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

Automne 2017

Chapitre 4 - Communication Interprocessus

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

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 s'échanger des informations (communication interprocessus).
- Il existe plusieurs mécanismes de communication interprocessus :
  - les données communes (variables, fichiers, segments de données),
  - les signaux,et
  - les messages.

Introduction (2)

- Les threads (POSIX) d'un processus partagent la zone de données globale, le tas, le code, la table des descripteurs de fichiers du processus, etc.

- Il est possible aussi de créer dynamiquement des segments de données et de les rattacher aux espaces d'adressage de plusieurs processus.

- 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.

Les tubes de communication UNIX

- Il existe plusieurs mécanismes de communication par envoi de messages :
  - les tubes de communication,
  - les files de messages
  - les sockets….

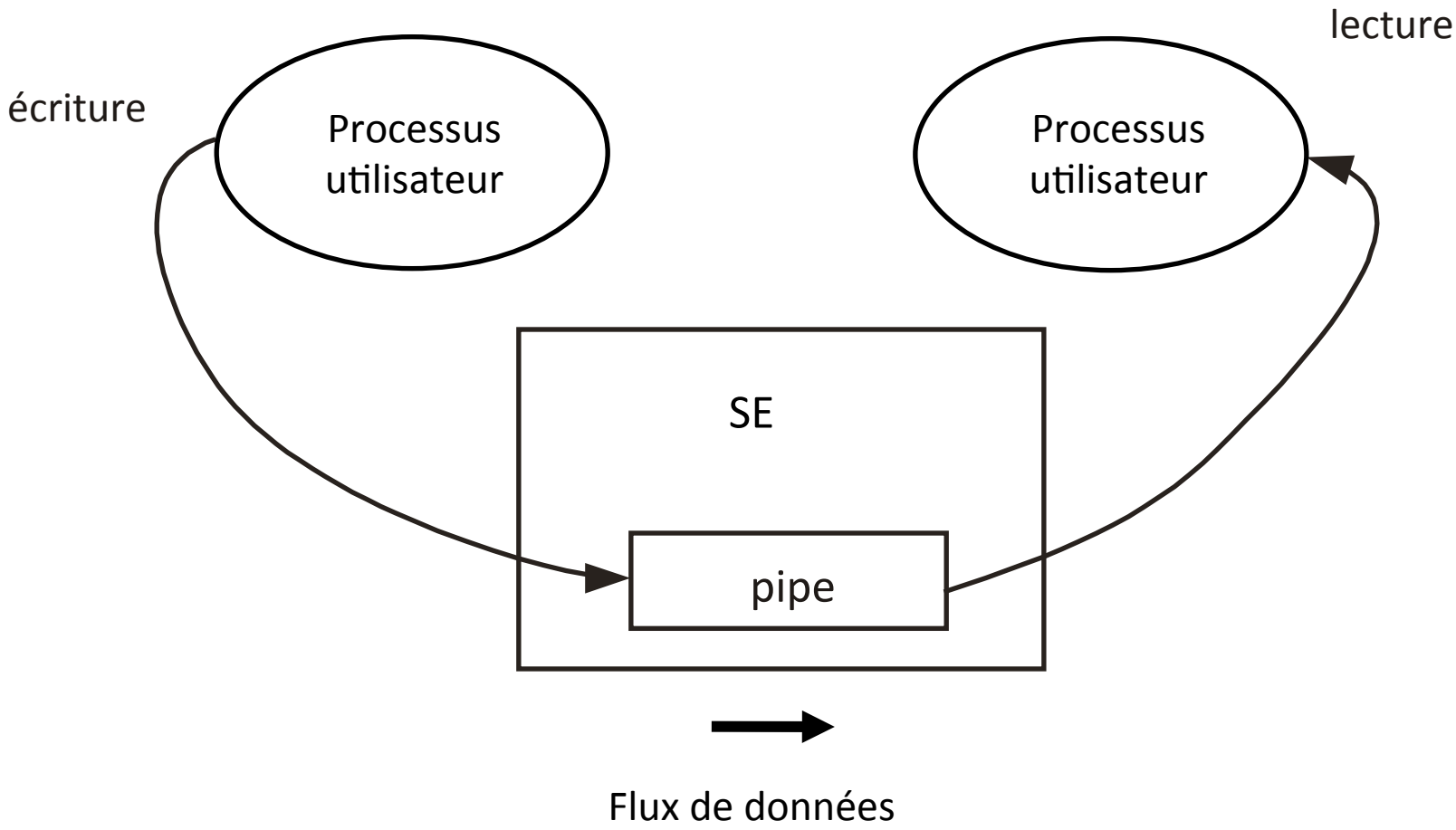
- Les tubes de communication permettent à deux ou plusieurs processus s'exécutant sur une même machine d'échanger des informations ➔ Communication locale.

