La communication par tubes

Luigi Santocanale

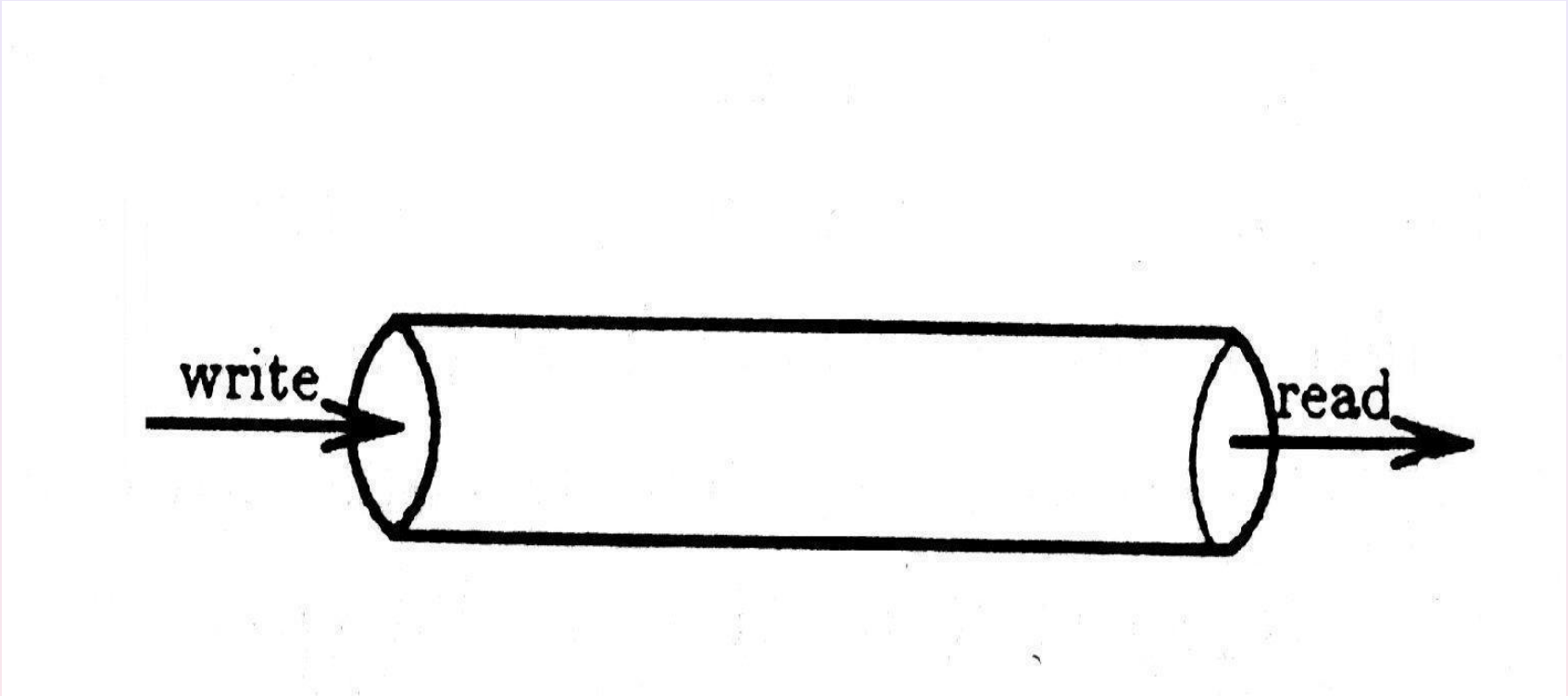
Laboratoire d'Informatique Fondamentale,
Centre de Mathématiques et Informatique,
39, rue Joliot-Curie - F-13453 Marseille

2 novembre 2004

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.

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.

