La communication par tubes

Luigi Santocanale

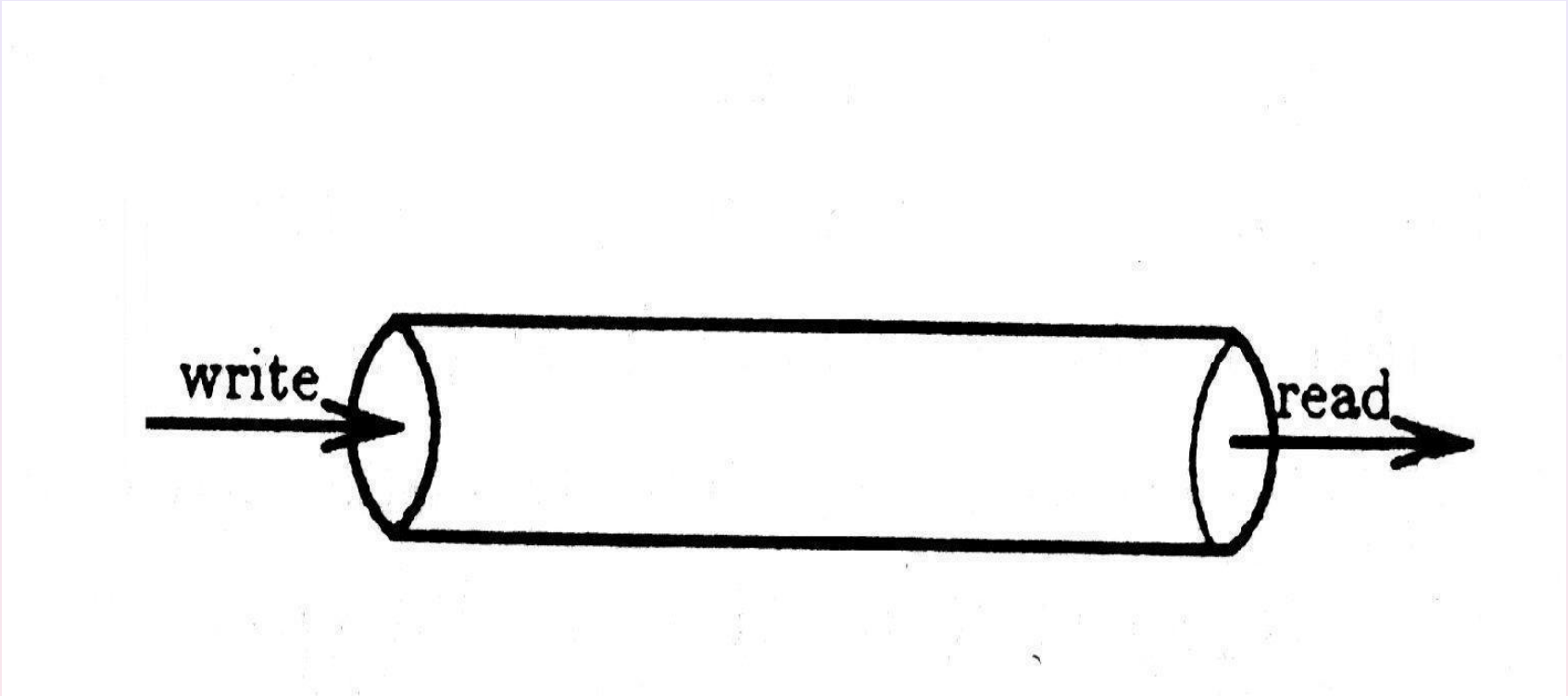
Laboratoire d'Informatique Fondamentale,
Centre de Mathématiques et Informatique,
39, rue Joliot-Curie - F-13453 Marseille

8 novembre 2005

Plan

Plan

Tube (pipe)



Propriétés de la communication par tubes

- Mécanisme de communication entre les processus.
- Communication unidirectionnelle.
- FIFO, « first in, first out » :
  premier entré, premier sorti.
- Communication mode STREAM, flot.
- Mécanisme de synchronisation inclus :
  - # écrivains : si 0, alors EOF.
  - # lecteurs : si 0, interdiction d'écrire.

Propriétés de la communication par tubes

- Mécanisme de communication entre les processus.
- Communication unidirectionnelle.
- FIFO, « first in, first out » :
  premier entré, premier sorti.
- Communication mode STREAM, flot.
- Mécanisme de synchronisation inclus :
  - # écrivains : si 0, alors EOF.
  - # lecteurs : si 0, interdiction d'écrire.

Propriétés de la communication par tubes

- Mécanisme de communication entre les processus.
- Communication unidirectionnelle.
- FIFO, « first in, first out » :
  premier entré, premier sorti.
- Communication mode STREAM, flot.
- Mécanisme de synchronisation inclus :
  - # écrivains : si 0, alors EOF.
  - # lecteurs : si 0, interdiction d'écrire.

Propriétés de la communication par tubes

- Mécanisme de communication entre les processus.
- Communication unidirectionnelle.
- FIFO, « first in, first out » :
  premier entré, premier sorti.
- Communication mode STREAM, flot.

- Mécanisme de synchronisation inclus :
  - # écrivains : si 0, alors EOF.
  - # lecteurs : si 0, interdiction d'écrire.

Propriétés de la communication par tubes

- Mécanisme de communication entre les processus.
- Communication unidirectionnelle.
- FIFO, « first in, first out » :
  premier entré, premier sorti.
- Communication mode STREAM, flot.

- Mécanisme de synchronisation inclus :
  - # écrivains : si 0, alors EOF.
  - # lecteurs : si 0, interdiction d'écrire.

Propriétés de la communication par tubes

- Mécanisme de communication entre les processus.
- Communication unidirectionnelle.
- FIFO, « first in, first out » :
  premier entré, premier sorti.
- Communication mode STREAM, flot.

- Mécanisme de synchronisation inclus :
  - # écrivains : si 0, alors EOF.
  - # lecteurs : si 0, interdiction d'écrire.

Propriétés de la communication par tubes

- Mécanisme de communication entre les processus.
- Communication unidirectionnelle.
- FIFO, « first in, first out » :
  premier entré, premier sorti.
- Communication mode STREAM, flot.

- Mécanisme de synchronisation inclus :
  - # écrivains : si 0, alors EOF.
  - # lecteurs : si 0, interdiction d'écrire.

Plan

Niveau implémentation

- Mécanisme appartenant au système de gestion des fichiers: fichiers type tube (pipe).
- Dans le *table des fichiers ouverts* :
  une seule entrée en lecture, une seule entrée en écriture.
- Lecture destructrice :
  position courante ou « offset » n'existe pas.
- Capacité finie, le tube peut être plein.
- # lecteurs = # processus ayant ouvert le tube en lecture.
- # écrivains = # processus ayant ouvert le tube en écriture.

Niveau implémentation

- Mécanisme appartenant au système de gestion des fichiers: fichiers type tube (pipe).
- Dans le *table des fichiers ouverts* :
  une seule entrée en lecture, une seule entrée en écriture.
- Lecture destructrice :
  position courante ou « offset » n'existe pas.
- Capacité finie, le tube peut être plein.
- # lecteurs = # processus ayant ouvert le tube en lecture.
- # écrivains = # processus ayant ouvert le tube en écriture.

Niveau implémentation

- Mécanisme appartenant au système de gestion des fichiers: fichiers type tube (pipe).
- Dans le *table des fichiers ouverts* :
  une seule entrée en lecture, une seule entrée en écriture.
- Lecture destructrice :
  position courante ou « offset » n'existe pas.
- Capacité finie, le tube peut être plein.
- # lecteurs = # processus ayant ouvert le tube en lecture.
- # écrivains = # processus ayant ouvert le tube en écriture.