- 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

- Les tubes anonymes (pipes) peuvent être considérés comme des fichiers temporaires.
- Ils permettent d'établir des liaisons unidirectionnelles de communication entre processus dépendants (le créateur d'un tube anonyme et ses descendants).
- Un tube de communication permet de mémoriser des informations et se comporte comme une file FIFO.
- Il est caractérisé par deux descripteurs de fichiers (lecture et écriture) et sa taille limitée (PIPE_BUF) est approximativement égale à 4KiO.
- L' opération de lecture dans un tube est destructrice : une information ne peut être lue qu'une seule fois d'un tube.
- Lorsque tous les descripteurs du tube sont fermés, le tube est détruit.
- Les tubes anonymes peuvent être créés par :
  - l'opérateur du shell « | »
  - l'appel système pipe( ). (man 2 pipe)

Les tubes anonymes (2)



Les tubes anonymes (3) : 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 d'information.

- 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 (4) : Opérateur pipe « | »

- Les deux processus s'exécutent 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 les données.
- De façon similaire, si le tube devient vide etle 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 (5) : 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.

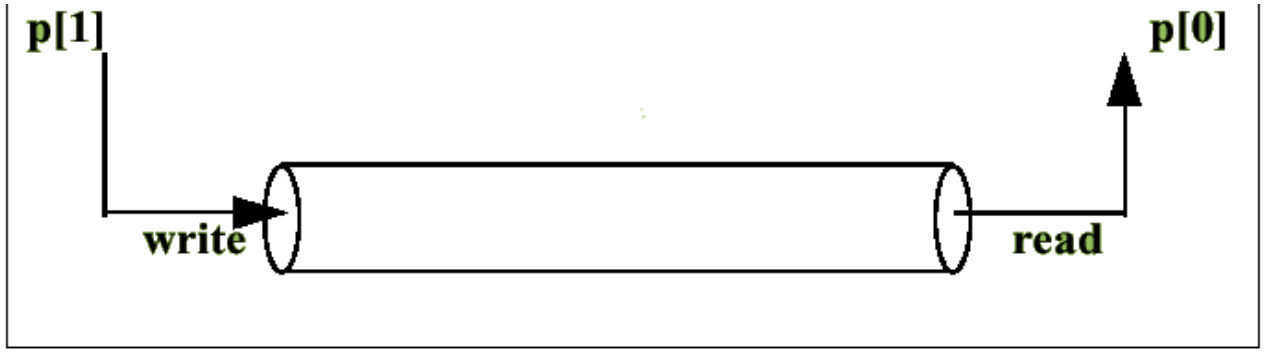
- Seul le processus créateur du tube et ses descendants (ses fils) peuvent accéder au tube (duplication de la table des descripteurs de fichiers).

