Noyau d'un système d'exploitation
INF2610

Chapitre 4 : Communication Interprocessus

Département de génie informatique et génie logiciel

POLYTECHNIQUE MONTRÉAL
AFFILIÉE À L'UNIVERSITÉ DE MONTRÉAL

Hiver 2019

Chapitre 4 - Communication Interprocessus

- **Introduction**
- **Les segments de données partagés**
- **Les tubes de communication UNIX**
  - **Tubes anonymes**
  - **Tubes nommés**
- **Les signaux**

Introduction

- Les systèmes d'exploitation offrent la possibilité de créer plusieurs processus ou fils (threads) concurrents qui coopèrent pour réaliser des applications complexes.
- Ces processus s'exécutent sur un même ordinateur (monoprocesseur ou multiprocesseur) ou sur des ordinateurs différents, et peuvent partager ou s'échanger des informations (communication interprocessus).
- Il existe plusieurs mécanismes de communication interprocessus :
  - les données communes (variables, fichiers, segments de données),
  - les messages et
  - les signaux (notifications).

Introduction (2)

Données partagées

- Chaque processus a un espace d'adressage privé partagé par tous ses threads. Ses threads partagent aussi la table des descripteurs de fichiers (TDF), la table des gestionnaires de signaux (TGS) du processus, etc.

- Il peut aussi créer dynamiquement des **segments de données** et rattacher à son espace d'adressage des segments de données créés. Chaque segment de données créé est partagé par tous les processus qui l'ont rattaché à leurs espaces d'adressage.

- Lors de la création d'un processus (fork),
  - la table des descripteurs de fichiers est dupliquée. Les processus créateur et créé partagent le même pointeur de fichier pour chaque fichier déjà ouvert lors de la création, et
  - le processus fils hérite et partage tous les segments de données rattachés à l'espace d'adressage du père.

Introduction (3)

Messages :

- Il existe plusieurs mécanismes de communication par messages dans les systèmes d'exploitation de la famille Unix :
  - **les tubes de communication (man 7 pipe),**
  - les files de messages (man 7 mq_overview)
  - les sockets (http://sdz.tdct.org/sdz/les-sockets.html), etc.

- **Les tubes de communication** permettent à deux ou plusieurs processus s'exécutant sur une même machine d'échanger des informations sous forme **de flots de caractères.**

- Les files de messages offrent la possibilité à deux ou plusieurs processus s'exécutant sur une même machine d'échanger des informations sous forme **de flots de messages délivrés selon la politique « le plus prioritaire d'abord »**.

Signaux :

- Les systèmes d'exploitation de la famille Unix gèrent un ensemble de signaux qui permettent aux processus de **réagir aux événements sans être obligés de tester en permanence leurs arrivés**.

Segments de données partagés

- Unix-Linux offrent plusieurs appels système pour créer, annexer et détacher dynamiquement des segments de données à l'espace d'adressage d'un processus.

- Les appels système de POSIX pour les segments de données partagés sont dans la librairie <sys/mman.h> (man 7 shm_overview) :
  - L'appel système **shm_open** permet de créer ou de retrouver un segment de données. Il retourne un descripteur de fichier.
  - L'appel système **mmap** permet d'attacher un segment de données à un processus.
  - L'appel système **munmap** permet de détacher un segment de données d'un processus.
  - L'appel système **shm_unlink** permet de supprimer le segment de données lorsqu'il sera détaché de tous les espaces d'adressage.

Segments de données partagés (2) : Exemple 1

- Les deux programmes suivants communiquent au moyen d'un segment de données créé par le premier.

- Le premier programme crée puis attache un segment de données à son espace d'adressage. Il écrit ensuite dans ce segment, attend 1 seconde puis détache le segment de son espace d'adressage.

- Le second programme ouvre puis attache le segment de données à son espace d'adressage. Il y accède en lecture avant de le détacher de son espace d'adressage. Enfin, il informe le système de supprimer le segment lorsqu'il n'est plus attaché à aucun espace d'adressage.

Segments de données partagés (3) : Exemple 1

```
// shmp1.c
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

int main( ) {

        off_t   taille = 4096;
        char   *nom    = "/inf2610shm";
        int     fd = shm_open(nom, O_RDWR|O_CREAT, 0600 );

        if(fd==-1) exit(1);
        printf ("Processus %d : a cree  ou ouvert un segment\n", getpid());
        ftruncate(fd, taille);
```

Segments de données partagés (4) : Exemple 1

```
// shmp1.c
    char *ptr = (char *) mmap(NULL, taille, PROT_READ|PROT_WRITE,
                              MAP_SHARED, fd, 0);
      if(ptr==NULL) exit(1);

      close(fd);
      printf("Processus %d : a attache le segment %s  a son espace
                  d'adressage: %p\n", getpid(), nom, ptr);

      char msg[100];
      sprintf(msg, "bonjour du processus %d\n",getpid());
      printf("Processus %d ecrit dans le segment le message %s\n",
            getpid(),msg);
      strcpy((char*)ptr,msg);

      munmap(ptr,taille);
      exit(0);
}
```

Segments de données partagés (5) : Exemple 1

```
// shmp2.c
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

 int main( ) {
       char   *nom    = "/inf2610shm";
       int     fd = shm_open(nom, O_RDWR, 0600 );

       if(fd==-1) exit(1);
       printf ("Processus %d a ouvert le segment %s\n", getpid(), nom);

        struct stat info;
       fstat(fd, &info);
       off_t taille = info.st_size ;
```

Segments de données partagés (6) : Exemple

```
// shmp2.c
   char *ptr = (char *) mmap(NULL, taille, PROT_READ|PROT_WRITE,
                                          MAP_SHARED, fd, 0);
    close(fd);

    printf ("Processus %d a attache le segment %s a son espace
              d'adressage :%p \n", getpid(), nom,ptr);
    if(ptr==NULL) exit(1);

    sleep(1);

    printf("Processus %d lit du segment le message  %s",
                    getpid(), ptr);

    munmap(ptr,taille);
    shm_unlink(nom);
    exit(0);
}
```

Segments de données partagés (7) : Exemple 1

```
jupiter$ gcc shmp1.c -o shmp1 -lrt
jupiter$ gcc shmp2.c -o shmp2 -lrt
jupiter$ ./shmp1 & ./shmp2
[1] 8542
Processus 8542 a cree ou ouvert un segment
Processus 8542 a attache le segment /inf2610shm  a son espace
d'adressage: 0x7f89f5819000
Processus 8542 ecrit dans le segment le message bonjour du processus
8542

Processus 8543 a ouvert le segment /inf2610shm
Processus 8543 a attache le segment /inf2610shm a son espace
d'adressage :0x7f4c8c11d000
Processus 8543 lit du segment le message  bonjour du processus 8542
[1]+  Fini                    ./shmp1
jupiter$
```

Segments de données partagés (8) : Exemple 1

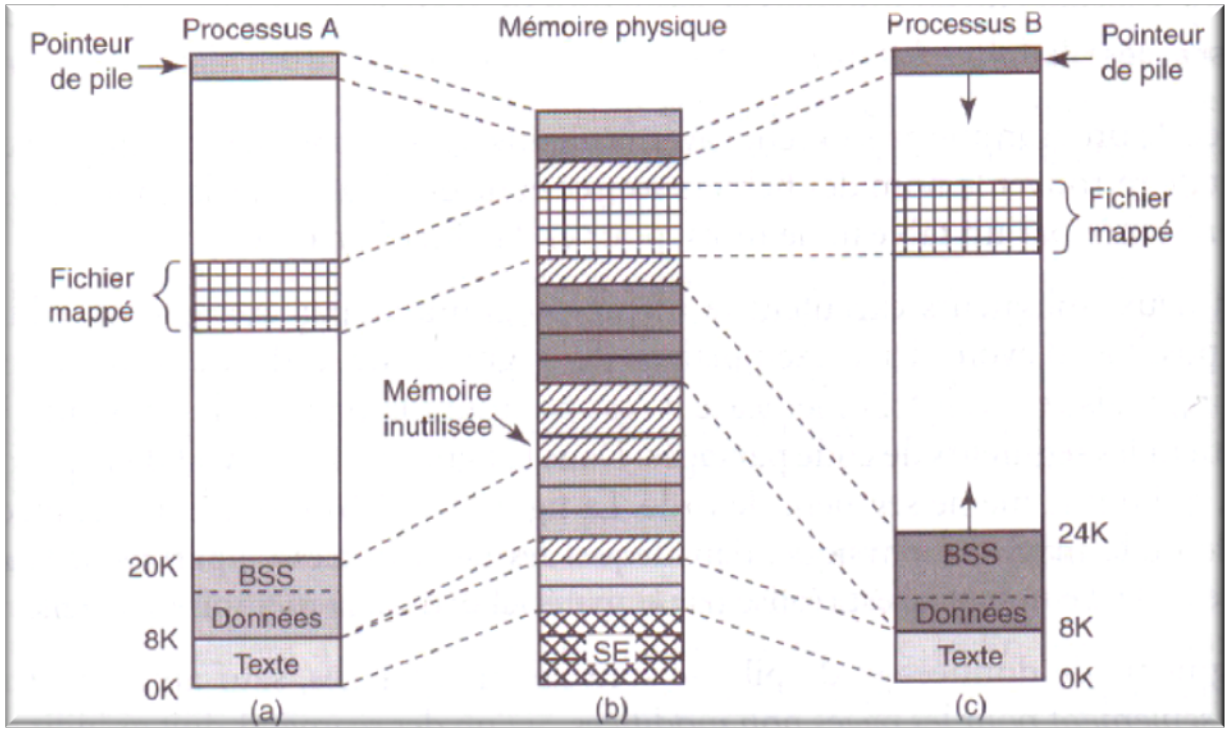
jupiter$ ./shmp1
Processus 26965 : a cree ou ouvert un segment
Processus 26965 : a attache le segment /inf2610shm a son espace d'adressage: 0x7f94a249f000
Processus 26965 ecrit dans le segment le message bonjour du processus 26965