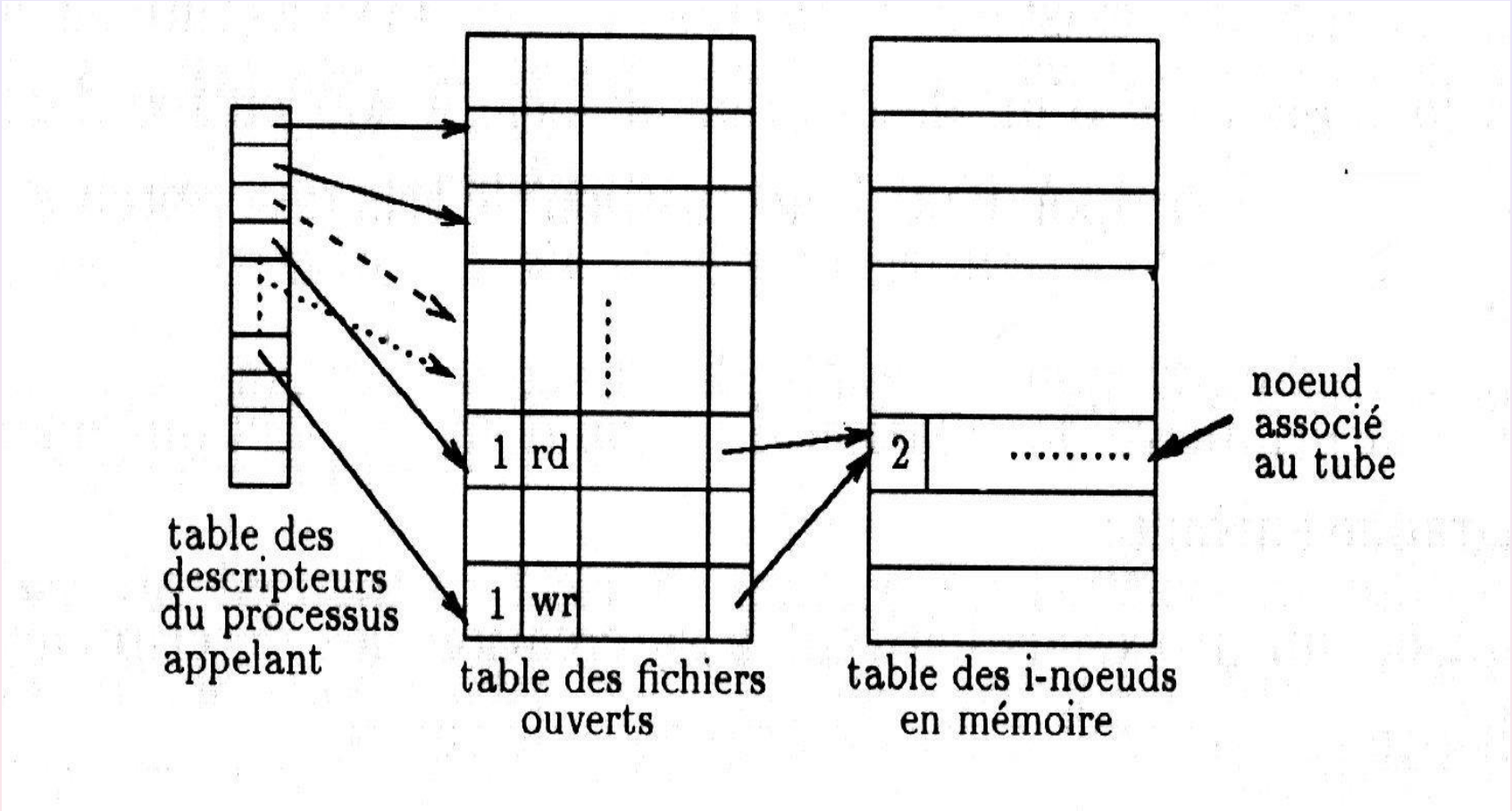
Niveau implémentation

- 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 % 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 % 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