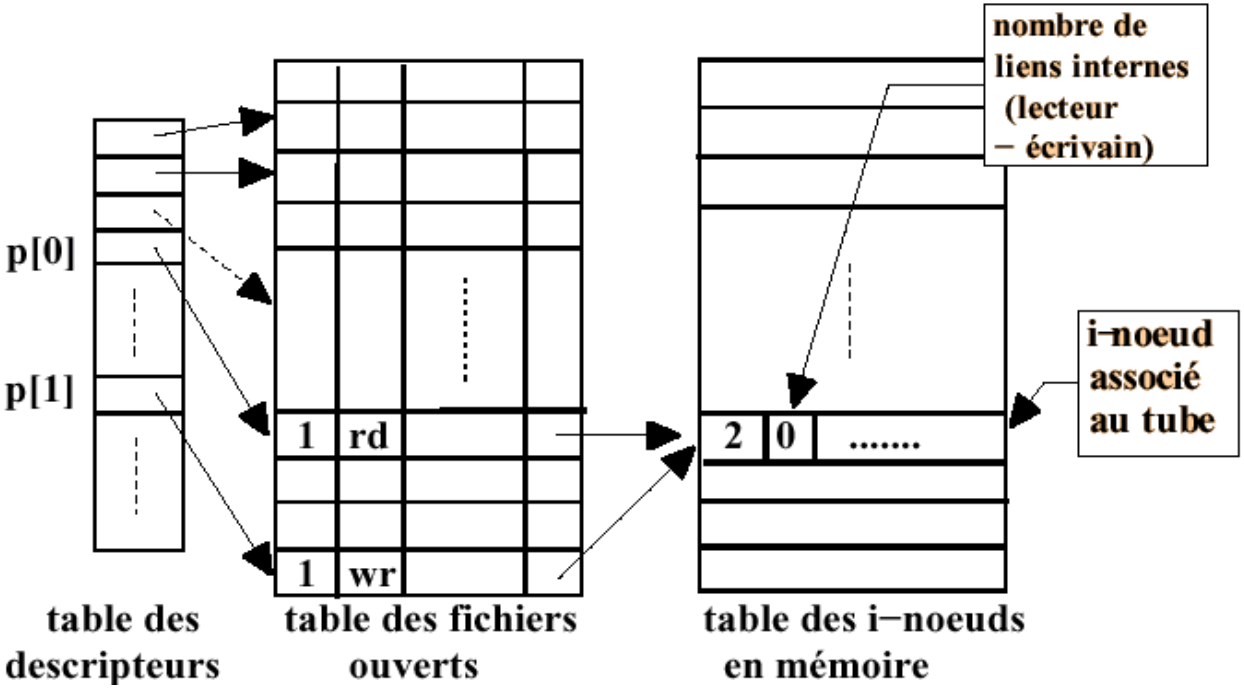
- Si le système ne peut pas créer de tube pour manque d'espace, l'appel système pipe() retourne la valeur -1, sinon il retourne la valeur 0.

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

Les tubes anonymes (6) : pipe

int p[2] ;
pipe (p) ;





Les tubes anonymes (7) : pipe

- Les tubes anonymes sont utilisés pour la communication entre un processus père et ses processus fils, avec un processus qui écrit sur 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( ) ;
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 fichier lorsqu'il veut mettre fin à la communication via le tube.