jupiter$ **ls /dev/shm**
Inf2610shm
Jupiter$ **cat /dev/shm/inf2610shm**
bonjour du processus 26965
jupiter$ ./shmp2
Processus 27038 a ouvert le segment /inf2610shm
Processus 27038 a attache le segment /inf2610shm a son espace d'adressage :0x7f9a4ebc2000
Processus 27038 lit du segment le message bonjour du processus 26965
jupiter$ **ls /dev/shm**
jupiter$

Segments de données partagés (9)

- Un processus peut mapper un fichier en mémoire (memory-mapped file) → faire correspondre un fichier à une partie de l'espace d'adressage du processus (les fonctions mmap et munmap).

- Plusieurs processus peuvent partager un fichier mappé en mémoire.



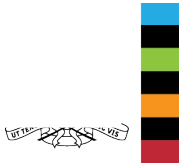
Tannenbaum

Segments de données partagés (10) : Exemple 2

```
//mmapf.c   mappage d'un fichier
#include <fcntl.h>
 #include <stdio.h>
 #include <stdlib.h>
 #include <unistd.h>
 #include <sys/stat.h>
 #include <sys/mman.h>

 int main(int argc, char *argv[]) {
      char   *projection;
      int    fd;
      struct stat attr;
      long   fsize;
      int    i;
      char   tmp;
      if(argc != 2)  exit(1);

      fd = open(argv[1],O_RDWR);
      if(fd==-1) exit(1);
      stat(argv[1], &attr);
```

Segments de données partagés (11) : Exemple 2

```
    fsize = attr.st_size;
    projection = (char *) mmap(NULL, fsize, PROT_READ |
                        PROT_WRITE, MAP_SHARED, fd, 0);
    if(projection ==NULL) exit(1);
    close(fd);
    for(i=0; i<fsize/2; i++) {
        tmp = projection[i];
        projection[i] = projection[fsize - i -1];
        projection[fsize - i -1] = tmp;
    }
    munmap((void *) projection, fsize);
    return 0;
}
```



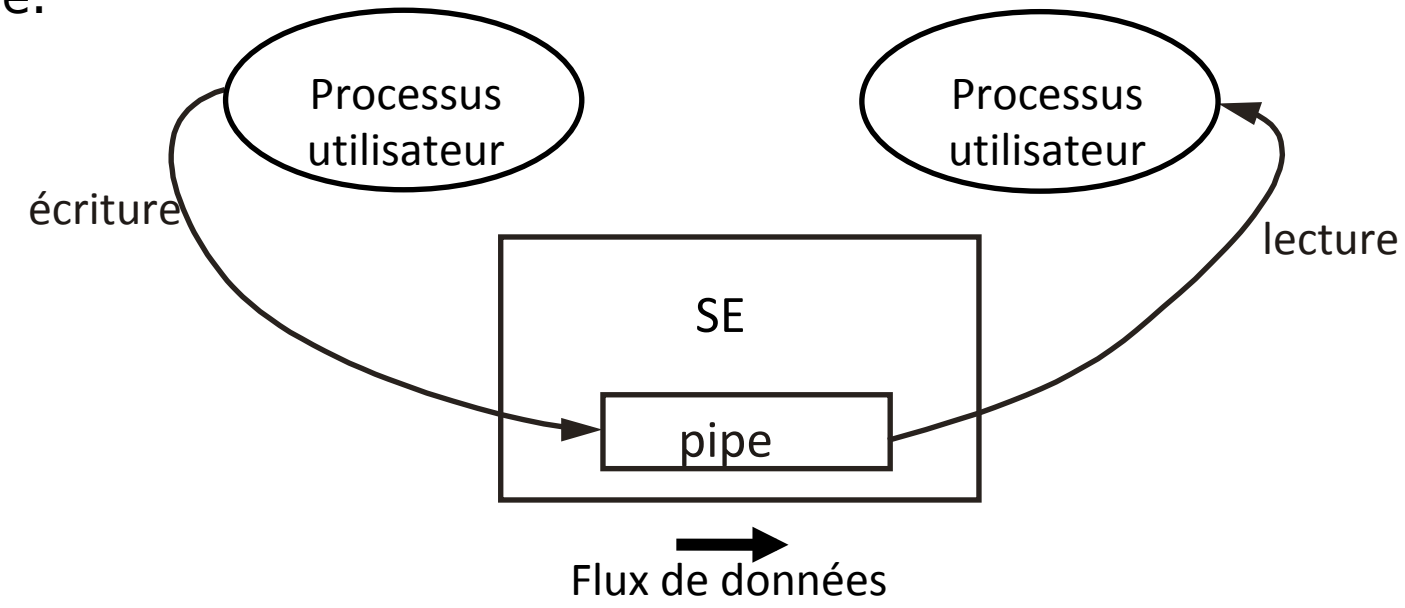
Segments de données partagés (12) : Exemple 2

```
jupiter$ gcc mmapf.c –o mmapf
jupiter$ cat fich1

Ici je parle enfin
A mon tour, en mon nom,
Au nom de mon pays,
Au nom de ma saison.
Je lui dis ma patrie
Et que c'est la rafale...
Verglas et poudrerie
Et bourrasque et froidure
Et blancheur et beauté.

jupiter$ ./mmapf fich1
jupiter$ cat fich1

 .??tuaeb te ruehcnalb tE
erudiorf te euqsarruob tE
eirerduop te salgreV
...elafar al tse'c euq tE
eirtap am sid iul eJ
.nosias am ed mon uA
,syap nom ed mon uA
,mon nom ne ,ruot nom A
nifne elrap ej icI
```

```
jupiter$ ./mmapf fich1
jupiter$ cat fich1

Ici je parle enfin
A mon tour, en mon nom,
Au nom de mon pays,
Au nom de ma saison.
Je lui dis ma patrie
Et que c'est la rafale...
Verglas et poudrerie
Et bourrasque et froidure
Et blancheur et beauté.

jupiter$
```

Les tubes de communication UNIX

- **Les tubes de communication** permettent d'établir des liaisons unidirectionnelles de communication entre processus d'une même machine.