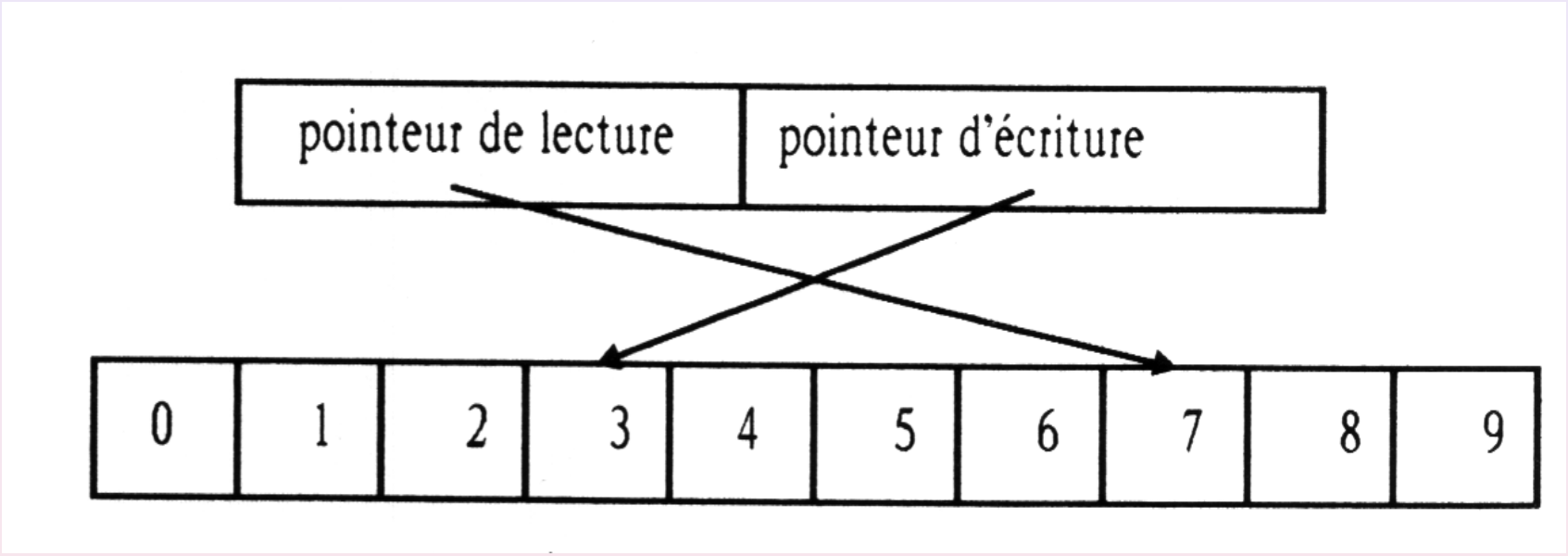
- Un tube est un i-noeud sur une périphérique dédiée aux tubes.
- Seulement les bloques adressés de façon directe sont utilisé.
- Les tubes sont implémentés en tant que *files circulaires* : pointeur d'écriture, pointeur de lecture.
- Les pointeurs sont mis à jour après chaque opération.
- Taille du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

```
ptr_lecture - ptr_ecriture % tailletotale
```

- Espace libre dans le tube :

```
ptr_ecriture - ptr_lecture % tailletotale
```

Tubes comme files circulaires (Bach 1989)



