROG  5

int main(int argc, char *argv[]) {
  int n, i, in, out;
  char nom[256];

  n = atoi(argv[1]);
  if(n == 0) {
    // Enregistrement de la procédure de nettoyage
    atexit(terminaison);
    /* Création des tubes */
    for(i = 0; i < NB_PROG; i++) {
      sprintf(nom, "%s%d", NOM_TUBE, i);
      mkfifo(nom, S_IRUSR | S_IWUSR);
    }
  }
  else
    sleep(1);
```

Extrait de code (2/4)

```
if(n % 2 == 0) {
  // Processus pairs : ouverture à droite puis à gauche
  sprintf(nom, "%s%d", NOM_TUBE, (n + 1) % NB_PROG);
  out = open(nom, O_WRONLY);
  sprintf(nom, "%s%d", NOM_TUBE, n);
  in = open(nom, O_RDONLY);
}
else {
  // Processus impaires : ouverture à gauche puis à droite
  sprintf(nom, "%s%d", NOM_TUBE, n);
  in = open(nom, O_RDONLY);
  sprintf(nom, "%s%d", NOM_TUBE, (n + 1) % NB_PROG);
  out = open(nom, O_WRONLY);
}
```

Extrait de code (3/4)

```
  if(n == 0) {
    // Le premier processus envoie un entier
    i = 0;
    write(out, &i, sizeof(int);
  }
  /* Attente d'un entier */
  read(in, &i, sizeof(int);

  if(n != 0) {
    // Écriture (sauf pour le premier processus)
    i++;
    write(out, &i, sizeof(int));
  }

  return EXIT_SUCCESS;
}
```

Extrait de code (4/4)

```
// Procédure appelée pour supprimer les tubes
void terminaison() {
  int i;
  char nom[256];

  for(i = 0; i < NB_PROG; i++) {
    sprintf(nom, "%s%d", NOM_TUBE, i);
    unlink(nom);
  }
}
```

Et pour l'exécution

- Première solution : ouvrir 5 terminaux différents
- Deuxième solution : utiliser un script qui exécute les 5 programmes
- Troisième solution : exploiter ses connaissances en système
  ↪ M. Rabat sera très content
- Idée :
  ↪ Créer cinq fils
  ↪ Exécuter le programme dans chaque fils

Exécution des programmes

```
int main() {
  int i;
  pid_t pid;
  char *arguments[3] = { "anneau", NULL, NULL };

  for(i = 0; i < 5; i++) {
    if((pid = fork()) == 0) {
      arguments[1] = (char*)malloc(sizeof(char) * 256);
      sprintf(arguments[1], "%d", i);
      execve("anneau", arguments, NULL);
      exit(EXIT_FAILURE);
    }
  }

  for(i = 0; i < 5; i++)
    wait(NULL);

  return EXIT_SUCCESS;
}
```

Les tubes anonymes et le shell

- Possibilité d'utiliser des tubes anonymes dans le shell :
  $\hookrightarrow$ Permet de transmettre un résultat d'une commande vers l'entrée d'une autre
- Utilisation du caractère : `|` (appelé justement `pipe`)

Exemple : que fait cette commande ?

```
ls | grep '.fig' | wc -l
```

Les tubes anonymes et le shell

- Possibilité d'utiliser des tubes anonymes dans le shell :
  ↪ Permet de transmettre un résultat d'une commande vers l'entrée d'une autre
- Utilisation du caractère : `|` (appelé justement `pipe`)

Exemple : que fait cette commande ?

```
ls | grep '.fig' | wc -l
```

Réponse

- `ls` : liste des fichiers
- `grep '.fig'` : extraction des lignes contenant `.fig`
- `wc -l` : compte le nombre de lignes

↪ Résultat : affiche le nombre de fichiers `.fig`

Exemple avec des programmes en C

Programme "prog1"

```
int main(int argc, char *argv[])
    {
  int i;
  for(i = 0; i < 5; i++) {
    printf("%d", i);
    fflush(stdout);
    sleep(1);
  }
  return EXIT_SUCCESS;
}
```

Programme "prog2"

```
int main(int argc, char *argv[])
    {
  int i;
  while(!feof(stdin)) {
    scanf("%d", &i);
    if(!feof(stdin))
      printf("Valeur␣%d\n", i);
  }
  return EXIT_SUCCESS;
}
```

Exécution

Dans le terminal : `./prog1 | ./prog2`