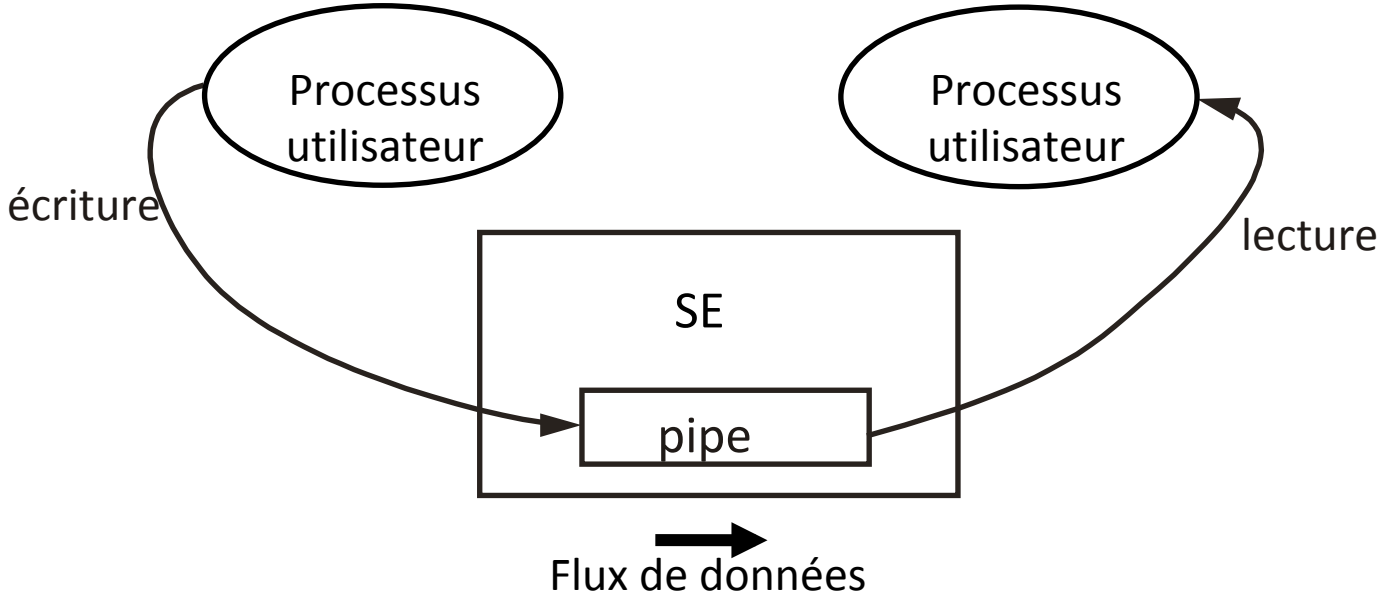


- Un tube de communication permet de mémoriser les informations produites par un ou plusieurs processus. Ces informations sont consommées par d'autres selon la discipline FIFO.

- L' opération de lecture dans un tube est destructrice : une information ne peut être lue qu'une seule fois d'un tube.

Les tubes de communication UNIX (2)

- Un tube de communication peut être vu comme **une file circulaire de taille finie**.



- On distingue deux types de tubes :
  - Les tubes anonymes (unamed pipe),
  - Les tubes nommés (named pipe) qui ont une existence dans le système de fichiers (un chemin d'accès).

Les tubes anonymes

- Un tube de communication anonyme (pipe anonyme) permet d'établir une liaison unidirectionnelle de communication **entre processus dépendants** : le créateur du tube anonyme et ses descendants.

- Il est considéré comme un fichier temporaire :
  - Lorsqu'il est créé, deux descripteurs de fichiers (de lecture et d'écriture) lui sont associés.
  - Lorsque tous les descripteurs du tube sont fermés, le tube est détruit.

- POSIX limite sa taille minimale à (**PIPE_BUF**) qui est égale à **4KiO** sous les machines Linux de l'École.

- Les tubes anonymes peuvent être créés par :
  - l'opérateur du shell « | »
  - l'appel système pipe( ). (man 2 pipe)

Les tubes anonymes (2) : Opérateur pipe « | »

- L'opérateur binaire « | » dirige la sortie standard d'un processus vers l'entrée standard d'un autre processus.

Exemple :

- La commande suivante crée deux processus reliés par un tube de communication (pipe).

who | wc –l

- Elle détermine le nombre d'utilisateurs connectés au système :
  - Le premier processus réalise la commande who.
  - Le second processus exécute la commande wc -l.

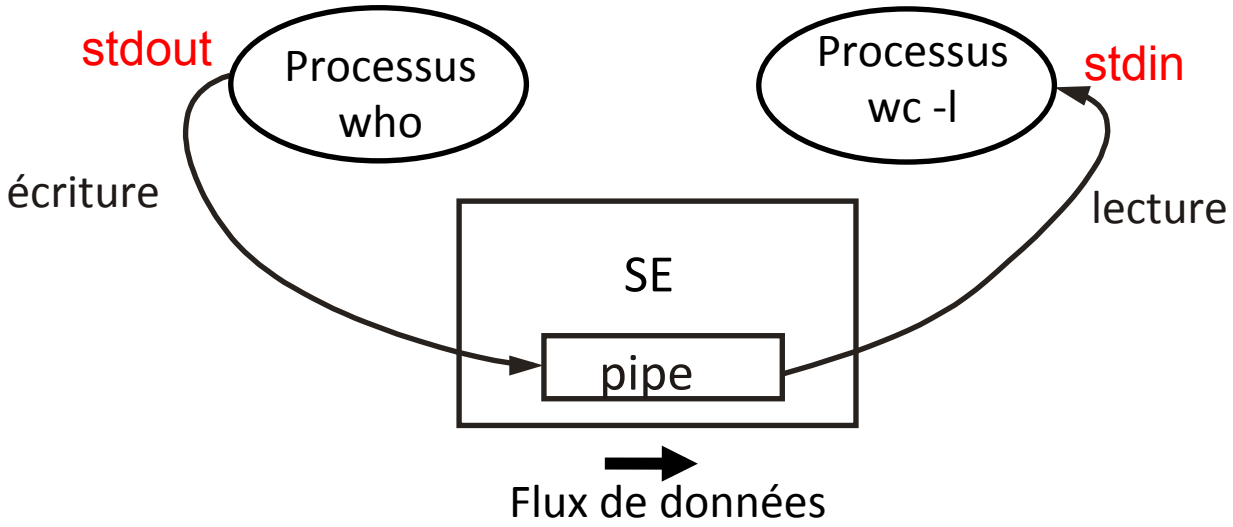
- Les résultats récupérés sur la sortie standard du premier processus sont dirigés vers l'entrée standard du deuxième processus (via le tube de communication qui les relie).

- Le processus réalisant la commande who dépose une ligne d'information par utilisateur du système sur le tube de communication.

- Le processus réalisant la commande wc –l, récupère ces lignes d'information pour en calculer le nombre total. Le résultat est affiché à l'écran.

Les tubes anonymes (3) : Opérateur pipe « | »

- Les deux processus s'exécutent en concurrence (ou en parallèle), les sorties du premier processus sont stockées dans le tube de communication.
- Si le tube est plein et le premier processus essaye d'écrire dans le tube, il sera bloqué jusqu'à ce qu'il y ait libération de l'espace nécessaire pour stocker ses données.
- De façon similaire, si le tube devient vide et le second processus tente de lire du tube, il sera bloqué jusqu'à ce qu'il y ait des données dans le tube ou une fin de fichier (eof).



Les tubes anonymes (4) : pipe

- Un tube de communication anonyme est créé par l'appel système
  int pipe(int p[2]).

- Cet appel système crée deux descripteurs de fichiers. Il retourne, dans p, les descripteurs de fichiers créés :
  - p[0] contient le descripteur réservé aux lectures à partir du tube
  - p[1] contient le descripteur réservé aux écritures dans le tube.

- Les descripteurs créés sont ajoutés à la table des descripteurs de fichiers du processus appelant.

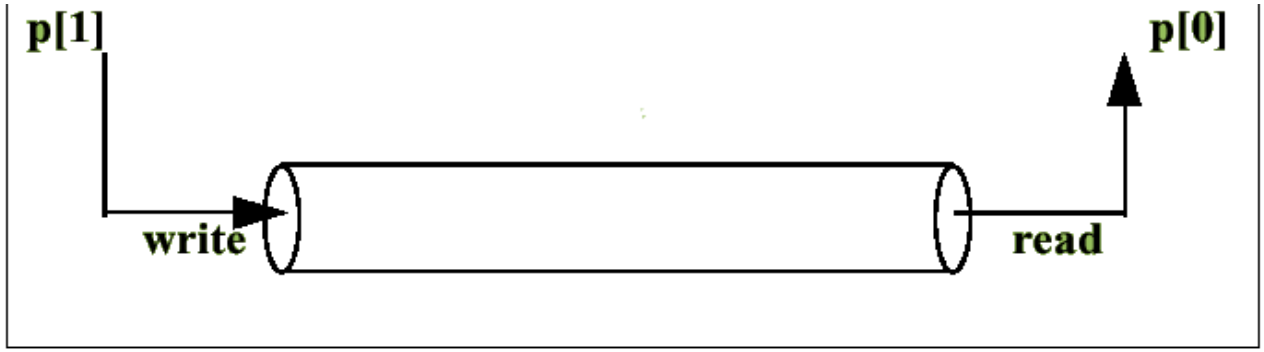
- Seul le processus créateur du tube et ses descendants peuvent accéder au tube (grâce à la duplication de la table des descripteurs de fichiers).