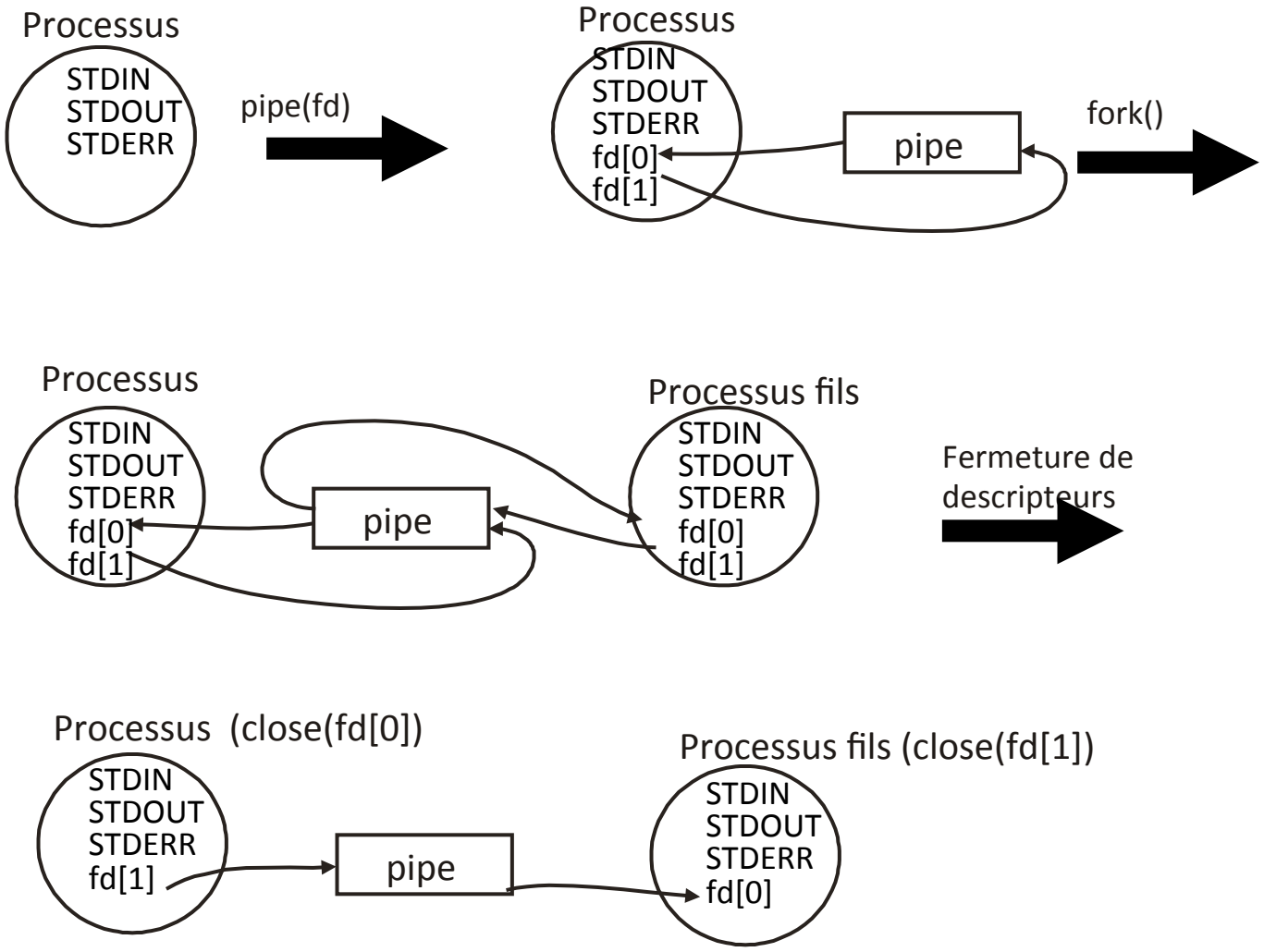
Les tubes anonymes (8) : Exemple 1 (man 2 pipe)

```
//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 (9) : 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 (10) : Exemple 1



Les tubes anonymes (11) : Exemple 1

```
> gcc testpipe5.c -o testpipe5

> ./testpipe5 "pipe   rtryu "
pipe   rtryu


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


> ./testpipe5     rtryu
rtryu
```

Les tubes anonymes (12) : Remarques

- Chaque tube a un **nombre de lecteurs et un nombre d'écrivains.**
- 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 (13) : 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 */).

```
> gcc testpipe5.c -o testpipe5
> ./testpipe5 "pipe   rtryu" &
[1] 15019
> pipe   rtryu
> 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 (14) : Redirection de stdin et stdout

- La duplication de descripteur permet 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.