Niveau implémentation

- Mécanisme appartenant au système de gestion des fichiers: fichiers type tube (pipe).
- Dans le *table des fichiers ouverts* :
  une seule entrée en lecture, une seule entrée en écriture.
- Lecture destructrice :
  position courante ou « offset » n'existe pas.
- Capacité finie, le tube peut être plein.
- # lecteurs = # processus ayant ouvert le tube en lecture.
- # écrivains = # processus ayant ouvert le tube en écriture.

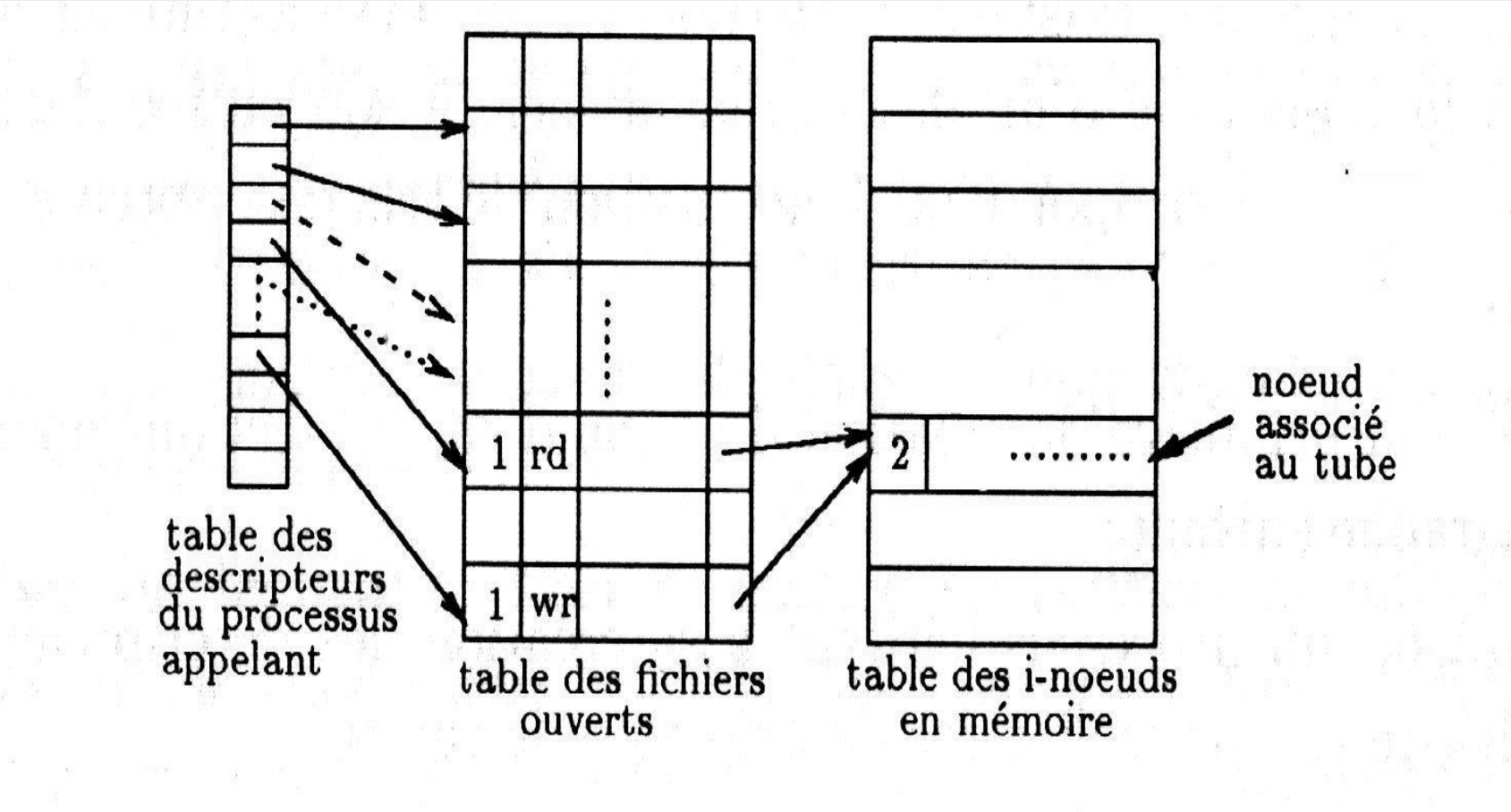
Niveau implémentation

- Mécanisme appartenant au système de gestion des fichiers: fichiers type tube (pipe).
- Dans le *table des fichiers ouverts* :
  une seule entrée en lecture, une seule entrée en écriture.
- Lecture destructrice :
  position courante ou « offset » n'existe pas.
- Capacité finie, le tube peut être plein.
- # lecteurs = # processus ayant ouvert le tube en lecture.
- # écrivains = # processus ayant ouvert le tube en écriture.

Niveau implémentation

- Mécanisme appartenant au système de gestion des fichiers: fichiers type tube (pipe).
- Dans le *table des fichiers ouverts* :
  une seule entrée en lecture, une seule entrée en écriture.
- Lecture destructrice :
  position courante ou « offset » n'existe pas.
- Capacité finie, le tube peut être plein.
- # lecteurs = # processus ayant ouvert le tube en lecture.
- # écrivains = # processus ayant ouvert le tube en écriture.

Organisation dans le noyau (Rifflet 1999)



Implémentation des tubes

- Un tube est un i-noeud sur une périphérique dédiée aux tubes.
- Seulement les bloques adressés de façon directe sont utilisé.
- Les tubes sont implémentés en tant que *files circulaires* : pointeur d'écriture, pointeur de lecture.
- Les pointeurs sont mis à jour après chaque opération.
- Taille du tube :

```
ptr_lecture - ptr_ecriture % taille_totale
```

- Espace libre dans le tube :

```
ptr_ecriture - ptr_lecture % taille_totale
```

Implémentation des tubes

- Un tube est un i-noeud sur une périphérique dédiée aux tubes.
- Seulement les bloques adressés de façon directe sont utilisé.
- Les tubes sont implémentés en tant que *files circulaires* : pointeur d'écriture, pointeur de lecture.
- Les pointeurs sont mis à jour après chaque opération.
- Taille du tube :

```
ptr_lecture - ptr_ecriture % taille_totale
```

- Espace libre dans le tube :

```
ptr_ecriture - ptr_lecture % taille_totale
```

Implémentation des tubes

- Un tube est un i-noeud sur une périphérique dédiée aux tubes.
- Seulement les bloques adressés de façon directe sont utilisé.
- Les tubes sont implémentés en tant que *files circulaires* : pointeur d'écriture, pointeur de lecture.
- Les pointeurs sont mis à jour après chaque opération.
- Taille du tube :

```
ptr_lecture - ptr_ecriture % taille_totale
```

- Espace libre dans le tube :

```
ptr_ecriture - ptr_lecture % taille_totale
```

Implémentation des tubes

- Un tube est un i-noeud sur une périphérique dédiée aux tubes.
- Seulement les bloques adressés de façon directe sont utilisé.
- Les tubes sont implémentés en tant que *files circulaires* : pointeur d'écriture, pointeur de lecture.
- Les pointeurs sont mis à jour après chaque opération.
- Taille du tube :

```
ptr_lecture - ptr_ecriture % taillletotale
```

- Espace libre dans le tube :

```
ptr_ecriture - ptr_lecture % taillletotale
```

