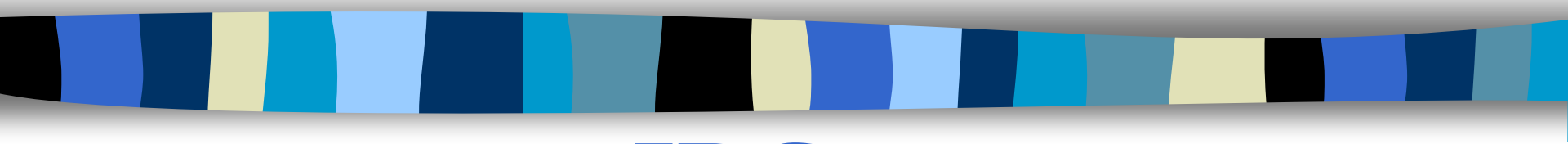
Chapitre 4



IPC

(Inter-Process Communication)

Plan du chapitre

- Introduction
- Les tubes
  - Tubes anonymes
  - Tubes nommés
- Les signaux
- Les files de messages
- Segments mémoire partagée

IPC
Introduction

- En général, plusieurs processus (threads) coopèrent pour réaliser une tâche complexe.
- Ces processus s'exécutent en parallèle sur un même ordinateur (monoprocesseur ou multiprocesseurs) ou sur des ordinateurs différents.
- Ils doivent s'échanger des informations (communication interprocessus).

➔ *Inter-Process Communication(IPC) :* Ensemble de mécanismes permettant la communication et la synchronisation entre processus.

IPC - Communication

- Les **tubes**
  - Permettent l'échange de message (ou de données) entre des processus
    - Flux de données entre deux processus
- Les **signaux**
  - Envoi asynchrone d'un signal ou *événement* à un ou plusieurs processus
- Les **files** de message (*message queue)*
  - Permet l'envoi de plusieurs messages simultanément
  - Utiliser par RPC (invocation de méthodes) distantes, par exemple
- Les segments de **mémoire partagée**

Tubes - Définition

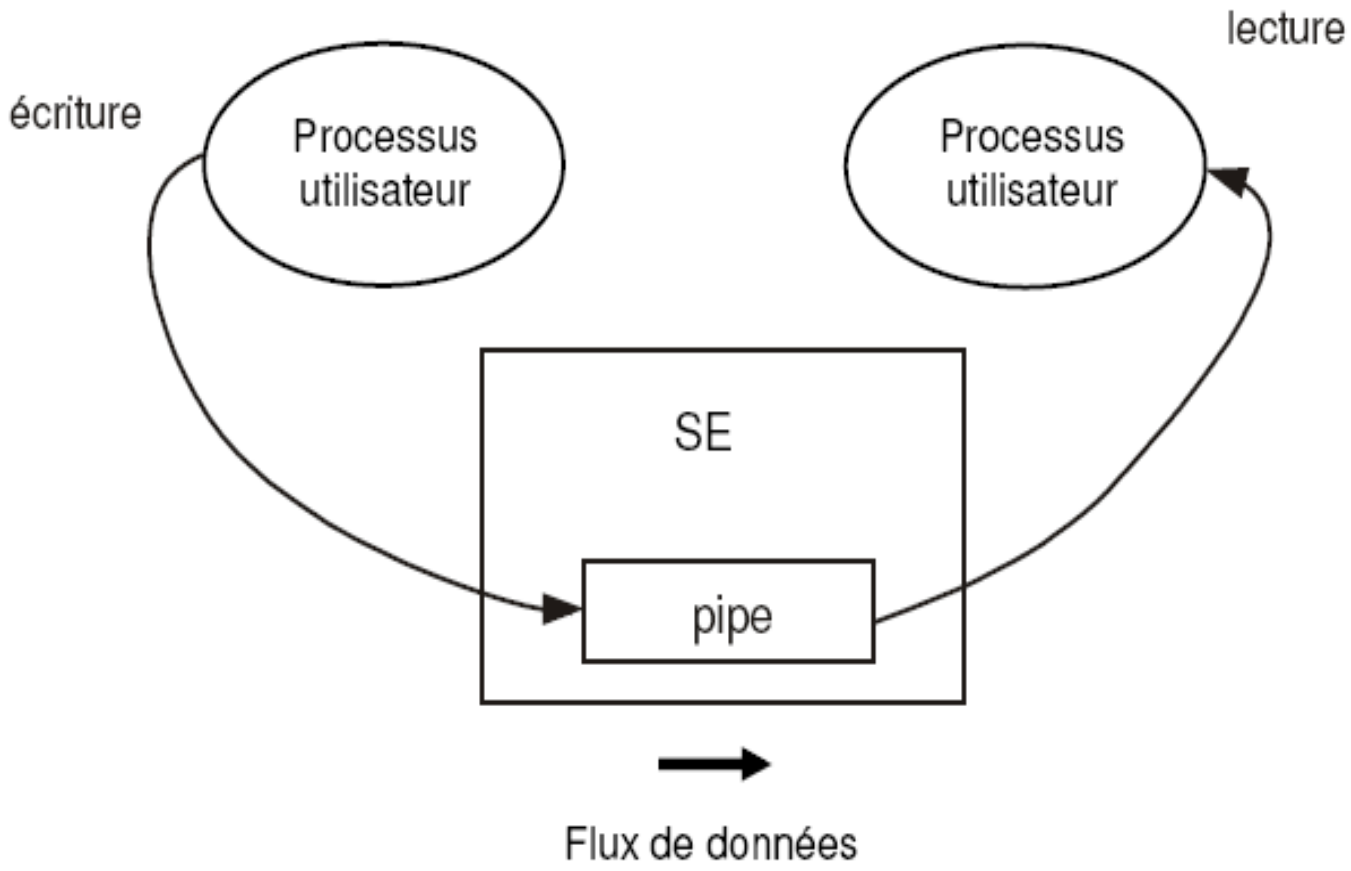
- Les tubes sont des **canaux de communication** entre processus dans une **même** machine.
  - C'est un mécanisme de communication qui interagit fortement avec le *système de fichiers.*
  - Un tube est un fichier d'un type spécial (type *pipe*).
  - Un tube possède donc deux descripteurs de fichier.
  - Il est possible d'utiliser les primitives *read()* et *write()* puisqu'il est possible d'acquérir un descripteur