Les tubes anonymes (15) : Exemple 2

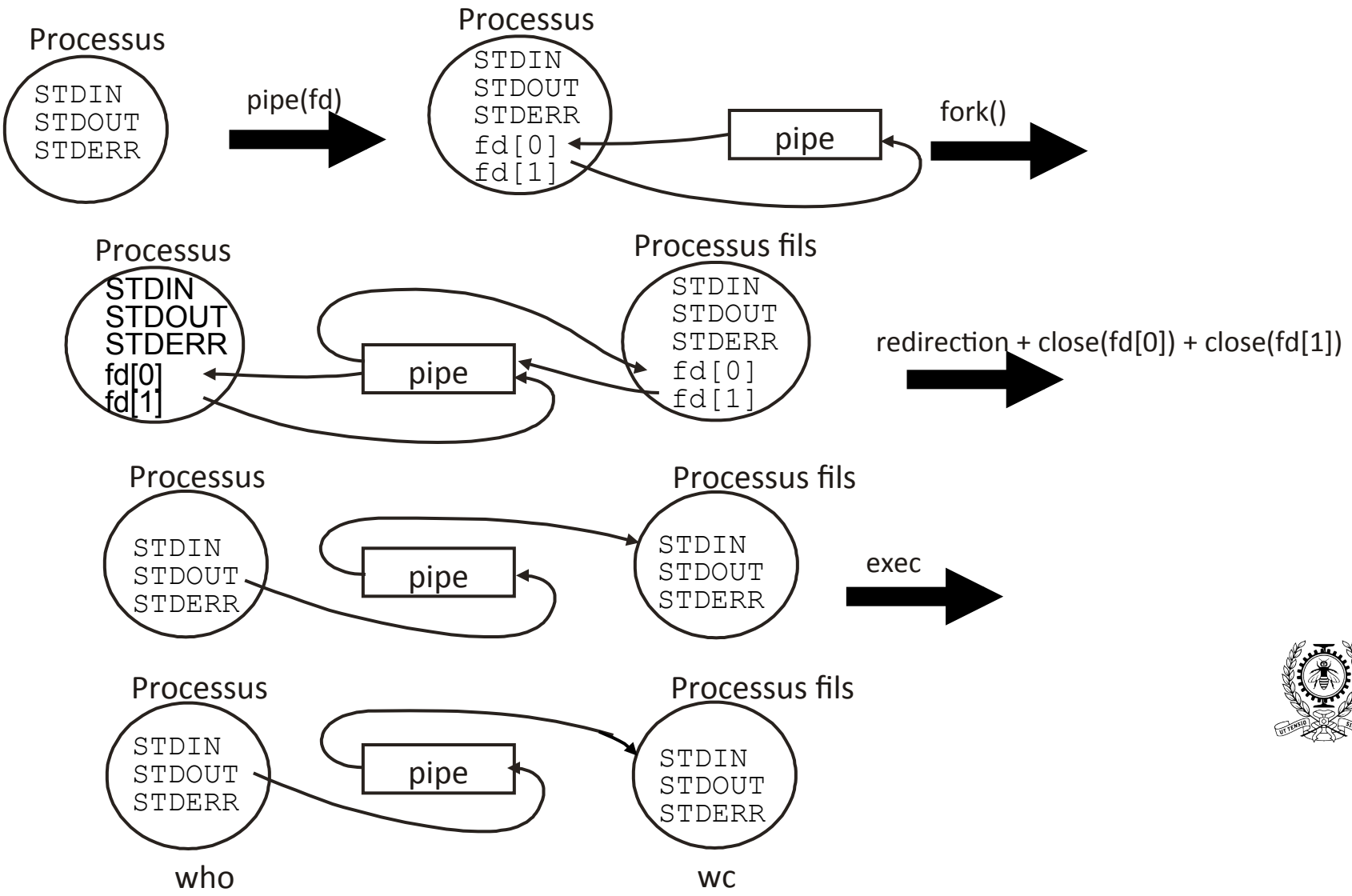
- Ce programme réalise l'exécution en parallèle de deux commandes shell. Un tube connecte stdin de la 1ière vers stdout de la 2ième.

```
//programme pipecom.c
#include <unistd.h>     //pour fork, close…
#include <stdio.h>
#define R 0
#define W 1
int main (int argc, char * argv [ ] )
{  int fd[2] ;
   pipe(fd) ;                  // creation d'un tube sans nom
   char message[100] ;   // pour récupérer un message
   int nboctets ;