Implémentation des tubes

- Un tube est un i-noeud sur une périphérique dédiée aux tubes.
- Seulement les bloques adressés de façon directe sont utilisé.
- Les tubes sont implémentés en tant que *files circulaires* : pointeur d'écriture, pointeur de lecture.
- Les pointeurs sont mis à jour après chaque opération.
- Taille du tube :

```
ptr_lecture - ptr_ecriture % tailletotale
```

- Espace libre dans le tube :

```
ptr_ecriture - ptr_lecture % tailletotale
```

Implémentation des tubes

- Un tube est un i-noeud sur une périphérique dédiée aux tubes.
- Seulement les bloques adressés de façon directe sont utilisé.
- Les tubes sont implémentés en tant que *files circulaires* : pointeur d'écriture, pointeur de lecture.
- Les pointeurs sont mis à jour après chaque opération.
- Taille du tube :

```
ptr_lecture - ptr_ecriture % tailletotale
```

- Espace libre dans le tube :

```
ptr_ecriture - ptr_lecture % tailletotale
```

Implémentation des tubes

- Un tube est un i-noeud sur une périphérique dédiée aux tubes.
- Seulement les bloques adressés de façon directe sont utilisé.
- Les tubes sont implémentés en tant que *files circulaires* : pointeur d'écriture, pointeur de lecture.
- Les pointeurs sont mis à jour après chaque opération.
- Taille du tube :

```
ptr_lecture - ptr_ecriture % tailletotale
```

- Espace libre dans le tube :

```
ptr_ecriture - ptr_lecture % tailletotale
```

Implémentation des tubes

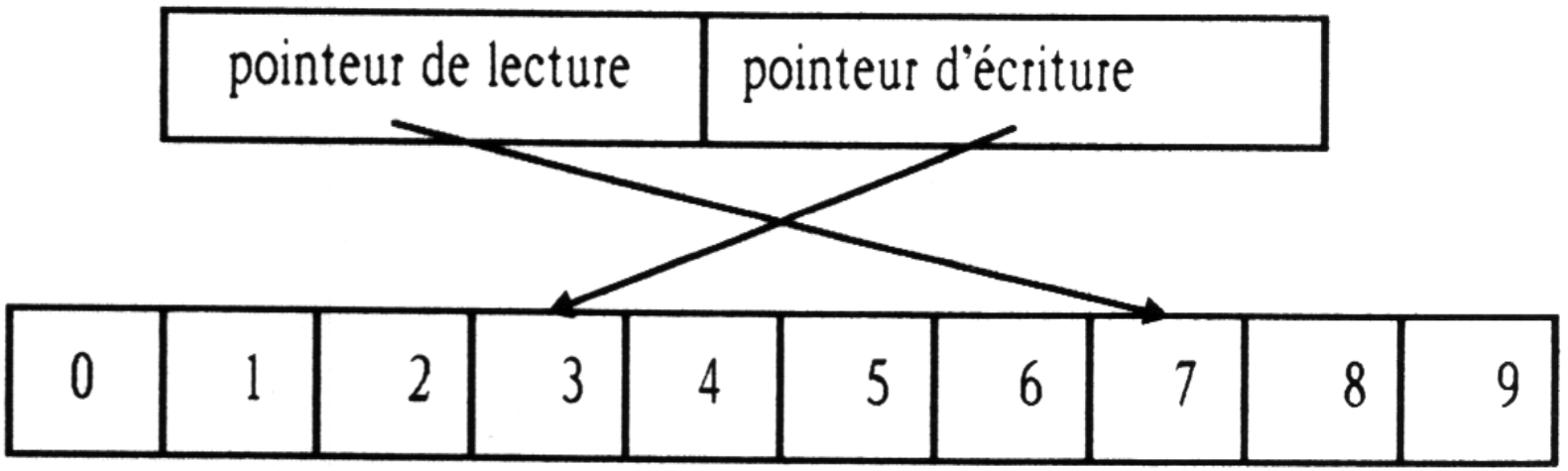
- Un tube est un i-noeud sur une périphérique dédiée aux tubes.
- Seulement les bloques adressés de façon directe sont utilisé.
- Les tubes sont implémentés en tant que *files circulaires* : pointeur d'écriture, pointeur de lecture.
- Les pointeurs sont mis à jour après chaque opération.
- Taille du tube :

```
ptr_lecture - ptr_ecriture % tailletotale
```

- Espace libre dans le tube :

```
ptr_ecriture - ptr_lecture % tailletotale
```

Implémentation des tubes par files circulaires (Bach 1989)



Programme : Du fichier pipe-fs-i.h (linux)

```
1  struct pipe_inode_info {
2     wait_queue_head_t wait;
3     char *base;
4     unsigned int len;
5     unsigned int start;
6     unsigned int readers;
7     unsigned int writers;
8     unsigned int waiting_writers;
9     unsigned int r_counter;
10    unsigned int w_counter;
11
12    ...
13
14 };
```

Types de tubes

- Tubes ordinaires ou anonymes : il lui correspond
  - un i-noeud.

  Communication entre processus de la même famille.

- Tubes nommées : il lui correspond
  - un i-noeud,
  - un nom, référence, chemin . . .

  Communication entre processus génériques.

Types de tubes

- Tubes ordinaires ou anonymes : il lui correspond
  - un i-noeud.

  Communication entre processus de la même famille.

- Tubes nommées : il lui correspond
  - un i-noeud,
  - un nom, référence, chemin ...

  Communication entre processus génériques.

Plan

Algorithme pipe (Bach 1986)

```
algorithme pipe
entrée: néant
sortie: descripteur de fichier en lecture
        descripteur de fichier en écriture
{
    s'affecter un nouvel i-noeud du périphérique tube (algorithme ialloc);
    attribuer un élément de la table des fichiers pour la lecture,
                                                un autre pour l'écriture;
    initialiser les éléments de la table des fichiers pour qu'ils
                                                pointent le nouvel i-noeud;
    attribuer un descripteur de fichier pour la lecture,
                un autre pour l'écriture, les initialiser pour qu'ils
                pointent leur élément respectif dans la table des fichiers;
    initialiser le compte référence de l'i-noeud à 2;
    initialiser le compte de lecteurs et d'écrivains de l'i-noeud à 1;
}
```

pipe

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

p[2] : adresse d'un tableau de descripteurs de dimension 2 à remplir :
p[0] : le descripteur de lecture,
p[1] : le descripteur d'écriture.

Retourne : 0/-1

pipe

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

p[2] : adresse d'un tableau de descripteurs de dimension 2 à remplir :
`p[0]` : le descripteur de lecture,
`p[1]` : le descripteur d'écriture.