- 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 (1)

- Les tubes anonymes (pipes) peuvent être considérés comme des fichiers temporaires. Ils permettent d'établir des communications entre processus dépendants.
- La communication est (de préférence) unidirectionnelle. Une communication bidirectionnelle nécessite deux tubes.
- Il est caractérisé par deux descripteurs de fichiers (lecture et écriture) et sa taille limitée (PIPE_BUF) approximativement égale à 4KO.
- La lecture est **destructive**, c'est-à-dire qu'elle ne s'effectue qu'une seule fois.
  - Gestion de type **FIFO**
- Lorsque tous les descripteurs du tube seront fermés, le tube est détruit.
- Les tubes anonymes peuvent être créés par :
  - l'opérateur « | »
  - l'appel système pipe( ).

Les tubes anonymes (2) communication unidirectionnelle



Les tubes anonymes (3)
L'opérateur « | »

- 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)
L'opérateur « | »

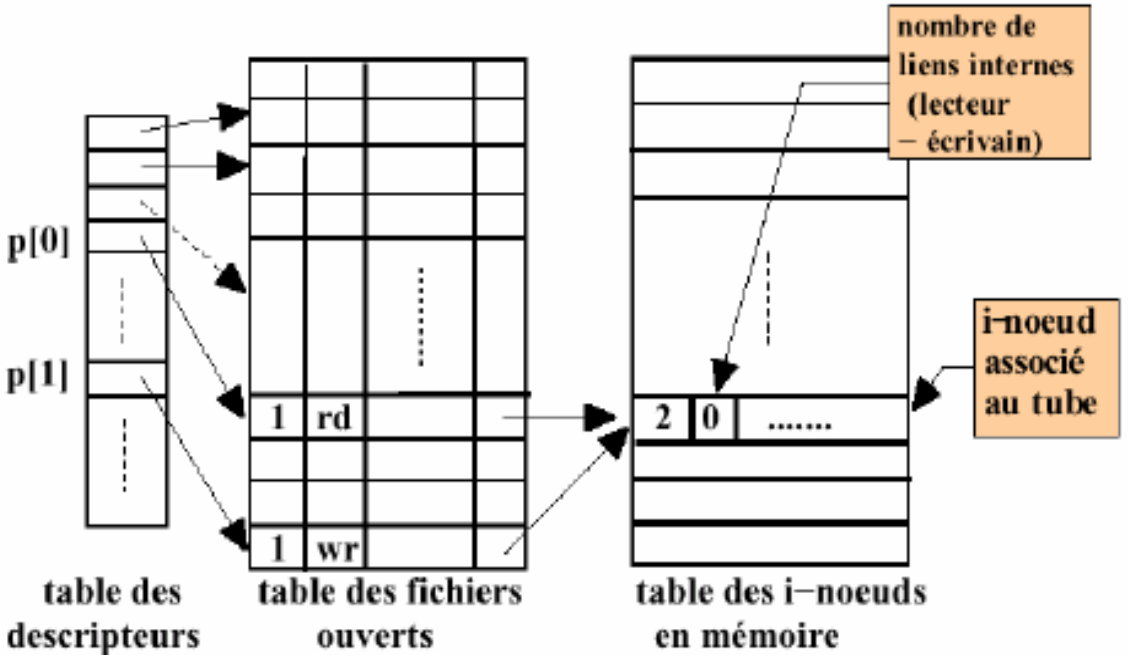
- Les deux processus s'exécutent en parallèle, les sorties du premier processus sont stockées dans le tube de communication.
- Lorsque le tube devient plein, le premier processus est suspendu jusqu'à ce qu'il y ait libération de l'espace nécessaire pour stocker des données.
- De façon similaire, lorsque le tube devient vide, le second processus est suspendu jusqu'à ce qu'il y ait des données sur le tube.