   char * phrase = " message envoyé au père par le fils";
   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 (16) : Exemple 2

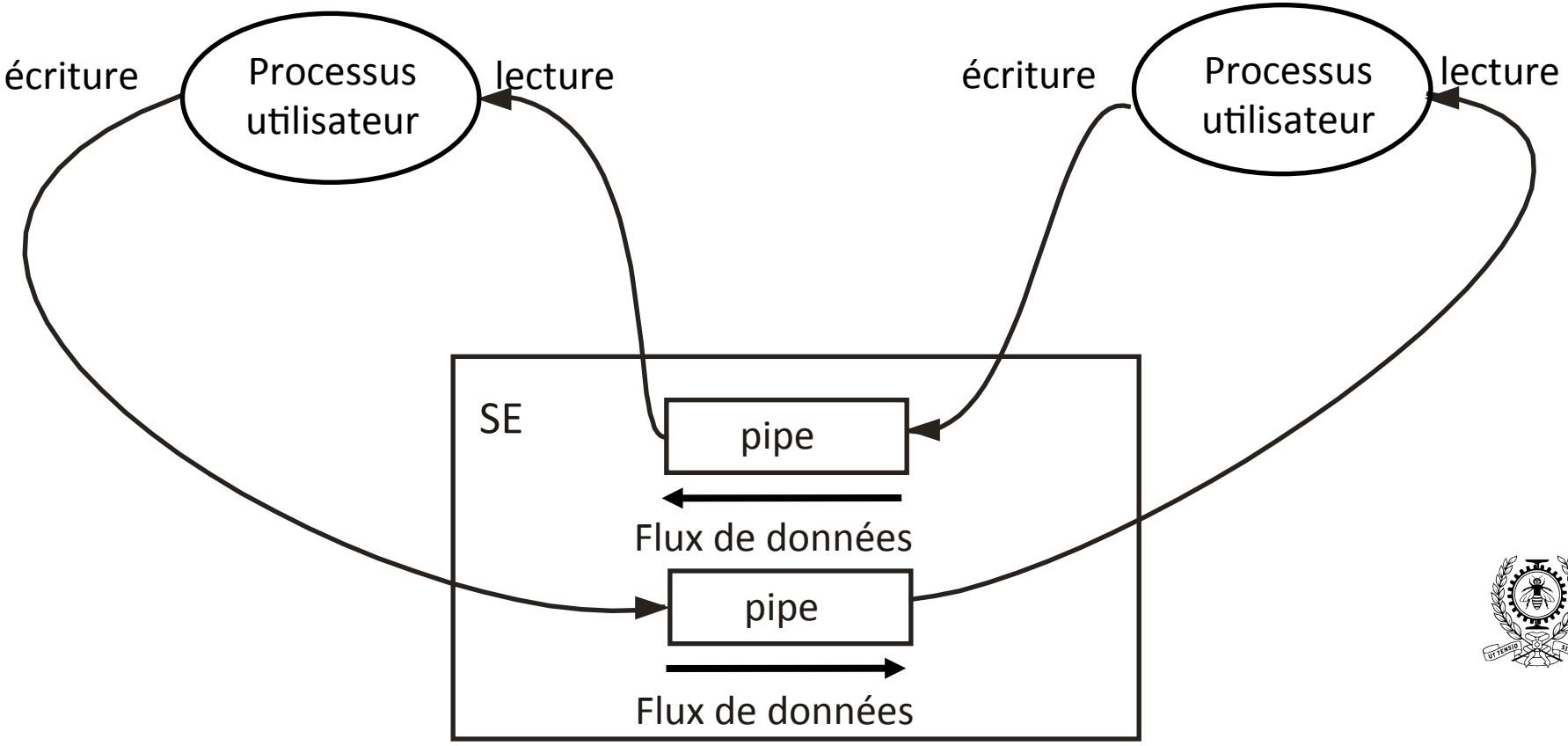
```
    else  // processus fils (lecteur)
    {     // fermeture du descripteur non utilisé d'écriture
          close(fd[W]) ;
          // copie fd[R] dans le descripteur 0
          dup2(fd[R],0) ;
          close (fd[R]) ; // fermeture du descripteur de lecture
          // exécute le programme lecteur
          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 (17) : Exemple 2