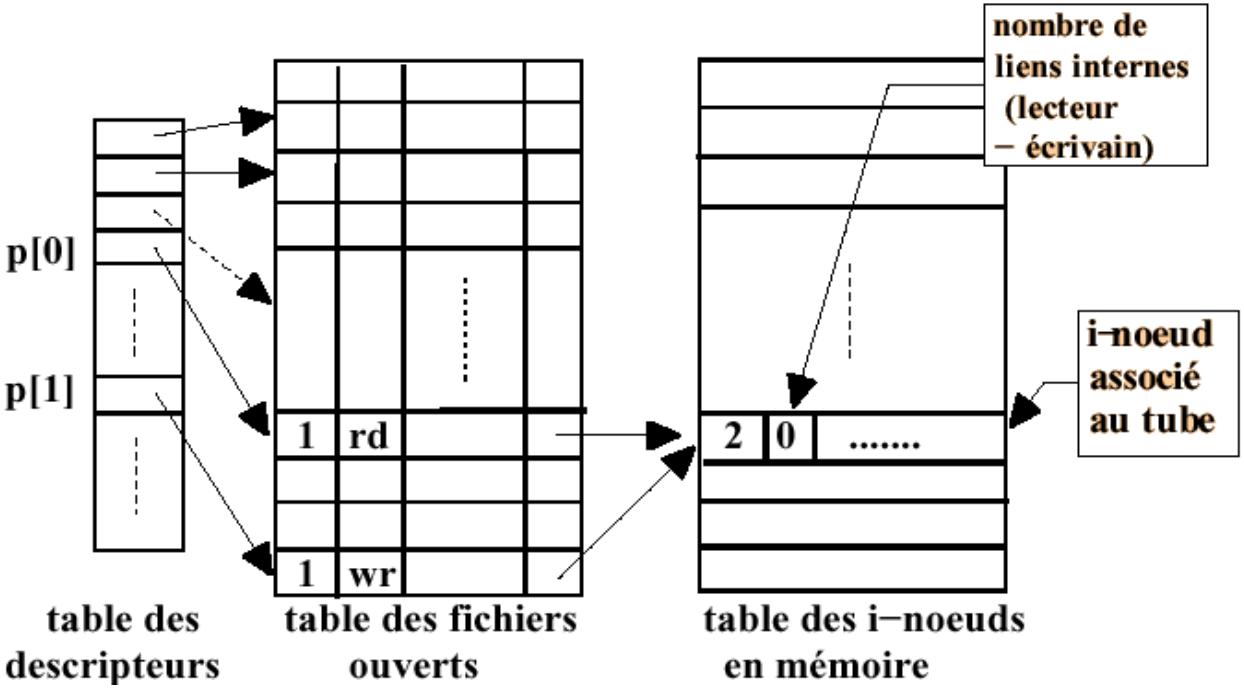
- Si le système ne peut pas créer de tube (pour, par exemple, manque d'espace) l'appel système pipe() retourne la valeur -1, sinon il retourne la valeur 0.

- L'accès au tube se fait via les descripteurs (comme pour les fichiers ordinaires).

Les tubes anonymes (5) : pipe

int p[2] ;
pipe (p) ;





Les tubes anonymes (6) : pipe

- Les tubes anonymes sont utilisés pour la communication entre un processus père et ses processus fils (ou descendants), avec au moins un processus qui écrit dans le tube, appelé processus écrivain, et un autre qui lit à partir du tube, appelé processus lecteur.

- La séquence d'événements pour une telle communication est comme suit :
  1. Le processus père crée un tube de communication anonyme en utilisant l'appel système pipe(fd) ;
  2. Le processus père crée un (ou plusieurs fils) en utilisant l'appel système fork( ) ;
  3. Le processus écrivain ferme le descripteur de fichier, non utilisé, de lecture du tube ;
  4. De même, le processus lecteur ferme le descripteur de fichier, non utilisé, d'écriture du tube ;
  5. Les processus communiquent en utilisant les appels système :

     read(fd[0], buffer, n) et write(fd[1], buffer,n);
  6. Chaque processus ferme son descripteur de fichier lorsqu'il veut mettre fin à la communication via le tube.

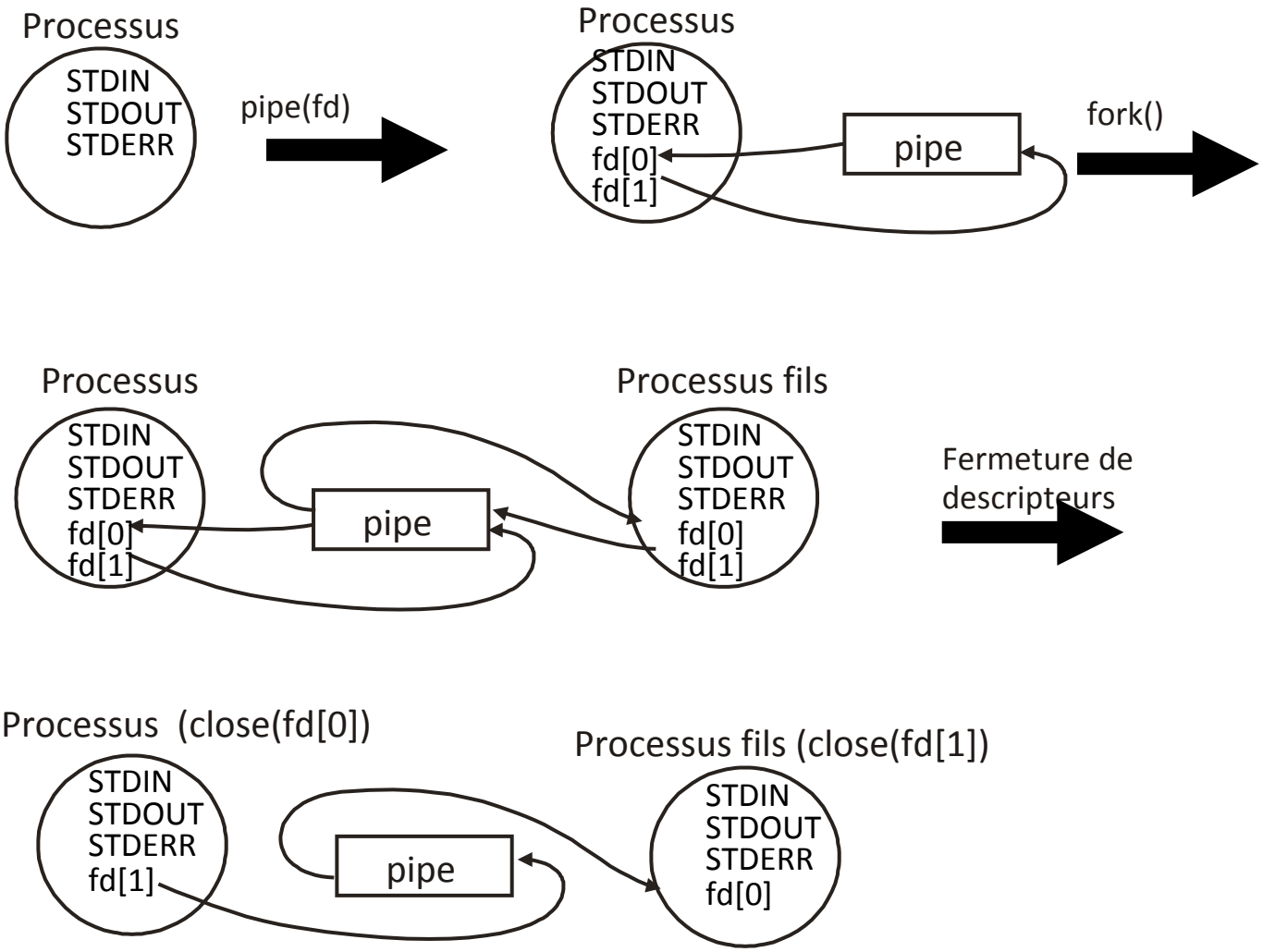
Les tubes anonymes (7) : Exemple 1 (man 2 pipe)

```
//programme testpipe5.c
#include <sys/wait.h>
 #include <assert.h>
 #include <stdio.h>
 #include <stdlib.h>
 #include <unistd.h>
 #include <string.h>

 int  main(int argc, char *argv[])
 {
    int fd[2];
    pid_t cpid;
    char buf;

    assert(argc == 2);

    if (pipe(fd) == -1) { perror("pipe"); exit(EXIT_FAILURE); }   // EXIT_FAILURE = 1
    cpid = fork();
```