Les tubes anonymes (5)
Création

- 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)
Création

```
int p[2] ;
pipe (p) ;
```



Les tubes anonymes (7)
Création

- Les tubes anonymes sont, en général, 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 fichier, non utilisé, de lecture du tube ;
  4. De même, le processus lecteur ferme le 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 3

- Dans l'exemple suivant, le processus père crée un tube de communication pour communiquer avec son processus fils. La communication est unidirectionnelle du processus fils vers le processus père.

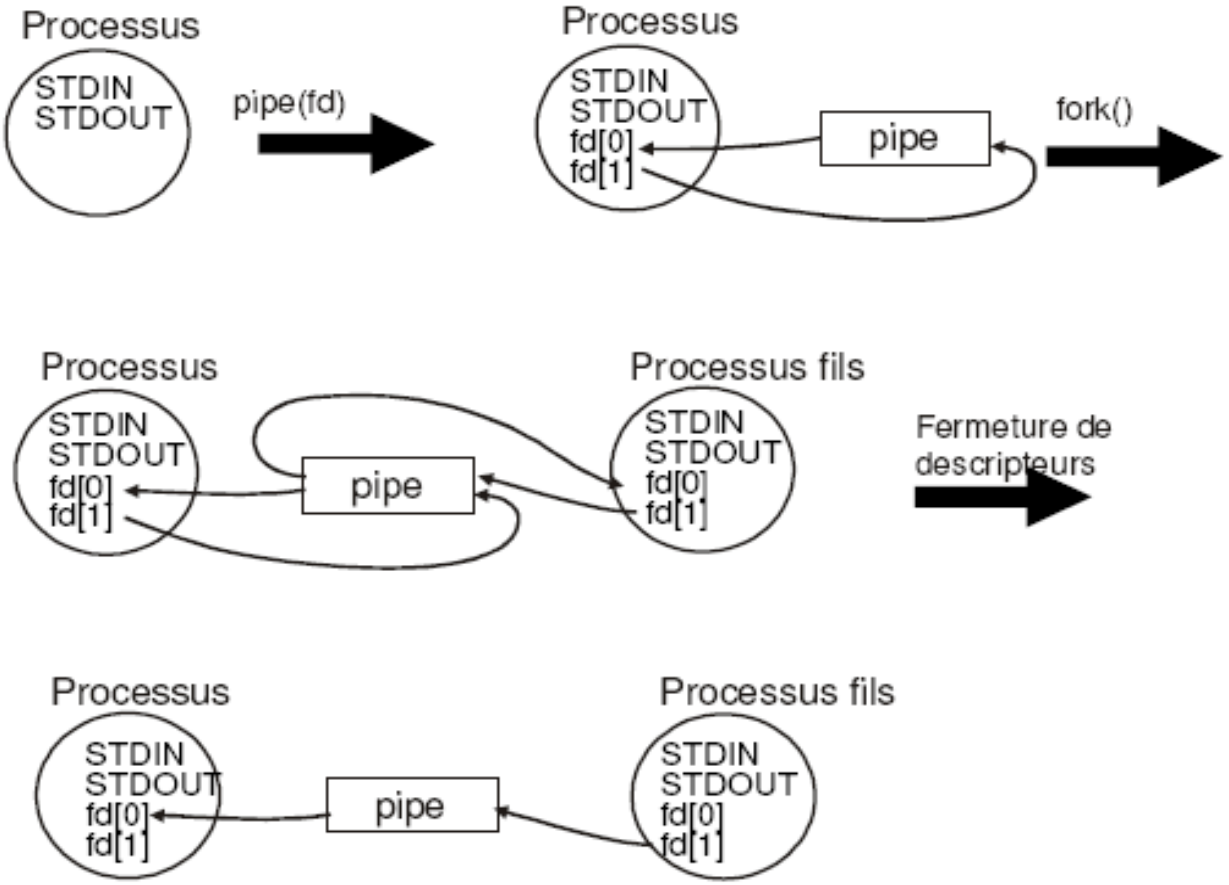
```
//programme upipe.c
#include <sys/types.h> //pour les types
#include <unistd.h> //pour fork, pipe, read, write, close
#include <stdio.h> // pour printf
#define R 0
#define W 1
int main ( )
{       int fd[2] ;
        pipe(fd) ; // création d'un tube sans nom
        char message[100] ; // pour récupérer un message
        int nboctets ;
        char * phrase = " message envoyé au père par le fils";
        if (fork() ==0) // création d'un processus fils
        {       close(fd[R]) ; // Le fils ferme le descripteur
   non utilisé de lecture
                // dépôt dans le tube du message
        write(fd[W],phrase, strlen(phrase)+1) ;
        close (fd[W]) ; // fermeture du descripteur d'écriture
        }
```