Les tubes anonymes (18) : 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 ;
  - Leur capacité maximale est de 40K.
  - 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);
      } else
        printf( " désolé, le tube n'est pas disponible \n");
    close(fd);
    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);
        } else
          printf( "désolé, le tube n'est pas disponible\n");
        close(fd);
    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 (spécifier O_NONBLOCK sinon).

- 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.

- Chaque signal :
  - a un nom,
  - a un numéro,
  - a un gestionnaire (handler) et
  - est, en général, associé à un type d'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), type d'événement, 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 l'exécution du processus), ou
  - continue (reprendre l'exécution si le processus 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 événement sans être obligés de tester en permanence leurs arrivées.

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'évènement). 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 bloqués (ceux dont le traitement est différé).
- 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 rattaché 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 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 <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(); // Mise en attente d'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);
}
}
```

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();                 // Mise en attente d'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");
   }
}
```

```
-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 est émis, il est mis en attente jusqu'à ce qu'il devienne non bloqué.
- L'appel système sigpending permet de récupérer les signaux en attente.

  int sigpending (sigset_t *set);

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 pour la création de segments partagés sont dans les librairies : <sys/ipc.h> et <sys/shm.h> :

  - L'appel système shmget permet de créer ou de retrouver un segment de données.

  - L'appel système shmat permet d'attacher un segment de données à un processus.

  - L'appel système shmctl permet, entre autres, de détacher un segment de données d'un processus.

Segments de données partagés (1) : Exemple 7

- Les deux programmes suivants communiquent au moyen d'un segment de données créé par le premier. Le segment de données est de clé 5. Seuls les processus du groupe peuvent y accéder.

- Le premier programme attache le segment créé à son espace d'adressage puis écrit dans ce segment la valeur 1190. Enfin, il détache après deux secondes le segment de son espace d'adressage.

- Le second programme attache le segment à son espace d'adressage puis accède en lecture au segment. Ensuite, il détache le segment de son espace d'adressage.

Segments de données partagés (2) : Exemple 7

```
// programme shm1.cpp
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdlib.h>
#include <stdio.h>
int main ( )
{      int *    adr1 ;
       int status, cle = 5;
       if( (status = shmget(cle, sizeof(int), IPC_CREAT | IPC_EXCL | 0600))==-1)
         exit(1);
     printf( "status = %d \n", status);
      if((adr1 =(int *) shmat(status,NULL,0)) == (int *)-1)
         exit(1);
    *adr1 = 1190;
   printf("adr1 = %p, *adr1 = %d \n", adr1, *adr1);
    sleep(2);
    if( shmctl(status, IPC_RMID, NULL) == -1)
         exit(1);
    exit(0);
}
```

Segments de données partagés (3) : Exemple 7

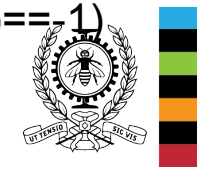
```
pascal> shm1 & shm2
[1] 32254
status = 1114113
adr1 = 0x10c6e8000, *adr1 = 1190
status = 1114113
adr2 = 0x1063fb000, *adr2 = 1190
```

```
// programme shm2.cpp
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdlib.h>
#include <stdio.h>
int main ( )
{       int *  adr2;
        int status, cle = 5;
        sleep(1);
        if( (status = shmget(cle, sizeof(int), 0))==-1)
                    exit(1);
     printf("status = %d \n", status);
        if((adr2 =(int*) shmat(status,NULL,0)) == (int *) -1)
          exit(1);
         printf("adr2 = %p, *adr2 = %d  \n", adr2, *adr2);
        if( shmctl(status, IPC_RMID, NULL) == -1)
          exit(1);
     exit(0);
}
```

Segments de données partagés (4) : Exemple 8

```
// programme shmperefils.cpp
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/wait.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#define MAX 1000000000
void *count(void *arg);
int main()
{
      int p, i;
       int status, cle = 15;
      // créer un segment de données par le père
    if( (status = shmget(cle, sizeof(unsigned long long), IPC_CREAT | IPC_EXCL | 0600))==-1)
                  exit(1);
    if( (status = shmget(cle, sizeof(unsigned long long), 0))==-1)
                  exit(1);
    printf ("status = %d \n", status);
```

```
void *count(void *arg) {
  volatile unsigned long long*var =
                (unsigned long long*) arg;
  volatile unsigned long long i;
  for (i = 0; i < MAX; i++)
      *var = *var + 1;
  return NULL;
}
```



Segments de données partagés (5) : Exemple 8