```
Valeur 0
Valeur 1
Valeur 2
Valeur 3
Valeur 4
```

Un peu plus loin avec la redirection

- Pour un programme donné, possible de rediriger les flux standards
- Utilisation des opérateurs `>`, `2>` et `<`
  ↪ *Exemple* : `./toto > sortie.txt 2> erreur.txt`
- Le contenu précedent des fichiers est perdu :
  ↪ Utilisation de `>>` ou `2>>` pour le conserver

Cas d'utilisation classiques

- Séparer la sortie standard de la sortie d'erreur :
  ↪ À condition d'avoir un programme "proprement" codé
- Faire du log pour du débogage
- Accélérer l'exécution pour un programme très bavard

Et les *fifos* ?

- Pour créer une *fifo*, utilisation de la commande `mkfifo` :
  - `mkfifo toto` : création du tube `toto`
  - `mkfifo toto -m r=rw` : spécifier les droits
- Pour la supprimer :
  - `rm`
  - `unlink` (comme pour l'appel système en C)

Les primitives `dup` et `dup2`

- Utilisées pour dupliquer un descripteur
- Copie exacte du descripteur spécifié :
  - Création d'un nouveau descripteur
  - Propriétés recopiées (position, propriétés d'ouverture...)
  - Utilisation du plus petit numéro disponible
- Possible de spécifier un numéro de descripteur :
  - Utilisation de `dup2`
  - Si le descripteur existe déjà, il est fermé avant

Fonction dup

En-têtes des fonctions (S2)

- `int dup(int oldfd)`
- `int dup2(int oldfd, int newfd)`
- *Inclusion* : `unistd.h`

Paramètre(s)

- `oldfd` : descripteur à copier
- `newfd` : ancien descripteur (fermé si nécessaire)

Valeurs retournées et erreurs générées

- Retourne le nouveau descripteur ou -1 en cas d'erreur
- Quelques erreurs possibles :
  - `EBADF` : descripteur invalide
  - `EMFILE` : trop de descripteurs ouverts par le processus

Exemple d'utilisation (1/2)

- Rediriger la sortie standard d'un processus vers l'entrée standard d'un autre processus
- Par exemple :
  - Programme exécuté dans le processus fils qui génère un flot de données vers la sortie standard
  - Programme exécuté dans le père qui récupère le flot de données dans son entrée standard
- Idée :
  - Utiliser un tube anonyme
  - Relier la sortie du processus fils à l'entrée du processus père...
  - ...en utilisant le tube

Exemple d'utilisation (2/2) : algorithme général

1. Création d'un tube anonyme (avec `pipe`)
2. Création d'un processus fils (avec `fork`)
3. Dans le fils :
   1. Recopie de la sortie du tube vers la sortie standard
   2. Exécution du programme "générateur"
4. Dans le père :
   1. Recopie de l'entrée du tube vers l'entrée standard
   2. Exécution du programme "lecteur"

Exemple d'utilisation avec dup (1/2)

```
int main() {
  int tube[2];
  char *arguments1[3] = { "/bin/ls", "-l", NULL };
  char *arguments2[3] = { "/usr/bin/wc", "-l", NULL };
  pid_t pid;

  pipe(tube);
  if((pid = fork()) == 0) {
    /* Dans le fils */
    close(tube[TUBE_LECTURE]);

    close(STDOUT_FILENO);
    dup(tube[TUBE_ECRITURE]);

    close(tube[TUBE_ECRITURE]);

    execve(arguments1[0], arguments1, NULL);
    exit(EXIT_FAILURE);
  }
  ...
```

Exemple d'utilisation avec dup (2/2)

```
    ...
    /* Dans le père */
    close(tube[TUBE_ECRITURE]);

    close(STDIN_FILENO);
    dup(tube[TUBE_LECTURE]);

    close(tube[TUBE_LECTURE]);

    execve(arguments2[0], arguments2, NULL);

    return EXIT_FAILURE;
}
```

Exemple d'utilisation avec dup2 (1/2)

```
int main() {
  int tube[2];
  char *arguments1[3] = { "/bin/ls", "-l", NULL };
  char *arguments2[3] = { "/usr/bin/wc", "-l", NULL };
  pid_t pid;

  pipe(tube);
  if((pid = fork()) == 0) {
    /* Dans le fils */
    close(tube[TUBE_LECTURE]);

    dup2(tube[TUBE_ECRITURE], STDOUT_FILENO);

    close(tube[TUBE_ECRITURE]);

    execve(arguments1[0], arguments1, NULL);
    exit(EXIT_FAILURE);
  }
  ...
```

Exemple d'utilisation avec dup2 (2/2)

```
    ...
    /* Dans le père */
    close(tube[TUBE_ECRITURE]);

    dup2(tube[TUBE_LECTURE], STDIN_FILENO);

    close(tube[TUBE_LECTURE]);

    execve(arguments2[0], arguments2, NULL);

    return EXIT_FAILURE;
}
```