Types de tubes

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

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

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

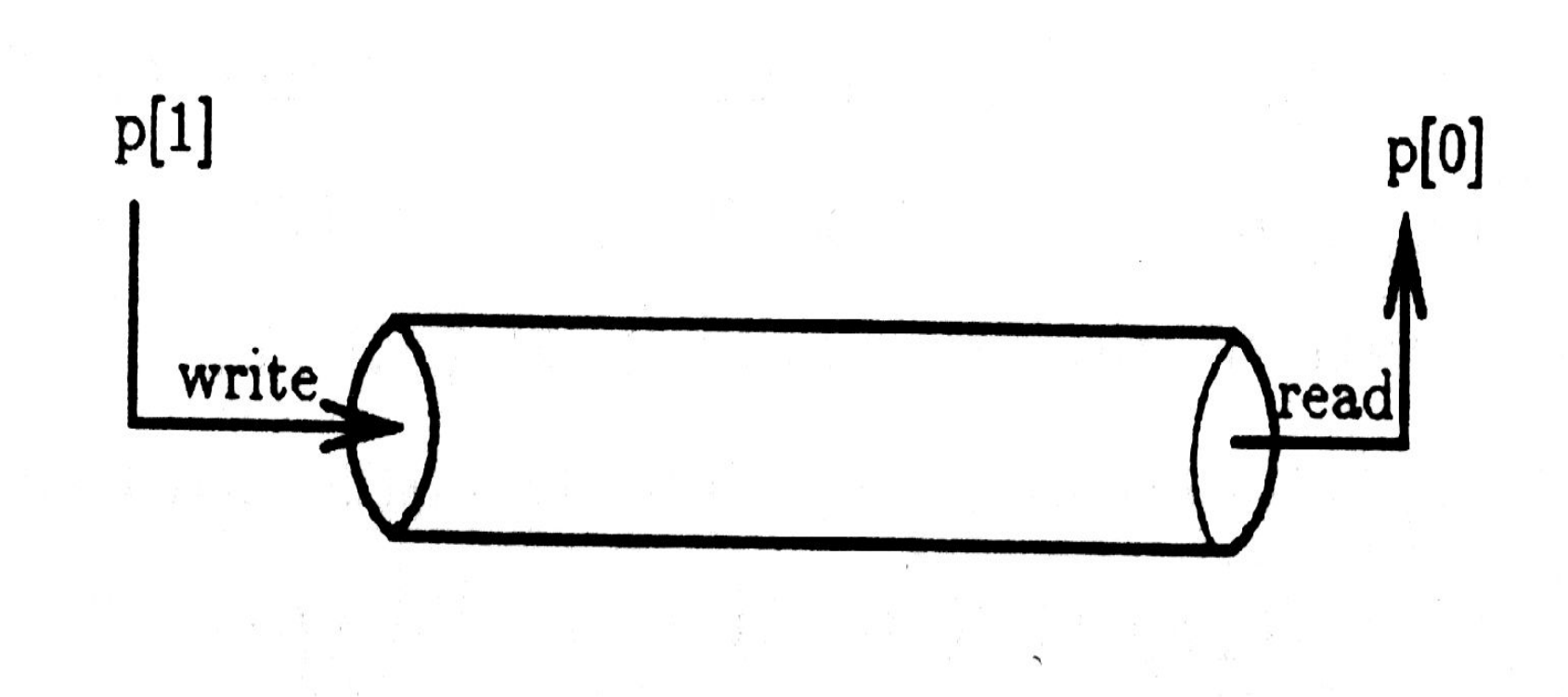
```
#include <unistd.h>
int pipe(int p[2]);
```

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);
 6 : void pere(int d_lecture);
 7 :
 8 : int main(void)
 9 : {
10 :     int p[2];
11 :
12 :     if (pipe(p) == -1)
13 :         exit(EXIT_FAILURE);
14 :
15 :     switch (fork())
16 :     {
17 :     case -1:
18 :         exit(EXIT_FAILURE);
19 :     case 0:                    /* Le fils écrive dans le tube */
20 :         close(p[0]);
21 :         fils(p[1]);
22 :     default:                   /* Le père lit du tube */
23 :         close(p[1]);
24 :         pere(p[0]);
25 :     }
26 :     exit(EXIT_SUCCESS);
27 : }
28 :
```

Programme : expipe.c

```
29 : void fils(int d_ecriture)
30 : {
31 :     char message[] = "abcde";
32 :     int no_ecrit;
33 :
34 :     no_ecrit = write(d_ecriture, message, sizeof(message));
35 :
36 :     printf("Le fils dit :\n"
37 :            "\tJ'ai écrit %d octets dans le fichier %d.\n", no_ecrit,
38 :            d_ecriture);
39 :     printf("\tCe que j'ai écrit : \"%s\"\n", message);
40 :
41 :     exit(EXIT_SUCCESS);
42 : }
43 :
44 : void pere(int d_lecture)
45 : {
46 :     char tampon[100];
47 :     int no_lu;
48 :
49 :     no_lu = read(d_lecture, tampon, sizeof(tampon));
50 :
51 :     printf("Le père dit :\n" "\tJ'ai lu %d octets du fichier %d.\n", no_lu,
52 :            d_lecture);
53 :     printf("\tCe que j'ai lu : \"%s\"\n", tampon);
54 :
55 :     exit(EXIT_SUCCESS);
56 : }
```

Session : expipe

```
[lsantoca@localhost lecture5]$ gcc -Wall -pedantic expipe.c
[lsantoca@localhost lecture5]$ 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.