Retourne : 0/-1

pipe

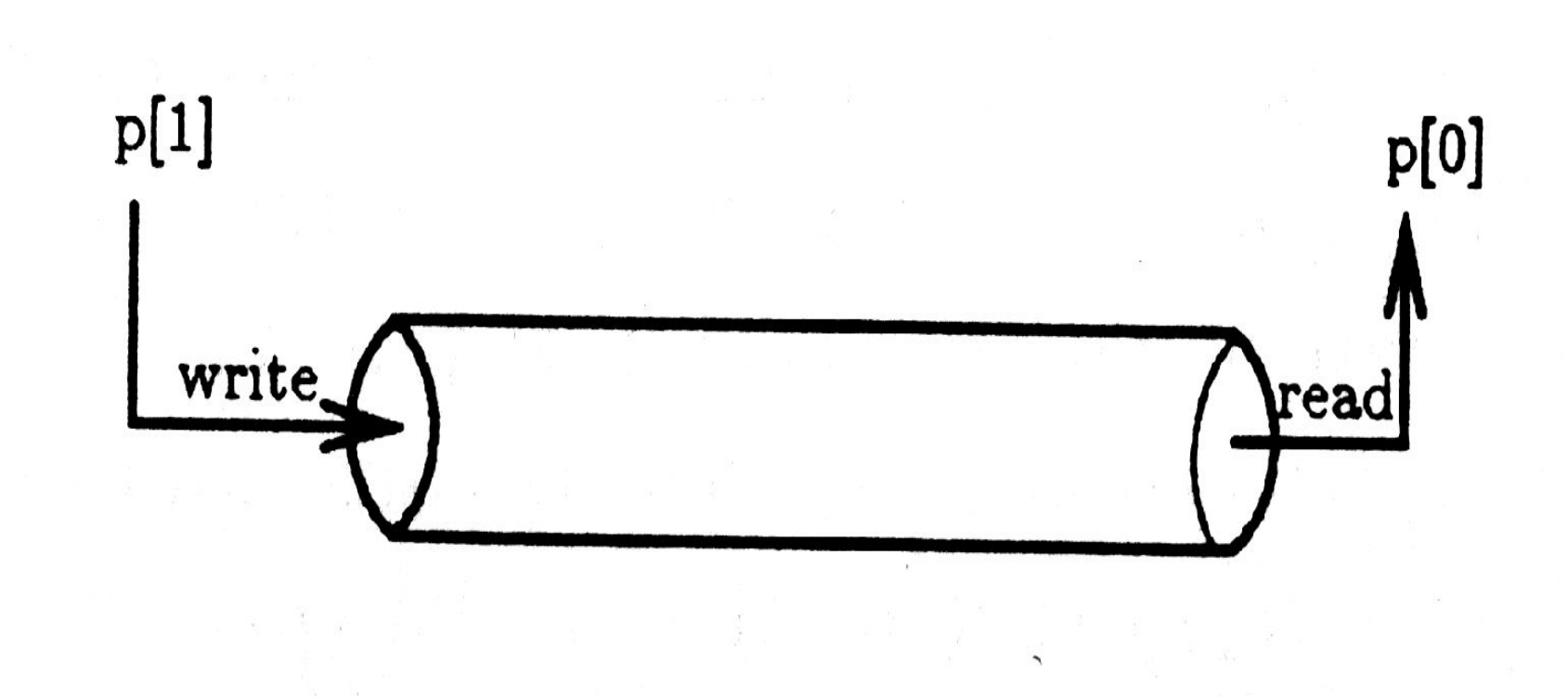
```
#include <unistd.h>
int pipe(int p[2]);
```

p[2] : adresse d'un tableau de descripteurs de dimension 2 à remplir :
`p[0]` : le descripteur de lecture,
`p[1]` : le descripteur d'écriture.

Retourne : 0/-1

Tube

(Rifflet 1999)



Programme : expipe.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4
5  void fils(int d_ecriture); void pere(int d_lecture);
6
7  int main(void)
8  {
9    int p[2];
10
11   if( pipe(p) == -1 ) exit(EXIT_FAILURE);
12
13   switch( fork() )
14     {
15     case -1 :
16       exit(EXIT_FAILURE);
17     case 0 : /* Le fils écrive dans le tube */
18       close(p[0]); /* Le fils ne lit pas du tube */
19       fils(p[1]);
20     default : /* Le père lit du tube */
21       close(p[1]); /* Le père n'écrit pas dans ce tube */
22       pere(p[0]);
23     }
24   exit(EXIT_SUCCESS);
25 }
```

Programme : expipe.c (II)

```
27  void fils(int d_ecriture)
28  {
29    char message[]="abcde";
30    int no_ecrit;
31
32    no_ecrit =
33      write(d_ecriture,message,sizeof(message));
34
35    printf("Le fils dit :\n"
36           "\tJ'ai écrit %d octets "
37           "dans le fichier %d.\n",
38           no_ecrit,d_ecriture);
39    printf("\tCe que j'ai écrit : \"%s\"\n",
40           message);
41
42    exit(EXIT_SUCCESS);
43  }
```

Programme : expipe.c (III)

```
45  void pere(int d_lecture)
46  {
47    char tampon[100];
48    int no_lu;
49
50    no_lu = read(d_lecture,tampon,sizeof(tampon));
51
52    printf("Le père dit :\n"
53           "\tJ'ai lu %d octets du fichier %d.\n",
54           no_lu,d_lecture);
55    printf("\tCe que j'ai lu : \"%s\"\n",tampon);
56
57    exit(EXIT_SUCCESS);
58  }
```

Session : expipe