Les tubes anonymes (9)
Exemple 3 (suite)

```
        else
        {       // Le père ferme le descripteur non utilisé
   d'écriture
                close(fd[W]) ;
                // extraction du message du tube
                nboctets = read (fd[R], message,100) ;
                printf ("Lecture %d octets : %s\n", nboctets,
   message) ;
                // fermeture du descripteur de lecture
                close (fd[R]) ;
        }
        return 0;
}
```

jupiter% gcc -o upipe upipe.c
jupiter% upipe
Lecture 36 octets : message envoyé au père par le fils

Les tubes anonymes (10)
Exemple 3



Les tubes anonymes (11)
Quelques précisions

- 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 dont le nombre de lecteurs est 0, génère le signal SIGPIPE.
- Par défaut, les lectures et les écritures sont bloquantes.

Les tubes anonymes (12)
Redirection des entrées et sorties standards

- La duplication de descripteur permet à un processus de créer un nouveau descripteur (dans sa table des descripteurs) synonyme d'un descripteur déjà existant.

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

  dup crée et retourne un descripteur synonyme à desc.

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

  dup2 transforme desc2 en un descripteur synonyme de desc1.

- Ces fonctions peuvent être utilisées pour réaliser des redirections des fichiers d'entrées et sorties standards vers les tubes de communication.