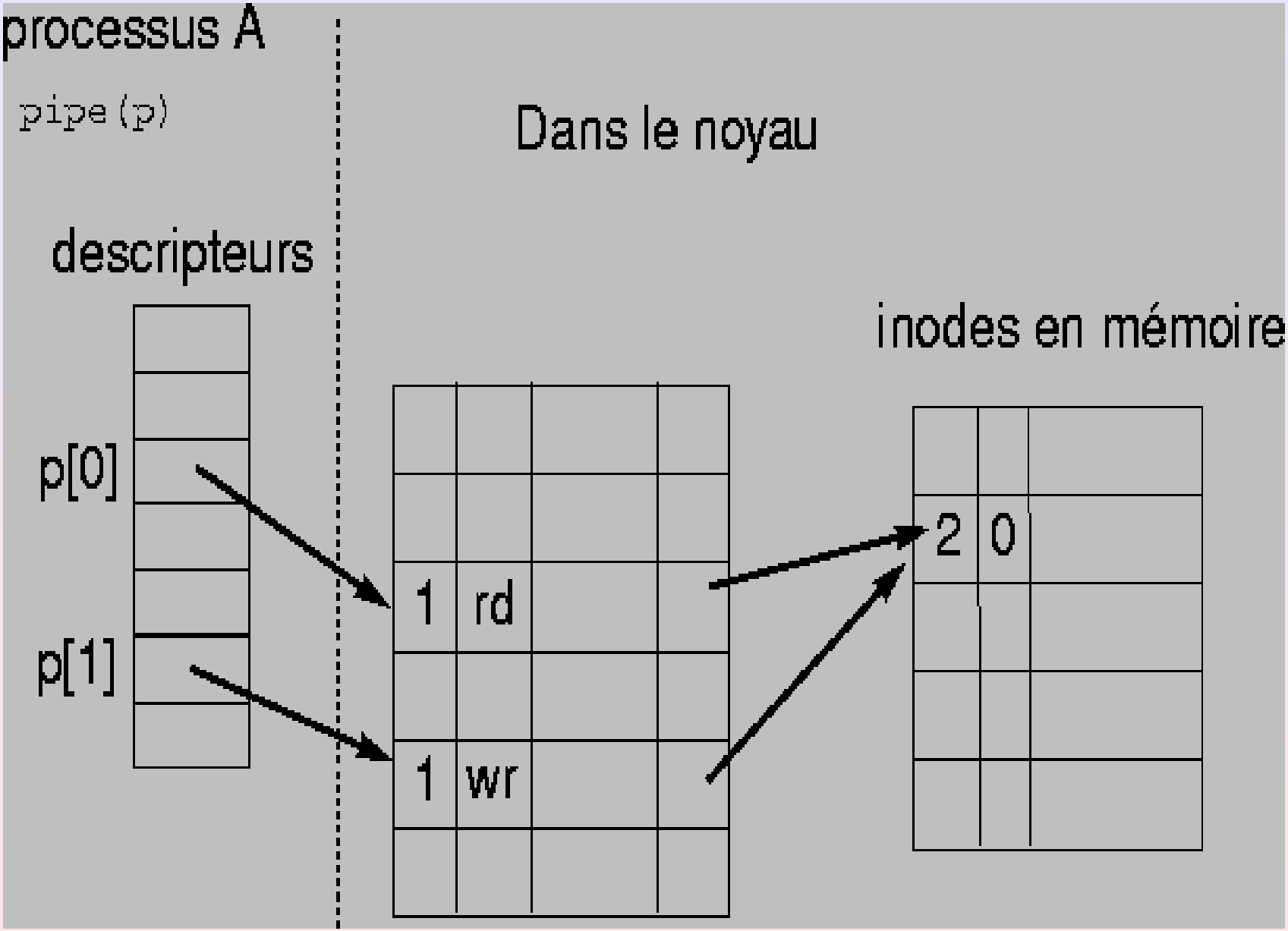
Considérations

- 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.