```
$ gcc -Wall -pedantic expipe.c
$ a.out
Le fils dit :
        J'ai écrit 6 octets dans le fichier 4.
        Ce que j'ai écrit : "abcde"
Le père dit :
        J'ai lu 6 octets du fichier 3.
        Ce que j'ai lu : "abcde"
```

Considérations

- Le tube est anonyme : tables compteur des liens nuls.
- Pour connaître ce fichier il faut en posséder un descripteur
  - en créant le tube,
  - par héritage.
- Utilisé par les descendants du processus créateur.
- Si on ferme le(s descripteur du) tube
  on ne peut pas plus l'utiliser.

Considérations

- Le tube est anonyme : tables compteur des liens nuls.
- Pour connaître ce fichier il faut en posséder un descripteur
  - en créant le tube,
  - par héritage.
- Utilisé par les descendants du processus créateur.
- Si on ferme le(s descripteur du) tube
  on ne peut pas plus l'utiliser.

Considérations

- Le tube est anonyme : tables compteur des liens nuls.
- Pour connaître ce fichier il faut en posséder un descripteur
  - en créant le tube,
  - par héritage.
- Utilisé par les descendants du processus créateur.
- Si on ferme le(s descripteur du) tube
  on ne peut pas plus l'utiliser.

Considérations

- Le tube est anonyme : tables compteur des liens nuls.
- Pour connaître ce fichier il faut en posséder un descripteur
  - en créant le tube,
  - par héritage.
- Utilisé par les descendants du processus créateur.
- Si on ferme le(s descripteur du) tube
  on ne peut pas plus l'utiliser.

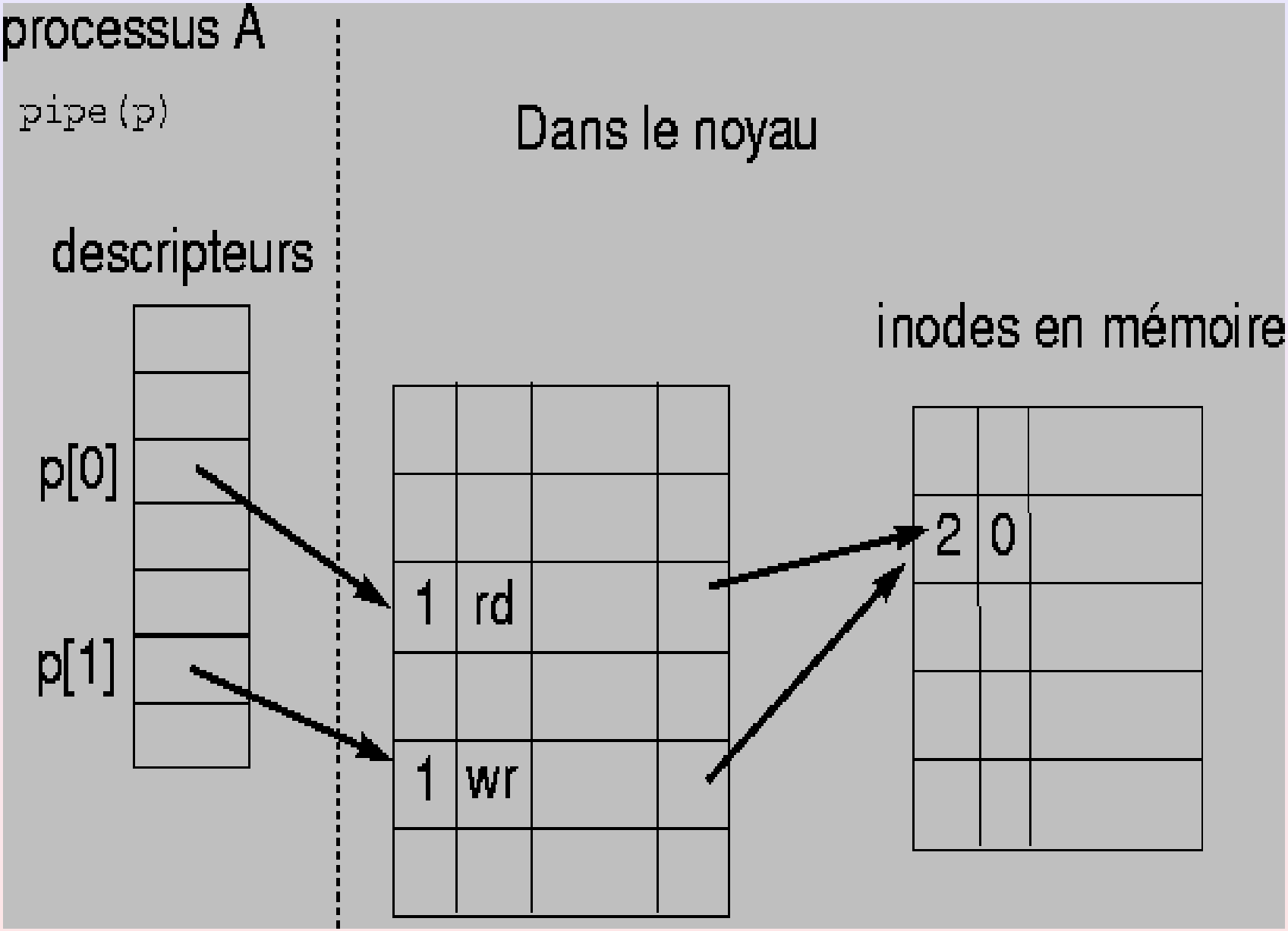
Considérations

- Le tube est anonyme : tables compteur des liens nuls.
- Pour connaître ce fichier il faut en posséder un descripteur
  - en créant le tube,
  - par héritage.
- Utilisé par les descendants du processus créateur.
- Si on ferme le(s descripteur du) tube
  on ne peut pas plus l'utiliser.

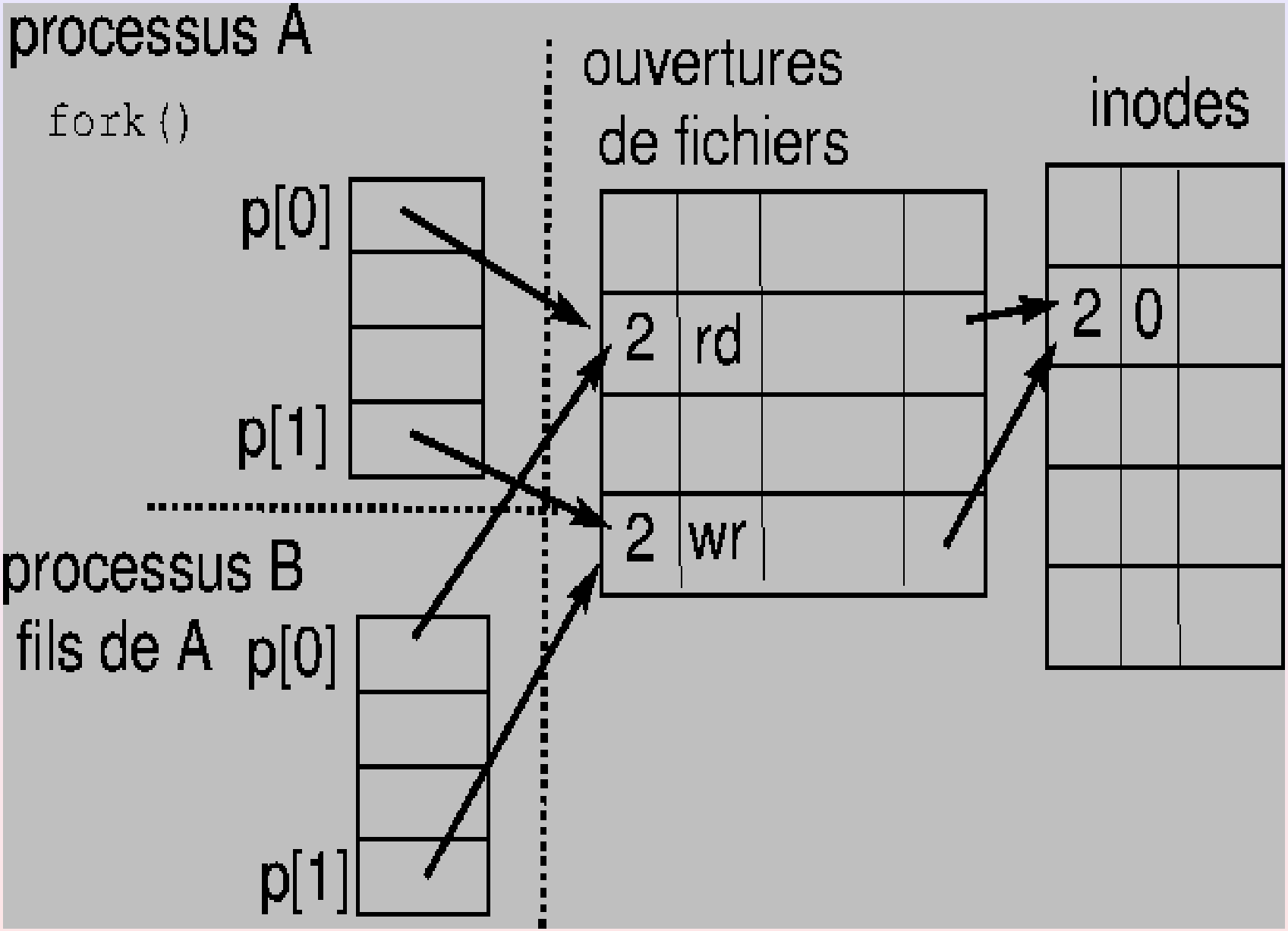
Considérations

- Le tube est anonyme : tables compteur des liens nuls.
- Pour connaître ce fichier il faut en posséder un descripteur
  - en créant le tube,
  - par héritage.
- Utilisé par les descendants du processus créateur.
- Si on ferme le(s descripteur du) tube
  on ne peut pas plus l'utiliser.

Création d'un tube anonyme (Revuz 1998)



Héritage d'un tube anonyme (Revuz 1998)



Plan

read

```
#include <unistd.h>
size_t read(int desc, void * buf, size_t taille);
```

desc : descripteur fichier ouvert en lecture,
buf : un pointer à tampon en mémoire,
taille : nombre maximale d'octets à lire.

Retourne : nombre d'octets lus.

Remarques : on a déjà vu cette primitive en rapport aux fichiers réguliers.

read

```
#include <unistd.h>
size_t read(int desc, void * buf, size_t taille);
```

desc : descripteur fichier ouvert en lecture,
buf : un pointer à tampon en mémoire,
taille : nombre maximale d'octets à lire.

Retourne : nombre d'octets lus.

Remarques : on a déjà vu cette primitive en rapport aux fichiers réguliers.

read