Les tubes anonymes (13)
Exemple 4

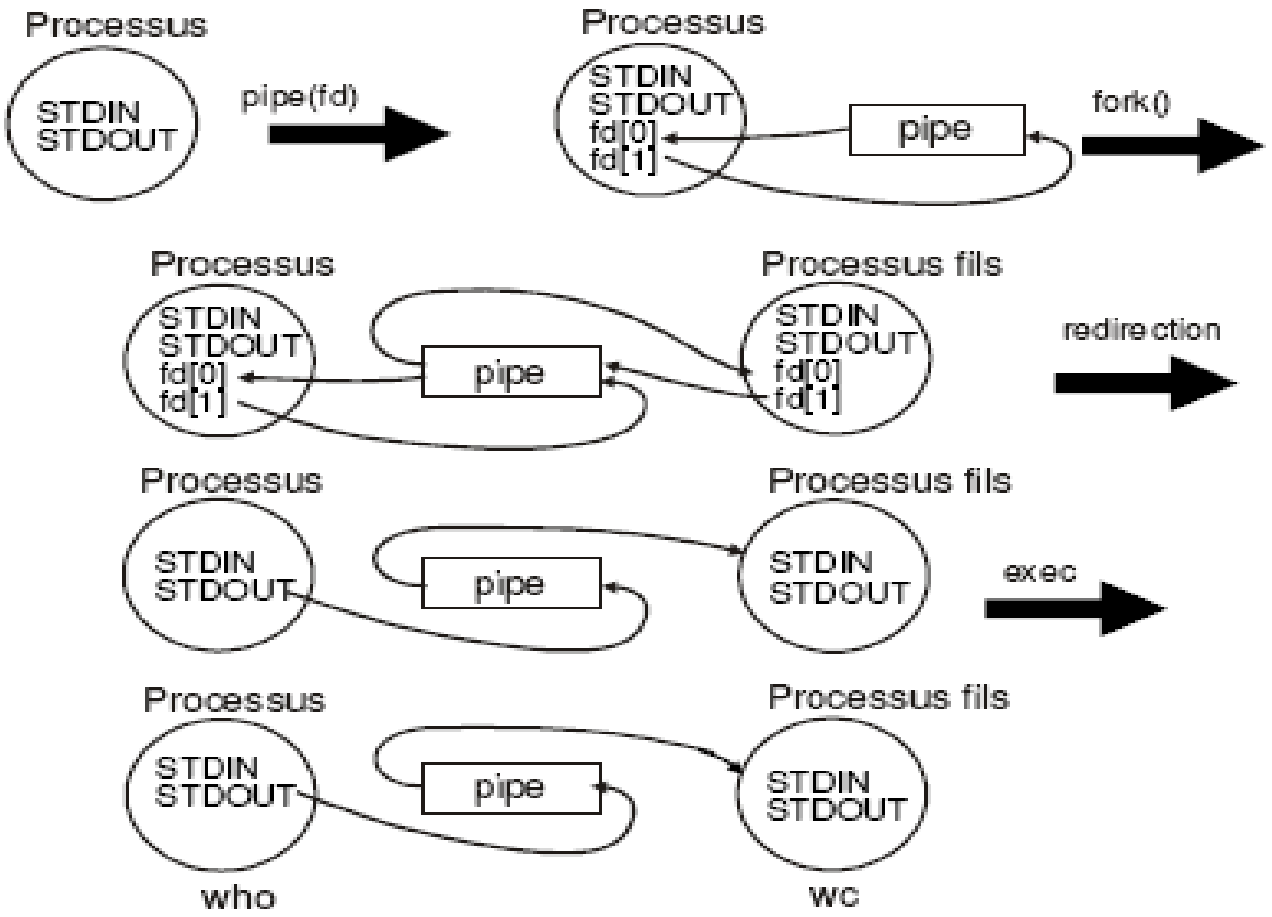
- Ce programme réalise l'exécution en parallèle de deux commandes shell. Un tube est créé pour diriger la sortie standard de la première commande vers l'entrée standard de la deuxième.

```
//programme pipecom.cpp
#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 non utilisé 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 le programme écrivain
                                perror("error dans execlp") ;
                }
```

Les tubes anonymes (14)
Exemple 4 (suite)

```
        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
```