Les tubes anonymes (8) : Exemple 1

```
if (cpid == -1) { perror("fork"); exit(EXIT_FAILURE); }

    if (cpid == 0) {    /* Child reads from pipe */
       close(fd[1]);         /* Close unused write end */
       while (read(fd[0], &buf, 1) > 0)
          write(STDOUT_FILENO, &buf, 1);   // write(1, &buf, 1);
       write(STDOUT_FILENO, "\n", 1);
       close(fd[0]);
       exit(EXIT_SUCCESS);  // EXIT_SUCCESS = 0

    } else {            /* Parent writes argv[1] to pipe */
       close(fd[0]);         /* Close unused read end */
       write(fd[1], argv[1], strlen(argv[1]));
       close(fd[1]);         /* Reader will see EOF */
       wait(NULL);             /* Wait for child */
       exit(EXIT_SUCCESS);
     }
}
```

Les tubes anonymes (9) : Exemple 1



Les tubes anonymes (10) : Exemple 1

```
jupiter$ gcc testpipe5.c -o testpipe5
jupiter$ ./testpipe5 "pipe   rtryu "
pipe   rtryu

jupiter$  ./testpipe5 pipe rtryu
Assertion failed: (argc == 2), function main, file testpipe.c, line 14.
Abort trap: 6

jupiter$ ./testpipe5     rtryu
rtryu

jupiter$
```

Les tubes anonymes (11) : Remarques

- Chaque tube a un **nombre de lecteurs (nombre de descripteurs en lecture) et un nombre d'écrivains (nombre de descripteurs en écriture).**

- La fonction read() d'un tube retourne 0 **(fin de fichier**), si **le tube est vide et le nombre d'écrivains est 0**.

- L'oubli de la fermeture de descripteurs peut mener à des situations d'interblocage d'un ensemble de processus.

- La fonction write() dans un tube génère **le signal SIGPIPE**, si le nombre de lecteurs est 0.

- Par défaut, **les lectures et les écritures sont bloquantes**.

Les tubes anonymes (12) : Interblocage

- On atteint une **situation d'interblocage** dans l'exemple 1, **si le processus père ne ferme pas son descripteur d'écriture dans le tube** (en mettant en commentaire ou supprimant l'instruction :

  close(fd[1]); /* Reader will see EOF */).

```
jupiter$  gcc testpipe5.c -o testpipe5
jupiter$  ./testpipe5 "pipe   rtryu" &
[1] 15019
jupiter$  pipe   rtryu
jupiter$  ps -l
F S   UID   PID  PPID  C PRI  NI ADDR WCHAN TIME CMD
0 S 11318 13399 13398  0  75   0 -  rt_sig 00:00:00 tcsh
0 S 11318 15019 13399  0  77   0 -  wait 00:00:00 testpipe5
1 S 11318 15020 15019  0  78   0 -  pipe_w 00:00:00 testpipe5
```

Les tubes anonymes (13) : Redirection de stdin et stdout

- Les fonctions dup et dup2 permettent à un processus de créer un nouveau descripteur (dans sa table des descripteurs de fichiers) « **synonyme** d'un descripteur déjà existant ».

  #include <unistd.h>
  int dup (int *desc*);

  dup crée et retourne un descripteur synonyme à desc. Le numéro associé au descripteur créé est le plus petit descripteur disponible dans la table des descripteurs de fichiers du processus.

  #include <unistd.h>
  int dup2(int desc1, int desc2);

  dup2 transforme desc2 en un descripteur synonyme de desc1.

- Ces fonctions peuvent être utilisées pour réaliser des redirections de fichiers (entrée standard, sortie standard, sortie erreur, etc.) vers les tubes de communication ou fichiers.

Les tubes anonymes (14) : Exemple 2

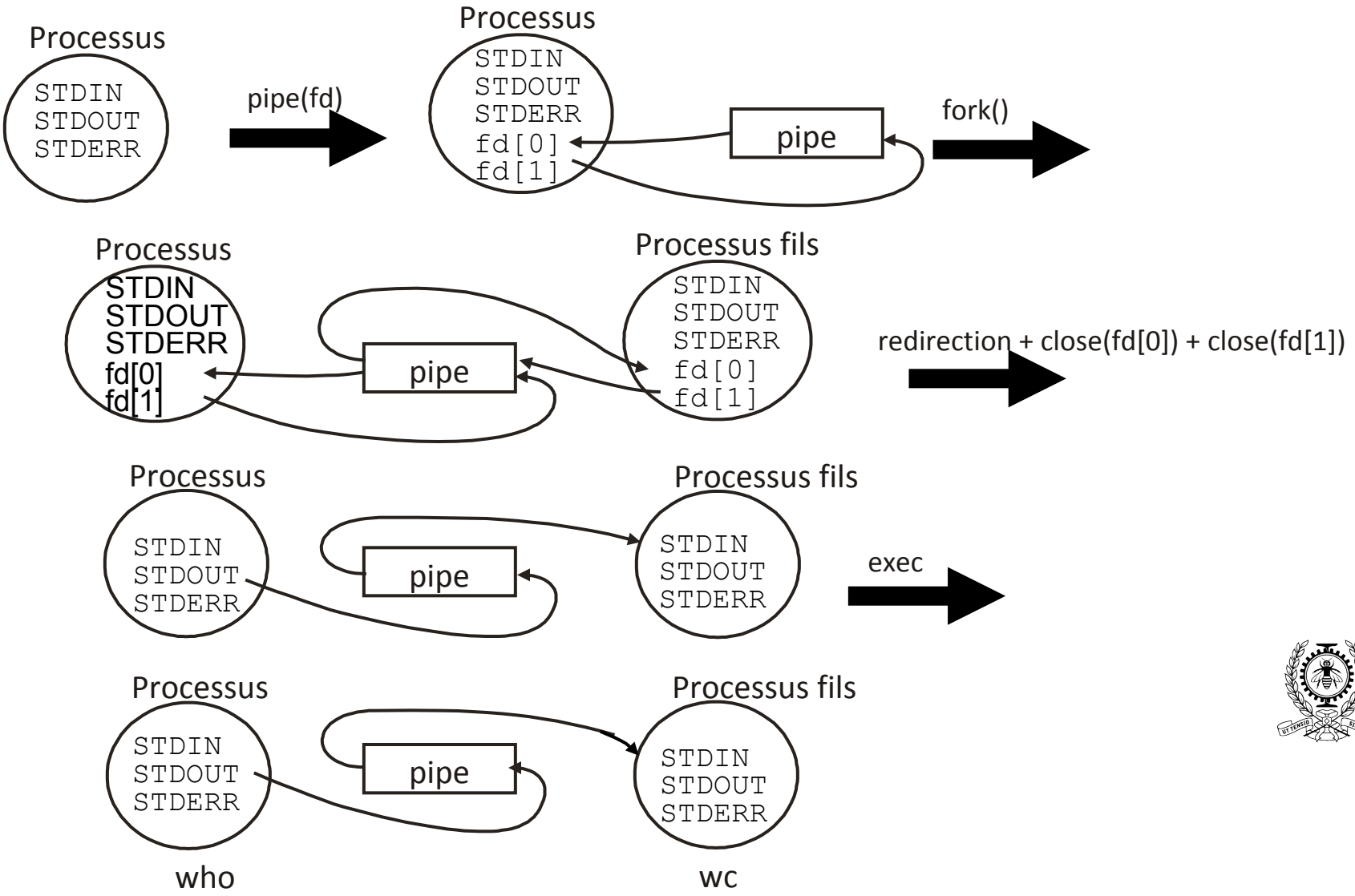
- Ce programme réalise l'exécution en concurrence/parallèle de deux commandes shell. Un tube connecte stdin de la 1ière vers stdout de la 2ième.

```
//programme pipecom.c
#include <unistd.h>     //pour fork, close…
#include <stdio.h>
#define R 0
#define W 1
int main ()
{  int fd[2] ;
   pipe(fd) ;                  // creation d'un tube sans nom
   if (fork() !=0)
   {   close(fd[R]); //Le père ferme le descripteur de lecture
       dup2(fd[W], 1) ; // copie fd[W] dans le descripteur 1
       close (fd[W]) ; // fermeture du descripteur d'écriture
      if(execlp(argv[1], argv[1], NULL) ==-1) // exécute l'écrivain
      perror("error dans execlp") ;
      }
```