```
#include <unistd.h>
size_t read(int desc, void * buf, size_t taille);
```

desc : descripteur fichier ouvert en lecture,
buf : un pointer à tampon en mémoire,
taille : nombre maximale d'octets à lire.

Retourne : nombre d'octets lus.

Remarques : on a déjà vu cette primitive en rapport aux fichiers réguliers.

read

```
#include <unistd.h>
size_t read(int desc, void * buf, size_t taille);
```

desc : descripteur fichier ouvert en lecture,
buf : un pointer à tampon en mémoire,
taille : nombre maximale d'octets à lire.

Retourne : nombre d'octets lus.

Remarques : on a déjà vu cette primitive en rapport aux fichiers réguliers.

read

```
#include <unistd.h>
size_t read(int desc, void * buf, size_t taille);
```

desc : descripteur fichier ouvert en lecture,
buf : un pointer à tampon en mémoire,
taille : nombre maximale d'octets à lire.

Retourne : nombre d'octets lus.

Remarques : on a déjà vu cette primitive en rapport aux fichiers réguliers.

read

```
#include <unistd.h>
size_t read(int desc, void * buf, size_t taille);
```

desc : descripteur fichier ouvert en lecture,
buf : un pointer à tampon en mémoire,
taille : nombre maximale d'octets à lire.

Retourne : nombre d'octets lus.

Remarques : on a déjà vu cette primitive en rapport aux fichiers réguliers.

Algorithme read

```
nb_lu = read(p[0], buf, TAILLE_BUF);
```

tube n'est pas vide :

on place nb_lu = min(taille_tube,TAILLE_BUF) dans buf

tube est vide :

ecrivains == 0 :

nb_lu = 0;

ecrivains != 0 :

lecture bloquante (par défaut) :

processus en sommeil

lecture non bloquante : -1

Attention à l'interbloquage !!!

Algorithme `read`

```
nb_lu = read(p[0], buf, TAILLE_BUF);
```

tube n'est pas vide :

on place `nb_lu = min(taille_tube,TAILLE_BUF)` dans `buf`

tube est vide :

ecrivains == 0 :

`nb_lu = 0;`

ecrivains != 0 :

lecture bloquante (par défaut) :

processus en sommeil

lecture non bloquante : -1

Attention à l'interbloquage !!!

Algorithme `read`

```
nb_lu = read(p[0], buf, TAILLE_BUF);
```

tube n'est pas vide :

on place `nb_lu = min(taille_tube,TAILLE_BUF)` dans `buf`

tube est vide :

ecrivains = 0 :

`nb_lu = 0`,

ecrivains ≠ 0 :

lecture bloquante (par défaut) :

processus en sommeil

lecture non bloquante : -1

Attention à l'interbloquage !!!

Algorithme `read`

```
nb_lu = read(p[0], buf, TAILLE_BUF);
```

tube n'est pas vide :

on place `nb_lu = min(taille_tube,TAILLE_BUF)` dans `buf`

tube est vide :

ecrivains = 0 :

`nb_lu = 0`,

ecrivains ≠ 0 :

lecture bloquante (par défaut) :

processus en sommeil

lecture non bloquante : -1

Attention à l'interbloquage !!!

Algorithme `read`

```
nb_lu = read(p[0], buf, TAILLE_BUF);
```

tube n'est pas vide :

on place `nb_lu = min(taille_tube,TAILLE_BUF)` dans `buf`

tube est vide :

ecrivains = 0 :

`nb_lu = 0`,

ecrivains ≠ 0 :

lecture bloquante (par défaut) :

processus en sommeil

lecture non bloquante : -1

Attention à l'interbloquage !!!

Algorithme `read`

```
nb_lu = read(p[0], buf, TAILLE_BUF);
```

tube n'est pas vide :

on place `nb_lu = min(taille_tube,TAILLE_BUF)` dans `buf`

tube est vide :

ecrivains = 0 :

`nb_lu = 0`,

ecrivains ≠ 0 :

lecture bloquante (par défaut) :

processus en sommeil

lecture non bloquante : -1

Attention à l'interbloquage !!!

Algorithme `read`

```
nb_lu = read(p[0], buf, TAILLE_BUF);
```

tube n'est pas vide :

on place `nb_lu = min(taille_tube,TAILLE_BUF)` dans `buf`

tube est vide :

ecrivains = 0 :

`nb_lu = 0`,

ecrivains ≠ 0 :

lecture bloquante (par défaut) :

processus en sommeil

lecture non bloquante : -1

Attention à l'interbloquage !!!

Programme : interbloquage.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <unistd.h>
 4
 5 #define ENTREETUBE 1 /* == sortie du processus */
 6 #define SORTIETUBE 0 /* == entrée du processus */
 7
 8 void codecommun(int d_lecture,int d_ecriture)
 9 {
10   char buf[100];
11   int no_lu,no_ecrit;
12
13   printf("[%d] Prêt pour lire.\n",getpid());
14   no_lu = read(d_lecture,buf,sizeof(buf));
15
16   printf("[%d] Prêt pour écrire.\n",getpid());
17   no_ecrit = write(d_ecriture,buf,sizeof(buf));
18
19   printf("[%d] No. lus : %d, no. écrits : %d.\n",
20          getpid(),no_lu,no_ecrit);
21 }
```

Programme : interbloquage.c (II)

```
23 int main(void)
24 {
25   int pf[2]; /* Tube père -> fils */
26   int fp[2]; /* Tube fils -> père */
27
28   if(pipe(pf) == -1 || pipe(fp) == -1)
29     exit(EXIT_FAILURE);
30
31   switch( fork() )
32     {
33     case -1 :
34       exit(EXIT_FAILURE);
35     case 0 : /* Le fils */
36       close(pf[ENTREETUBE]);
37       close(fp[SORTIETUBE]);
38       codecommun(pf[SORTIETUBE],fp[ENTREETUBE]);
39       break;
40     default : /* Le père */
41       close(pf[SORTIETUBE]);
42       close(fp[ENTREETUBE]);
43       codecommun(fp[SORTIETUBE],pf[ENTREETUBE]);
44     }
45   exit(EXIT_SUCCESS);
46 }
```

Session : interbloquage

```
$ gcc -Wall -pedantic interbloquage.c
$ a.out & ps -o pid,cmd,s
[8791] Prêt pour lire.
[8790] Prêt pour lire.
[3] 8790
  PID CMD                      S
 4707 /bin/bash                S
 8790 a.out                    S
 8791 a.out                    S
 8792 ps -o pid,cmd,s          R
$
```

Plan

write

```
#include <unistd.h>
size_t write(int desc, void * buf, size_t noocts);
```

desc : descripteur fichier ouvert en écriture
buf : un pointer à un tampon
noocts : le nombre d'octets qu'on veut écrire

Retourne : nombre caractères écrits, -1 si erreur

write

```
#include <unistd.h>
size_t write(int desc, void * buf, size_t noocts);
```

desc : descripteur fichier ouvert en écriture
buf : un pointer à un tampon
noocts : le nombre d'octets qu'on veut écrire

Retourne : nombre caractères écrits, -1 si erreur

write

```
#include <unistd.h>
size_t write(int desc, void * buf, size_t noocts);
```

desc : descripteur fichier ouvert en écriture
buf : un pointer à un tampon
noocts : le nombre d'octets qu'on veut écrire

Retourne : nombre caractères écrits, -1 si erreur

write

```
#include <unistd.h>
size_t write(int desc, void * buf, size_t noocts);
```

desc : descripteur fichier ouvert en écriture
buf : un pointer à un tampon
noocts : le nombre d'octets qu'on veut écrire

Retourne : nombre caractères écrits, -1 si erreur

write

```
#include <unistd.h>
size_t write(int desc, void * buf, size_t noocts);
```

desc : descripteur fichier ouvert en écriture
buf : un pointer à un tampon
noocts : le nombre d'octets qu'on veut écrire

Retourne : nombre caractères écrits, -1 si erreur

Algorithme `write`

```
nb_ecrit = write(p[1], buf, n)
```

Si `n < PIPE_BUF` : écriture atomique.

lecteurs = 0 :
signal `SIGPIPE` envoyé à l'écrivain.

lecteurs ≠ 0 :
écriture bloquante (par défaut) :
on revient quand exactement n caractères ont été écrits
lecture non bloquante (option `O_NONBLOCK` positionné) :
`n > PIPE_BUF` :
ret. nombre < n
`n ≤ PIPE_BUF` et tube a n places libres :
écriture atomique.
`n ≤ PIPE_BUF` et tube n'a pas n places libres :
aucune écriture, retourne 0.