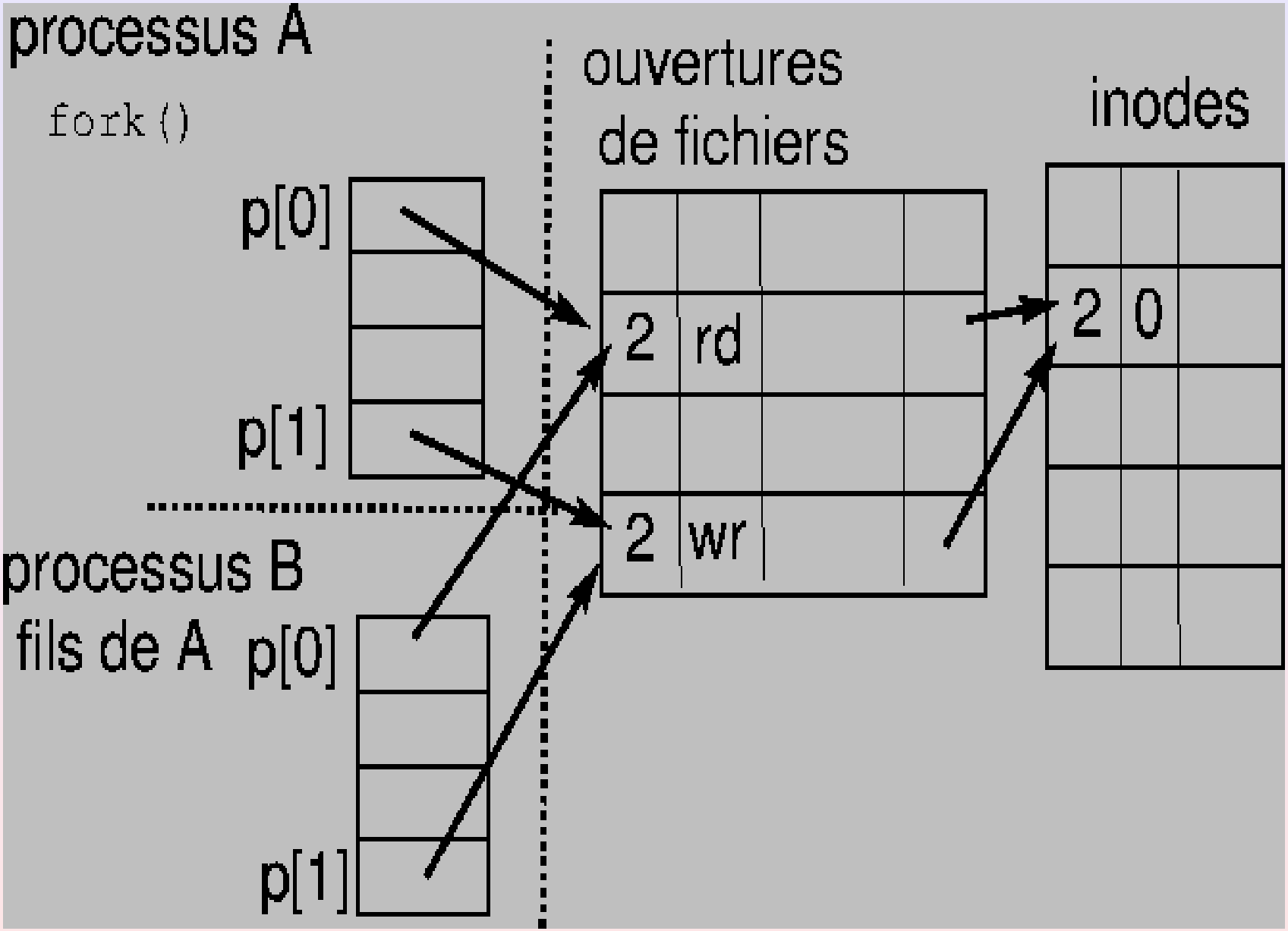
Considérations

- 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 !!!

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 ENTREE 0
 6 : #define SORTIE 1
 7 :
 8 : void codecommun(int d_lecture, int d_ecriture)
 9 : {
10 :     char buf[100];
11 :     int no_lu, no_ecrit;
12 :
13 :     no_lu = read(d_lecture, buf, sizeof(buf));
14 :     no_ecrit = write(d_ecriture, buf, sizeof(buf));
15 :     printf("[%d] No. lus : %d, no. écrits : %d.\n", getpid(), no_lu,
16 :            no_ecrit);
17 : }
18 :
```

Programme : interbloquage.c

```
19 : int main(void)
20 : {
21 :     int pf[2];                  /* Tube père -> fils */
22 :     int fp[2];                  /* Tube fils -> père */
23 :
24 :     if (pipe(pf) == -1 || pipe(fp) == -1)
25 :         exit(EXIT_FAILURE);
26 :
27 :     switch (fork())
28 :     {
29 :     case -1:
30 :         exit(EXIT_FAILURE);
31 :     case 0:                     /* Le fils */
32 :         close(pf[ENTREE]);
33 :         close(fp[SORTIE]);
34 :         codecommun(fp[ENTREE], pf[SORTIE]);
35 :         break;
36 :     default:                    /* Le père */
37 :         close(pf[SORTIE]);
38 :         close(fp[ENTREE]);
39 :         codecommun(pf[ENTREE], fp[SORTIE]);
40 :     }
41 :     exit(EXIT_SUCCESS);
42 : }
```

Session : interbloquage