Les tubes anonymes (15) : Exemple 2

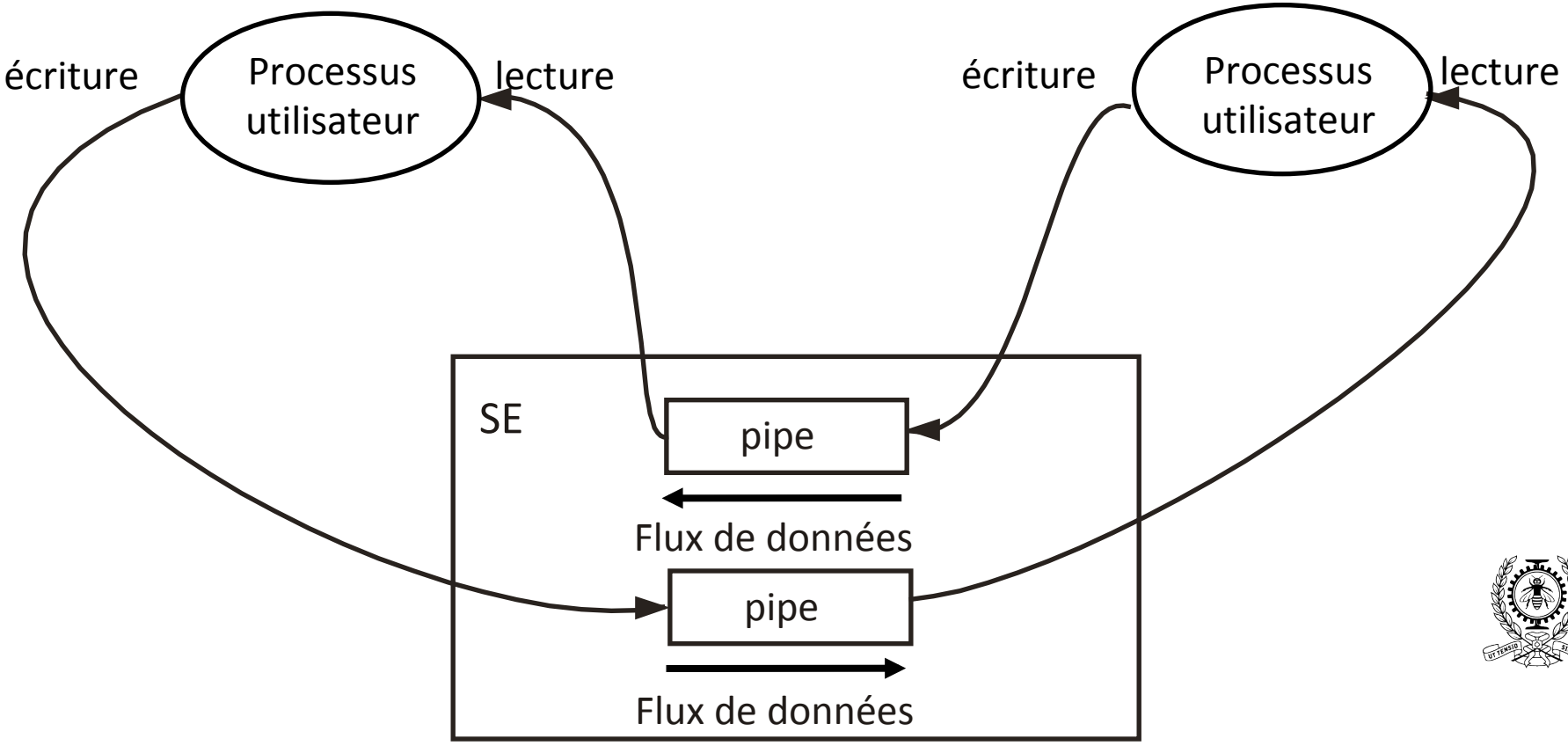
```
    else  // processus fils (lecteur)
    {     // fermeture du descripteur non utilisé d'écriture
          close(fd[W]) ;
          // copie fd[R] dans le descripteur 0
          dup2(fd[R],0) ;
          close (fd[R]) ; // fermeture du descripteur de lecture
          // exécute le programme lecteur
          execlp(argv[2], argv[2], NULL) ;
          perror("connect") ;
    }
    return 0 ;
}

// fin du programme pipecom.c
jupiter% gcc -o pipecom pipecom.c
jupiter% pipecom who wc
      9     54     489
```

Les tubes anonymes (16) : Exemple 2



Les tubes anonymes (17) : Communication bidirectionnelle

- La communication bidirectionnelle est possible en utilisant deux tubes (un pour chaque sens de communication).



Les tubes nommés

- Les tubes de communication nommés fonctionnent aussi comme des files de discipline FIFO (first-in-first-out) avec des lectures destructrices.
- Ils sont plus polyvalents que les tubes anonymes car ils offrent, en plus, les avantages suivants :
  - Ils ont chacun un nom qui existe dans le système de fichiers et sont considérés comme des fichiers spéciaux ;
  - Ils peuvent être utilisés par des processus indépendants, à condition qu'ils s'exécutent sur une même machine.
  - Ils existeront jusqu'à ce qu'ils soient supprimés explicitement (unlink);
  - Leur taille est plus grande que celle des tubes anonymes.
  - Ils sont créés par la commande **« mkfifo» ou « mknod »** ou par l'appel système **mknod() ou mkfifo()**.

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

Les tubes nommés (2) : Commande mkfifo

jupiter% mkfifo mypipe

Affichage des attributs du tube créé

```
jupiter% ls -l mypipe
prw-------   1 usname grname       0 sep  12 11:10 mypipe
```

Modification des permissions d'accès

```
jupiter% chmod g+rw mypipe
jupiter% ls -l mypipe
prw-rw----   1 usname grname       0 sep  12 11:12 mypipe
```

Remarque : p indique que c'est un tube.

- Une fois le tube créé, il peut être utilisé pour réaliser la communication entre deux processus.
- Chacun des deux processus ouvre le tube, l'un en mode écriture et l'autre en mode lecture.

Les tubes nommés (3) : Exemple 3

```
// programme writer.c envoie un message sur le tube mypipe
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
int main()
{   int fd;
        char message[100];
        sprintf(message, "bonjour du writer [%d]", getpid());

        //Ouverture du tube  mypipe en mode écriture
        fd = open("mypipe", O_WRONLY);
        printf("ici writer[%d] \n", getpid());
        if (fd!=-1)
        {       // Dépot d'un message dans le tube
                write(fd, message, strlen(message)+1);
                close(fd);
        } else
             printf( " désolé, le tube n'est pas disponible \n");
    return 0;
}
```

Les tubes nommés (4) : Exemple 3

```
// programme reader.c lit un message à partir du tube mypipe
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
int main()
{   int fd,n;
        char message[100];
    // ouverture du tube mypipe en mode lecture
        fd = open("mypipe", O_RDONLY);
        printf("ici reader[%d] \n",getpid());
        if (fd!=-1)
        { // récupérer un message du tube, taille maximale est 100.
                while ((n = read(fd,message,100))>0)
                 // n est le nombre de caractères lus
                       printf("%s\n", message);
                close(fd);
        } else  printf( "désolé, le tube n'est pas disponible\n");
    return 0;
}
```

Les tubes nommés (5) : Exemple 3

- Après avoir compilé séparément les deux programmes, il est possible de lancer leurs exécutions en arrière plan.

- Les processus ainsi créés communiquent via le tube de communication mypipe.

```
jupiter% gcc -o writer     writer.c
jupiter% gcc -o reader     reader.c
```

- Lancement de l'exécution d'un writer et d'un reader:

```
jupiter% writer&    reader&
[1] 1156
[2] 1157
ici writer[1156]
ici reader[1157]
bonjour du writer [1156]
[2]    Done                 reader
[1]  + Done                 writer
```

Les tubes nommés (6) : Exemple 3

- Lancement de l'exécution de deux writers et d'un reader.

```
jupiter% writer& writer& reader&
[1] 1196
[2] 1197
[3] 1198
ici writer[1196]
ici writer[1197]
ici reader[1198]
bonjour du writer [1196]
bonjour du writer [1197]
[3]    Done                 reader
[2]  + Done                 writer
[1]  + Done                 writer
```

Les tubes nommés (7) : Remarques

- Par défaut, l'ouverture d'un tube nommé est bloquante (sinon, spécifier l'option O_NONBLOCK).