Algorithme `write`

```
nb_ecrit = write(p[1], buf, n)
```

Si `n < PIPE_BUF` : écriture atomique.

lecteurs = 0 :
signal SIGPIPE envoyé à l'écrivain.

lecteurs ≠ 0 :
écriture bloquante (par défaut) :
on revient quand exactement n caractères ont été écrits
lecture non bloquante (option O_NONBLOCK positionnée) :
n > PIPE_BUF :
ret nombre < n
n ≤ PIPE_BUF et tube a n places libres :
écriture atomique.
n ≤ PIPE_BUF et tube n'a pas n places libres:
aucune écriture, retourne 0.

Algorithme `write`

```
nb_ecrit = write(p[1], buf, n)
```

Si `n < PIPE_BUF` : écriture atomique.

lecteurs = 0 :

signal `SIGPIPE` envoyé à l'écrivain.

lecteurs ≠ 0 :

écriture bloquante (par défaut) :

on revient quand exactement `n` caractères ont été écrits.

lecture non bloquante (option `O_NONBLOCK` positionné) :

`n` > `PIPE_BUF`

ret. nombre < `n`

`n` ≤ `PIPE_BUF` et tube a `n` places libres :

écriture atomique.

`n` ≤ `PIPE_BUF` et tube n'a pas `n` places libres:

aucune écriture, retourne 0.

Algorithme `write`

```
nb_ecrit = write(p[1], buf, n)
```

Si `n < PIPE_BUF` : écriture atomique.

lecteurs $= 0$:

signal `SIGPIPE` envoyé à l'écrivain.

lecteurs $\neq 0$:

écriture bloquante (par défaut) :

on revient quand exactement `n` caractères ont été écrits.

lecture non bloquante (option `O_NONBLOCK` positionné) :

`n` > `PIPE_BUF`

ret nombre < `n`

`n` $\leq$ `PIPE_BUF` et tube a `n` places libres :

écriture atomique.

`n` $\leq$ `PIPE_BUF` et tube n'a pas `n` places libres:

aucune écriture, retourne 0.

Algorithme `write`

```
nb_ecrit = write(p[1], buf, n)
```

Si `n < PIPE_BUF` : écriture atomique.

lecteurs = 0 :

signal `SIGPIPE` envoyé à l'écrivain.

lecteurs ≠ 0 :

écriture bloquante (par défaut) :

on revient quand exactement `n` caractères ont été écrits.

lecture non bloquante (option `O_NONBLOCK` positionné) :

`n > PIPE_BUF`

ret nombre < `n`

`n` ≤ `PIPE_BUF` et tube a `n` places libres :

écriture atomique.

`n` ≤ `PIPE_BUF` et tube n'a pas `n` places libres:

aucune écriture, retourne 0.

Algorithme `write`

```
nb_ecrit = write(p[1], buf, n)
```

Si `n < PIPE_BUF` : écriture atomique.

\# lecteurs $= 0$:

signal `SIGPIPE` envoyé à l'écrivain.

\# lecteurs $\neq 0$:

écriture bloquante (par défaut) :

on revient quand exactement `n` caractères ont été écrits.

lecture non bloquante (option `O_NONBLOCK` positionné) :

`n > PIPE_BUF`

ret nombre $<$ `n`

`n` $\leq$ `PIPE_BUF` et tube a `n` places libres :

écriture atomique.

`n` $\leq$ `PIPE_BUF` et tube n'a pas `n` places libres:

aucune écriture, retourne 0.

Algorithme `write`

```
nb_ecrit = write(p[1], buf, n)
```

Si `n < PIPE_BUF` : écriture atomique.

\# lecteurs $= 0$:

signal `SIGPIPE` envoyé à l'écrivain.

\# lecteurs $\neq 0$:

écriture bloquante (par défaut) :

on revient quand exactement `n` caractères ont été écrits.

lecture non bloquante (option `O_NONBLOCK` positionné) :

`n > PIPE_BUF`

ret nombre < `n`

`n` $\leq$ `PIPE_BUF` et tube a `n` places libres :

écriture atomique.

`n` $\leq$ `PIPE_BUF` et tube n'a pas `n` places libres:

aucune écriture, retourne 0.

Algorithme `write`

```
nb_ecrit = write(p[1], buf, n)
```

Si `n < PIPE_BUF` : écriture atomique.

\# lecteurs $= 0$:
signal `SIGPIPE` envoyé à l'écrivain.

\# lecteurs $\neq 0$:

écriture bloquante (par défaut) :
on revient quand exactement `n` caractères ont été écrits.

lecture non bloquante (option `O_NONBLOCK` positionné) :

`n > PIPE_BUF`
ret nombre < `n`

`n` $\leq$ `PIPE_BUF` et tube a `n` places libres :
écriture atomique.

`n` $\leq$ `PIPE_BUF` et tube n'a pas `n` places libres:
aucune écriture, retourne 0.

Algorithme `write`

```
nb_ecrit = write(p[1], buf, n)
```

Si `n < PIPE_BUF` : écriture atomique.

\# lecteurs $= 0$:
signal `SIGPIPE` envoyé à l'écrivain.

\# lecteurs $\neq 0$:

écriture bloquante (par défaut) :
on revient quand exactement `n` caractères ont été écrits.

lecture non bloquante (option `O_NONBLOCK` positionné) :

`n > PIPE_BUF`
ret nombre < `n`

`n` $\leq$ `PIPE_BUF` et tube a `n` places libres :
écriture atomique.

`n` $\leq$ `PIPE_BUF` et tube n'a pas `n` places libres:
aucune écriture, retourne 0.