```
[lsantoca@localhost lecture5]$ gcc -Wall -pedantic interbloquage.c
[lsantoca@localhost lecture5]$ a.out & ps -ao cmd,s
[4] 3840
CMD                S
emacs -rv -fn *- S
a.out              S
a.out              S
ps -ao 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` ≤ `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 ≠ 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.

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);
 7 : void pere(int d_lecture);
 8 :
 9 : int main(void)
10 : {
11 :     int p[2];
12 :
13 :     if (pipe(p) == -1)
14 :         exit(EXIT_FAILURE);
15 :
16 :     switch (fork())
17 :     {
18 :     case -1:
19 :         exit(EXIT_FAILURE);
20 :     case 0:                    /* Le fils écrive dans le tube */
21 :         close(p[0]);
22 :         fils(p[1]);
23 :     default:                   /* Le père lit du tube */
24 :         close(p[1]);
25 :         pere(p[0]);
26 :     }
27 :     exit(EXIT_SUCCESS);
28 : }
29 :
```

Programme : execriture.c

```
30 : void fils(int d_ecriture)
31 : {
32 :     char buf[3 * PIPE_BUF + 34];
33 :     size_t no_ecrit;
34 :
35 :     no_ecrit = write(d_ecriture, buf, sizeof(buf));
36 :     printf("Sortie de la primitive write avec %d caracteres écrits.\n",
37 :            no_ecrit);
38 :
39 :     exit(EXIT_SUCCESS);
40 : }
41 :
42 : void pere(int d_lecture)
43 : {
44 :     char tampon[PIPE_BUF];
45 :     size_t no_lu;
46 :     int i = 0;
47 :
48 :     while ((no_lu = read(d_lecture, tampon, sizeof(tampon))) > 0)
49 :         printf("Lecture %d: %d octets lus.\n", ++i, no_lu);
50 :
51 :     exit(EXIT_SUCCESS);
52 : }
```

Session : execriture

```
[lsantoca@localhost lecture5]$ gcc -Wall -pedantic execriture.c
[lsantoca@localhost lecture5]$ a.out
Lecture 1: 4096 octets lus.
Lecture 2: 4096 octets lus.
Lecture 3: 4096 octets lus.
Sortie de la primitive 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.

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);
 6 : void pere(int d_lecture);
 7 :
 8 : int main(void)
 9 : {
10 :     int p[2];
11 :
12 :     if (pipe(p) == -1)
13 :         exit(EXIT_FAILURE);
14 :
15 :     switch (fork())
16 :     {
17 :     case -1:
18 :         exit(EXIT_FAILURE);
19 :     case 0:                     /* Le fils écrive dans le tube */
20 :         close(p[0]);
21 :         fils(p[1]);
22 :     default:                    /* Le père lit du tube */
23 :         /* close(p[1]); *//* On oubli de fermer ce descripteur. */
24 :         pere(p[0]);
25 :     }
26 :     exit(EXIT_SUCCESS);
27 : }
```

Programme : fermeturedesc.c

```
28 :
29 : void fils(int d_ecriture)
30 : {
31 :     char message[] = "abcde";
32 :
33 :     write(d_ecriture, message, sizeof(message));
34 :
35 :     exit(EXIT_SUCCESS);
36 : }
37 :
38 : void pere(int d_lecture)
39 : {
40 :     char tampon[100];
41 :
42 :     while (read(d_lecture, tampon, sizeof(tampon)) > 0)
43 :         printf("%s\n", tampon);
44 :     printf("Fin du père");
45 :     exit(EXIT_SUCCESS);
46 : }
```

Session : fermeturedesc

```
[lsantoca@localhost lecture5]$ gcc -Wall -pedantic fermeturedesc.c
[lsantoca@localhost lecture5]$ a.out
abcde
^C
[lsantoca@localhost lecture5]$
```

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 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 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);
```