- Si un processus ouvre un tube nommé en lecture(resp. écriture)

-> le processus sera bloqué jusqu'à ce qu'un autre processus ouvre le tube en écriture (resp. lecture).

- Attention aux situations d'interblocage

```
/* processus 1 */                    /* processus 2 */

int f1, f2;                          int f1, f2;
...                                  ....
f1 =open("fifo1", O_WRONLY);         f2=open("fifo2", O_WRONLY);
f2 =open("fifo2", O_RDONLY);         f1=open("fifo1", O_RDONLY);
...                                  ...
```

Les signaux

- Tout système d'exploitation de la famille UNIX gère un ensemble de signaux (une sorte d'interruptions logicielles).

- Chaque signal :
  - a un nom,
  - a un numéro,
  - a un gestionnaire (handler) et
  - est, en général, associé à un événement.

- La commande « man 7 signal » ou « man signal » permet de lister tous les signaux gérés par le système (nom, numéro, effet de son traitement (son gestionnaire), événement associé, etc.) :

  SIGINT 2, SIGQUIT 3, SIGKILL 9, SIGPIPE 13, SIGALRM 14, SIGUSR1 30,10,16, SIGUSR2 31,12,17, SIGCHLD 20,17,18, SIGCONT 19,18,25, SIGSTOP 17,19,23, etc.

Les signaux (2)

- Les gestionnaires associés par le système d'exploitation aux signaux :
  - abort (génération d'un fichier *core* et terminaison du processus),
  - terminaison (sans génération d'un fichier core),
  - ignore (le signal est ignoré),
  - stop (suspension de l'exécution du processus destinataire), ou
  - continue (reprendre l'exécution si le processus destinataire est suspendu sinon le signal est ignoré).
- Ces gestionnaires sont appelés gestionnaires par défaut des signaux.
- Par exemple, le gestionnaire par défaut de SIGUSR1 et SIGUSR2 est la terminaison, celui de SIGCHLD est « ignore ».
- Les signaux permettent aux processus de réagir aux événements sans être obligés de tester en permanence leurs arrivés.

Les signaux (3)

- De **nombreuses erreurs détectées par le matériel** comme l'exécution d'une instruction non autorisée (division par 0) ou l'emploi d'une adresse non valide, **sont converties en signaux qui sont envoyés au processus fautif**.

- Lorsqu'un processus se termine, un signal SIGCHLD est envoyé à son père.

- Lorsqu'un processus essaye d'écrire dans un tube rompu (sans lecture), un signal SIGPIPE lui est envoyé.

- De façon simplifiée, lorsqu'un signal est envoyé à un processus, le système interrompra (dès que possible) l'exécution du processus pour lui permettre de réagir au signal (exécuter le gestionnaire du signal) avant de poursuivre (éventuellement) son exécution.

➔ Un signal est une sorte d'interruption logicielle asynchrone qui a pour but d'informer de l'arrivée d'un événement (outil de base de notification d'évènements). Il ne véhicule pas d'information.

Les signaux (4) : Comment envoyer un signal à un processus ?

- L'appel système qui permet d'envoyer un signal à un ou plusieurs processus, kill (man 2 kill)

  ```
  #include <sys/types.h>
  #include <signal.h>
  int kill ( pid_t pid, int sig);
  ```

  Si pid >0, le signal sig est envoyé au processus pid, si pid = 0, le signal est envoyé à tous les processus du groupe de l'appelant. Il retourne 0 en cas de succès et -1 en cas d'erreur.

- Dans le cas du système UNIX, un processus utilisateur peut envoyer un signal à un autre processus. Les deux processus doivent appartenir au même propriétaire ou le processus émetteur du signal est un super-utilisateur.

- La commande kill permet aussi d'envoyer un signal à un ou plusieurs processus (man kill).

Les signaux (5) : Réception et traitement d'un signal

- La réception d'un signal est matérialisée par un bit positionné à 1 dans un tableau associé au processus (tableau des signaux en attente) → risque de perte de signaux.
- Un processus peut différer le traitement d'un signal reçu. Chaque processus a un masque des signaux qui indique les signaux à bloquer (ceux dont le traitement est à différer).
- Le système vérifie si un processus a reçu un signal non bloqué aux transitions suivantes: passage du mode noyau à utilisateur, avant de passer à l'état bloqué, ou en sortant de l'état bloqué.
- Si c'est le cas, le signal reçu est traité en exécutant son gestionnaire.
- Si le traitement du signal ne termine pas le processus, après ce traitement, le processus reprendra le code interrompu à l'instruction qui suit celle exécutée juste avant le gestionnaire.
- Le traitement des signaux reçus se fait dans le contexte d'exécution du processus.

Les signaux (6) : Peut-on redéfinir le gestionnaire d'un signal ?

- Le système d'exploitation permet à un processus de redéfinir pour certains signaux leur gestionnaire (remplacer le traitement par défaut par un autre).

- Les signaux SIGKILL et SIGSTOP ne peuvent être ni ignorés, ni bloqués. On ne peut pas non plus redéfinir leurs gestionnaires par défaut (les capturer).

- Par exemple, la touche d'interruption Ctrl+C génère un signal SIGINT qui est envoyé a tous les processus rattachés au terminal. Par défaut, ce signal arrête le processus. Le processus peut associer à ce signal un autre gestionnaire.

Les signaux (7) :
Comment redéfinir le gestionnaire d'un signal ?

- Les fonctions signal et sigaction permettent de redéfinir le gestionnaire d'un signal.

- La fonction signal(3) du langage C (non fiable ← Différentes implémentations) :

```
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal ( int signum,   sighandler_t handler );
```

  - Le premier paramètre est le numéro ou le nom du signal à capturer
  - Le second est la fonction (le gestionnaire) à exécuter à l'arrivée du signal (ou SIG_DFL, l'action par défaut, ou SIG_IGN pour ignorer)
  - signal retourne le gestionnaire précédent ou SIG_ERR en cas d'erreur.

Les signaux (8) :
Comment redéfinir le gestionnaire d'un signal ?

#include <signal.h>

int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact) ;

- La structure sigaction :
  - **void (*sa_handler)(int)** ; /* le gestionnaire */
  - **sigset_t sa_mask** ; /* le masque : les signaux à bloquer durant l'exécution du gestionnaire*/
  - int sa_flags ; /* options */ . . .
- On peut associer un même gestionnaire à des signaux différents.

Les signaux (9) : Attente d'un signal

- L'appel système pause() bloque l'appelant jusqu'au prochain signal.

```
#include <unistd.h>
int pause (void);
```

- L'appel système sigsuspend(mask) remplace le masque de signaux du processus appelant avec le masque fourni dans mask et suspend le processus jusqu`au prochain signal.

```
#include <signal.h>
int sigsuspend(const sigset_t *mask);
```

- L'appel système sleep(v) suspend l'appelant jusqu'au prochain signal ou l'expiration du délai (v secondes).

```
#include <unistd.h>
void sleep  (int );
```

Les signaux (10) : Exemple 4 (signal SIGINT)

```
// signaux0.c
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int count = 0;
static void action(int sig)
{   ++count ;
    write(1,"capture du signal SIGINT\n", 26) ;
}
int main()
{   // Spécification de l'action du signal
    signal (SIGINT, action);
    printf("Debut:\n");
    do  {
          sleep(1);
        } while (count <3);
    return 0;
}
```

```
d5333-09>  gcc signaux0.c -o signaux0
d5333-09> signaux0
Debut:
capture du signal SIGINT
capture du signal SIGINT
capture du signal SIGINT
```

Les signaux (11) : Exemple 5 (SIGTERM)

```
// test_signaux.c
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
static void action(int sig)
{
    printf("On peut maintemant m'eliminer\n");
    signal(SIGTERM, SIG_DFL);
}

int main()
{
    if( signal(SIGTERM, SIG_IGN) == SIG_ERR)
        perror("Erreur de traitement du code de l'action\n");
    if( signal(SIGUSR2, action) == SIG_ERR)
        perror("Erreur de traitement du code de l'action\n");
    while (1)
        pause();
}
```