Programme : execriture.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <limits.h>
5
6  void fils(int d_ecriture); void pere(int d_lecture);
7
8  int main(void)
9  {
10   int p[2];
11
12   if(pipe(p) == -1) exit(EXIT_FAILURE);
13
14   switch( fork() )
15     {
16     case -1 :
17       exit(EXIT_FAILURE);
18     case 0 : /* Le fils écrive dans le tube */
19       close(p[0]);
20       fils(p[1]);
21     default : /* Le père lit du tube */
22       close(p[1]);
23       pere(p[0]);
24     }
25   exit(EXIT_SUCCESS);
26 }
```

Programme : execriture.c (II)

```
28 void fils(int d_ecriture)
29 {
30   char buf[3 * PIPE_BUF + 34];
31   size_t no_ecrit;
32
33   no_ecrit = write(d_ecriture,buf,sizeof(buf));
34   printf("Sortie de \"write\" avec %d caracteres écrits.\n",
35          no_ecrit);
36
37   exit(EXIT_SUCCESS);
38 }
39
40 void pere(int d_lecture)
41 {
42   char tampon[PIPE_BUF];
43   size_t no_lu; int i=0;
44
45   while((no_lu = read(d_lecture,tampon,sizeof(tampon))) > 0)
46     printf("Lecture %d: %d octets lus.\n",++i,no_lu);
47
48   exit(EXIT_SUCCESS);
49 }
```

Session : execriture

```
$ gcc -Wall -pedantic execriture.c
$ a.out
Lecture 1: 4096 octets lus.
Lecture 2: 4096 octets lus.
Lecture 3: 4096 octets lus.
Sortie de "write" avec 12322 caracteres écrits.
Lecture 4: 34 octets lus.
$
```

Plan

fcntl

```
#include <fcntl.h>
int fcntl(int desc, int cmd, int opts);
```

desc : descripteur du fichier ouvert dont on souhaite modifier les caracteristiques.

cmd : F_GETFL : accès au drapeau du fichier,
F_SETFL : modification du drapeau du fichier.

opts : |de :
O_NONBLOCK : rendre lecture/écriture non bloquante.

Sommaire :
accès ou modification des caractéristiques d'un fichier déjà ouvert.

fcntl

```
#include <fcntl.h>
int fcntl(int desc, int cmd, int opts);
```

desc : descripteur du fichier ouvert dont on souhaite modifier les caracteristiques.

cmd : F_GETFL : accès au drapeau du fichier,
F_SETFL : modification du drapeau du fichier.

opts : |de :
O_NONBLOCK : rendre lecture/écriture non bloquante.

Sommaire :
accès ou modification des caractéristiques d'un fichier déjà ouvert.

fcntl

```
#include <fcntl.h>
int fcntl(int desc, int cmd, int opts);
```

desc : descripteur du fichier ouvert dont on souhaite modifier les caracteristiques.

cmd : F_GETFL : accès au drapeau du fichier,
F_SETFL : modification du drapeau du fichier.

opts : |de :
O_NONBLOCK : rendre lecture/écriture non bloquante.

Sommaire :
accès ou modification des caractéristiques d'un fichier déjà ouvert.

fcntl

```
#include <fcntl.h>
int fcntl(int desc, int cmd, int opts);
```

desc : descripteur du fichier ouvert dont on souhaite modifier les caracteristiques.

cmd : F_GETFL : accès au drapeau du fichier,
F_SETFL : modification du drapeau du fichier.

opts : |de :
O_NONBLOCK : rendre lecture/écriture non bloquante.

Sommaire :
accès ou modification des caractéristiques d'un fichier déjà ouvert.

fcntl

```
#include <fcntl.h>
int fcntl(int desc, int cmd, int opts);
```

desc : descripteur du fichier ouvert dont on souhaite modifier les caracteristiques.

cmd : F_GETFL : accès au drapeau du fichier,
F_SETFL : modification du drapeau du fichier.

opts : |de :
O_NONBLOCK : rendre lecture/écriture non bloquante.

Sommaire :
accès ou modification des caractéristiques d'un fichier déjà ouvert.

Exemple

Rendre la lecture dans un tube non bloquante :

```
{
...
status_lecture = fcntl(p[0],F_GETFL);
fcntl(p[0],F_SETFL,status_lecture | O_NONBLOCK);
...
}
```

close

```
#include <unistd.h>
int close(int desc);
```

desc : le descripteur du fichier qu'on veut fermer

Remarques : ne pas fermer un tube peut entraîner un bloquage.

close

```
#include <unistd.h>
int close(int desc);
```

desc : le descripteur du fichier qu'on veut fermer

Remarques : ne pas fermer un tube peut entraîner un bloquage.

close

```
#include <unistd.h>
int close(int desc);
```

desc : le descripteur du fichier qu'on veut fermer

Remarques : ne pas fermer un tube peut entraîner un bloquage.

Programme : fermeturedesc.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4
5  void fils(int d_ecriture); void pere(int d_lecture);
6
7  int main(void)
8  {
9    int p[2];
10
11   if(pipe(p) == -1) exit(EXIT_FAILURE);
12
13   switch( fork() )
14     {
15     case -1 :
16       exit(EXIT_FAILURE);
17     case 0 : /* Le fils écrive dans le tube */
18       close(p[0]);
19       fils(p[1]);
20     default : /* Le père lit du tube */
21       /* close(p[1]); */ /* On oubli de fermer ce descripteur. */
22       pere(p[0]);
23     }
24   exit(EXIT_SUCCESS);
25 }
```

Programme : fermeturedesc.c (II)

```
27  void fils(int d_ecriture)
28  {
29    char message[]="abcde";
30
31    write(d_ecriture,message,sizeof(message));
32
33    exit(EXIT_SUCCESS);
34  }
35
36  void pere(int d_lecture)
37  {
38    char tampon[100];
39
40    while(read(d_lecture,tampon,sizeof(tampon)) > 0)
41      printf("%s\n",tampon);
42    printf("Fin du père");
43    exit(EXIT_SUCCESS);
44  }
```

Session : fermeturedesc

```
$ gcc -Wall -pedantic fermeturedesc.c
$ a.out
abcde
^C
$
```

Plan

fdopen, popen

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

FILE * fdopen(int *desc*, const char * *mode*);

desc : descripteur du fichier ouvert.

mode : l'un de "r", "w", "a", "r+", "w+", "a+".

Sommaire :
transforme un descripteur d'un fichier ouvert en un flot C.

FILE * popen(const char * *cmd*, const char * *mode*);

cmd : la ligne de commande qu'on passe au shell.

mode : "r" : la sortie standard de la commande est redirigée sur le flot

"w" : la sortie standard du programme est envoyée au stdin de la commande.

Sommaire : il exécute la commande *cmd*, crée une tube entre les deux processus, un extrémité du tube est transformé en flot C.

fdopen, popen

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

FILE * fdopen(int *desc*, const char * *mode*);

desc : descripteur du fichier ouvert.

mode : l'un de "r", "w", "a", "r+", "w+", "a+".

Sommaire :
transforme un descripteur d'un fichier ouvert en un flot C.

FILE * popen(const char * *cmd*, const char * *mode*);

cmd : la ligne de commande qu'on passe au shell.

mode : "r" : la sortie standard de la commande est redirigée sur le flot
"w" : la sortie standard du programme est envoyée au stdin de la commande.

Sommaire : il exécute la commande *cmd*, crée une tube entre les deux processus, un extrémité du tube est transformé en flot (

fdopen, popen

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

FILE * fdopen(int *desc*, const char * *mode*);

desc : descripteur du fichier ouvert.

mode : l'un de "r", "w", "a", "r+", "w+", "a+".

Sommaire :
transforme un descripteur d'un fichier ouvert en un flot C.

FILE * popen(const char * *cmd*, const char * *mode*);

cmd : la ligne de commande qu'on passe au shell.

mode : "r" : la sortie standard de la commande est redirigée sur le flot
"w" : la sortie standard du programme est envoyée au stdin de la commande.

Sommaire : il exécute la commande *cmd*, crée une tube entre les deux processus, un extrémité du tube est transformé en flot C.

Exemple : fdopen

```
int p[2];
FILE *ptr;

...
pipe(&p);
ptr = fdopen(p[1],"w");
fprintf(ptr,"Une façon d'écrire dans un tube.");
```

Exemple : popen

```
FILE *ptr;
int heure,minutes;
...
ptr = popen("date +%Hh%D","r");
fscanf(ptr,"%dh%d",&heure,&minutes);
```