```
    unsigned long long * pa=NULL;
     // “mapper" le segment de données dans l’espace d’adressage du processus père
    if((pa =(unsigned long long *) shmat(status,NULL,0)) == (unsigned long long  *)-1)
        exit(1);
    *pa=0; // initialisation
    printf(" Father %d: init value of *pa =%lld \n”, getpid(), *pa);
    for (i = 0; i < 2; i++) { // creation de deux fils
        if ((p = fork()) < 0) return 1;
        if (p == 0) {   count(pa);
        printf(“Child %d: *pa=%lld\n", getpid(), *pa);
        if( shmctl(status, IPC_RMID, NULL) == -1)
                    exit(1);
        return 0;
        }
    }
    for (i = 0; i < 2; i++) { wait(NULL);} // attendre la fin des deux fils
    printf(“*pa=%lld \n", *pa);
    if( shmctl(status, IPC_RMID, NULL) == -1)
        exit(1);
    return 0;
}
```

Segments de données partagés (6) : Exemple 8

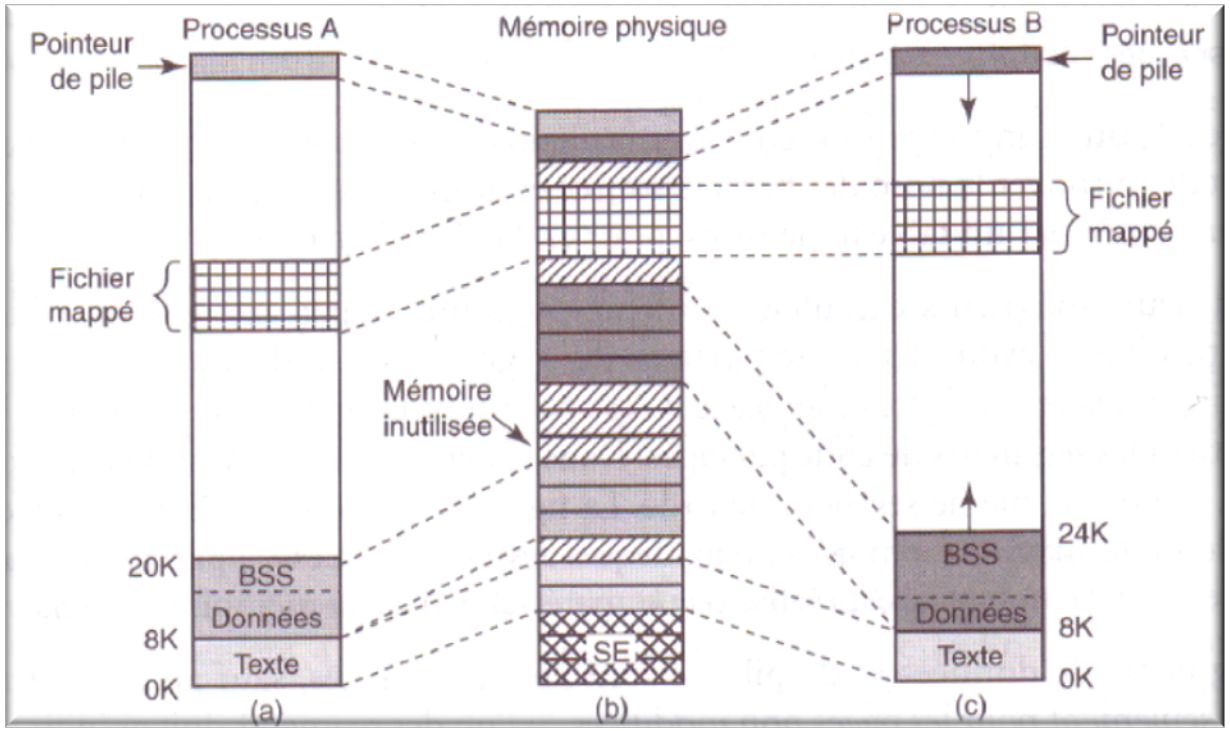
```
jupiter% gcc shmperefils.cpp -o shmperefils

jupiter% time ./shmperefils
status = 589825
 Father 31604: init value of *pa = 0
child 31606 *pa=971283654
child 31605 *pa=1005569185
*pa=1005569185

real    0m6.472s
user    0m12.842s
sys     0m0.009s
```

```
jupiter% time ./shmperefils
status = 655361
 Father 31616: init value of *pa = 0
child 31617 *pa=873629461
child 31618 *pa=928326212
*pa=928326212

real    0m6.635s
user    0m13.117s
sys     0m0.008s
```

Segments de données partagés (7)

- 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.



Lectures suggérées

- Notes de cours: Chapitre 5
(http://www.groupes.polymtl.ca/inf2610/documentation/notes/chap5.pdf)

- Chapitre 5 (pp 85 - 104)

M. Mitchell, J. Oldham, A. Samuel - Programmation Avancée sous Linux- *Traduction : Sébastien Le Ray* (2001) **Livre disponible dans le dossier Slides Automne 2017 du site moodle du cours.**

- 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 2017 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 2017 du site moodle du cours.**

Les tubes anonymes : Transfert de descripteurs de fichiers au fils 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$
```