Les signaux (12) : Exemple 5 (SIGTERM)

```
bash-2.05b$ gcc -o test-signaux test-signaux.c
bash-2.05b$ ./test-signaux &
[1] 4664
bash-2.05b$ ps
PID TTY TIME CMD
4664 pts/2 00:00:00 test-signaux
bash-2.05b$ kill -SIGTERM 4664
bash-2.05b$ ps
PID TTY TIME CMD
4664 pts/2 00:00:00 test-signaux
bash-2.05b$ kill -SIGUSR2 4664
bash-2.05b$ On peut maintenant m'eliminer
bash-2.05b$ ps
PID TTY TIME CMD
4664 pts/2 00:00:00 test-signaux
bash-2.05b$ kill -SIGTERM 4664
bash-2.05b$ ps
PID TTY TIME CMD
4668 pts/2 00:00:00 ps
[1]+ Terminated ./test-signaux
bash-2.05b$
```

Les signaux (13) : Exemple 6 (échange de signaux)

```
// signaux1.c
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
static void action(int sig)
{
    switch (sig)
    {
        case SIGUSR1: printf("Signal SIGUSR1 reçu\n");
                      break;
        case SIGUSR2: printf("Signal SIGUSR2 reçu\n");
                      break;
        default:      break;
    }
}
```

Les signaux (14) : Exemple 6 (échange de signaux)

```
int main()
{   struct sigaction new_action, old_action;
     int i, pid, etat;


    new_action.sa_handler = action;
    sigemptyset (&new_action.sa_mask);
    new_action.sa_flags = 0;

    if( sigaction(SIGUSR1, &new_action,NULL) <0)
         perror("Erreur de traitement du code de l'action\n");

    if( sigaction(SIGUSR2, &new_action,NULL) <0)
         perror("Erreur de traitement du code de l'action\n");
```

Les signaux (15) : Exemple 6 (échange de signaux)

```
-bash-3.2$ gcc signaux1.c -o signaux1
-bash-3.2$ ./signaux1
Parent : terminaison du fils
Parent : fils a termine  13094 : 1 : 15 : 15
```

```
   if((pid = fork()) == 0){
      kill(getppid(), SIGUSR1);

      for(;;)     pause(); // attendre à répétition un signal

   }else {

      kill(pid, SIGUSR2); // Envoyer un signal à l'enfant
      printf("Parent : terminaison du fils\n");
      kill(pid, SIGTERM); // Signal de terminaison à l'enfant
      pid = wait(&etat); // attendre la fin de l'enfant
      printf("Parent: fils a termine %d : %d : %d : %d\n",
          pid, WIFSIGNALED(etat), WTERMSIG(etat), SIGTERM);
   }
return 0;
}
```

Le père envoie SIGUSR2 et SIGTERM puis se met en attente de son fils.
Le fils est tué par le signal SIGTERM

Les signaux (16) : Exemple 6’ (échange de signaux)

```
   if((pid = fork()) == 0){
      pause();
      kill(getppid(), SIGUSR1);          // Envoyer signal au parent.
      for(;;)   pause();                 // attendre à répétition un signal
   } else {
      kill(pid, SIGUSR2);                // Envoyer un signal à l'enfant
      pause();
      printf("Parent : terminaison du fils\n");
      kill(pid, SIGTERM); // Signal de terminaison à l'enfant
      wait(&etat); // attendre la fin de l'enfant
      printf("Parent : fils a terminé\n");
   } return0;
}
```

```
-bash-3.2$ ./signaux1&
[1] 26602
Signal SIGUSR2 reçu
ps –l
F S   UID   PID       PPID  C  PRI  NI ADDR SZ WCHAN  TTY        TIME CMD
0 S 11318 18080 18079  0  80   0 -  1340 wait   pts/1    00:00:00 bash
0 S 11318 26602 18080  0  80   0 -   400 pause  pts/1    00:00:00 signaux1
0 R 11318 26603 18080  0  80   0 -  1173 -      pts/1    00:00:00 ps
1 S 11318 26604 26602  0  80   0 -   401 pause  pts/1    00:00:00 signaux1
```

Les signaux (17) : Masquage d'un signal

- L'appel système sigprocmask permet de modifier le masque des signaux (bloquer (masquer) ou débloquer un ensemble de signaux).

```
#include <signal.h>
int sigprocmask(   int how,   // SIG_BLOCK,  SIG_UNBLOCK ou SIG_SETMASK
    const sigset_t * set,
    sigset_t*        oldset     // oldset reçoit l'ensemble des signaux bloqués avant
                                   d'effectuer l'action indiquée par how
    );
```

SIG_BLOCK : pour ajouter les signaux de **set** à l'ensemble des signaux bloqués.

SIG_UNBLOCK : pour enlever les signaux de **set** de l'ensemble des signaux bloqués.

SIG_SETMASK : pour remplacer l'ensemble des signaux bloqués par **set**.

- Lorsqu'un signal bloqué (figure dans le masque) est émis, il est mis en attente jusqu'à ce qu'il devienne non bloqué (retiré du masque).
- L'appel système sigpending permet de récupérer les signaux en attente.

  int sigpending (sigset_t *set);

Lectures suggérées

- Les signaux sous Linux (pp 221-230)

Patrick Cegielski "Conception de systèmes d'exploitation - Le cas Linux", 2nd edition Eyrolles, 2003. **Livre disponible dans le dossier Slides Automne 2018 du site moodle du cours.**

- Communication par tubes sous Linux (pp 505-515)

Patrick Cegielski "Conception de systèmes d'exploitation - Le cas Linux", 2nd edition Eyrolles, 2003. **Livre disponible dans le dossier Slides Automne 2018 du site moodle du cours.**

Les tubes anonymes : Transfert de descripteurs de fichiers via exec

Annexe

```
//programme sprintf.cpp
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
int main()
{  int fd[2];
  pipe(fd);
if (fork()==0)
  {      char chaine[10];
        close(fd[0]); sprintf(chaine, "%d\n",
      fd[1]);
        execl("./filssprintf", "./filssprintf", chaine,
      NULL);
  }  else {    printf("ici père : fd[1] = %d\n",fd[1]);
         close(fd[1]);     // lire du pipe
         wait(NULL); close(fd[0]);
         printf("le père se termine\n");
   }
return 0;
}
```

```
// programme filssprintf.cpp
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main(int argc, char*argv[])
{   int x = atoi(argv[1]);
  printf ("ici fils %d \n", x);
 // écrire dans le pipe
  close(x);
  return 0;
}
```

```
-bash-3.2$ g++ filssprintf.cpp -o flissprintf
-bash-3.2$ g++ sprintf.cpp -o sprintf
-bash-3.2$ ./sprintf
ici père : fd[1] = 4
ici fils 4
le père se termine
-bash-3.2$
```

Les tubes anonymes : Taille

Annexe

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
int main() {
  printf("%d\n",PIPE_BUF);
 return 0;
}
```

```
jupiter$ gcc pipebuf.cpp -o pipebuf
jupiter$ ./pipebuf
4096
```

Segments de données partagés entre père et fils Annexe

```
#include <fcntl.h>
 #include <stdio.h>
 #include <stdlib.h>
 #include <unistd.h>
 #include <sys/stat.h>
 #include <sys/mman.h>
#include<sys/wait.h>
 int main(int argc, char *argv[]) {
 char   *projection;
 int    fd;
 struct stat attr;
 long   fsize;
 int    i;
 char   tmp;
 if(argc != 2)  exit(1);

fd = open(argv[1],O_RDWR);
 if(fd==-1) exit(1);
 stat(argv[1], &attr);
```

Segments de données partagés entre père et fils

Annexe

```
fsize = attr.st_size;
projection = (char *) mmap(NULL, fsize, PROT_READ |
                                    PROT_WRITE, MAP_SHARED, fd, 0);
if(projection == (char *) MAP_FAILED)        exit(1);
close(fd);
if (fork()==0) // le fils
{   for(i=0; i<fsize/2; i++) {
        tmp = projection[i];
        projection[i] = projection[fsize - i -1];
        projection[fsize - i -1] = tmp;
     }
    munmap((void *) projection, fsize);
    printf("fin du fils \n");
    return 0;
}  // le père
wait(NULL);
for(i=0; i<fsize; i++) {
   write(1, &projection[i],1);}
return 0;
}
```

Segments de données partagés entre père et fils

Annexe

```
jupiter$ gcc mmap12.c -o mmap12
jupiter$ cat fich1

12345678
91011121314
jupiter$ ./mmap12 fich1
fini du fils

41312111019
87654321
jupiter$ cat fich1

41312111019
87654321
jupiter$
```