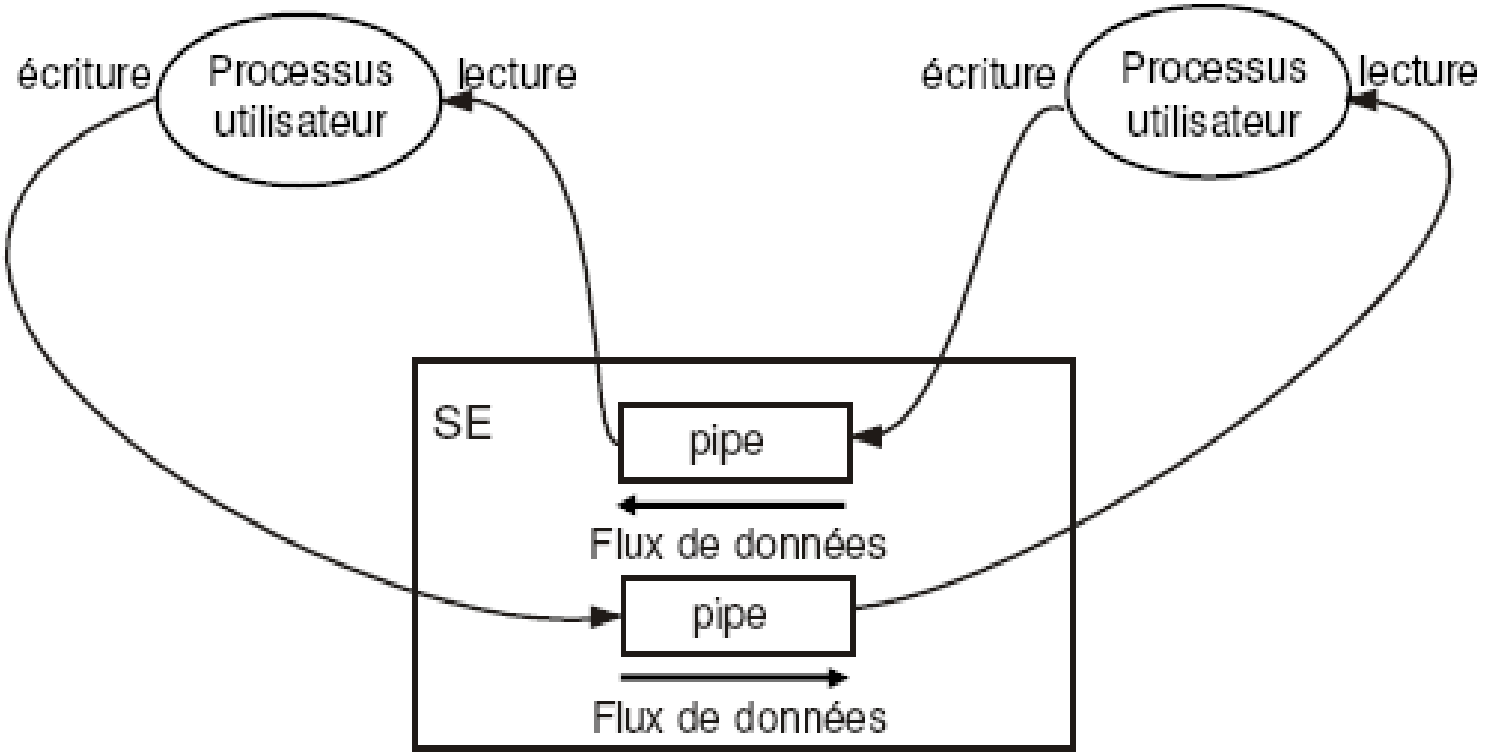
Les tubes anonymes (15)
Exemple 4 (suite)



Les tubes anonymes (16)
Remarques

- Le processus fils de l'exemple précédent a inclus le caractère nul dans le message envoyé au processus père.
- Si le processus écrivain envoie plusieurs messages de longueurs variables sur le tube, il est nécessaire d'établir des règles qui permettent au processus lecteur de déterminer la fin d'un message (protocole de communication).
- Par exemple, le processus écrivain peut précéder chaque message par sa longueur ou terminer chaque message par un caractère spécial comme un retour chariot ou le caractère nul.
- La communication bidirectionnelle est possible en utilisant deux tubes (un pour chaque sens de communication).

Les tubes anonymes (17)
Communication bidirectionnelle



Les tubes nommés

- Les tubes de communication nommés fonctionnent aussi comme des files de discipline FIFO (first in first out).
- Ils sont plus intéressants que les tubes anonymes car ils offrent, en plus, les avantages suivants :
  - Ils ont chacun un nom qui existe dans le système de fichiers (table des fichiers); 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 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 5 : programme writer

```
// 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 sur 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 5 : programme reader

```
// 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)
Exécution des programmes writer et reader

- 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)
Exécution des programmes writer et reader

- 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)
Quelques précisions

- 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) alors qu'il n'y a aucun processus qui ait fait une ouverture en écriture (resp. lecture).
  -> le processus sera bloqué jusqu'à ce qu'un processus effectue une ouverture en écriture (resp. en lecture).
- Attention au situation d'interblocage

```
/* processus 1 */
int f1, f2;
...
f1 =open("fifo1", O_WRONLY);
f2 =open("fifo2", O_RDONLY);
...
```

```
/* processus 2 */
int f1, f2;
….
f2=open("fifo2", O_WRONLY);
f1=open("fifo1", O_RDONLY);
…
```

Signaux (1)
Définition

- Les signaux jouent le rôle de sonnette d'alarme, ils permettent de communiquer des informations à un processus afin qu'il prenne une action immédiate. (Exemple : la sonnerie du téléphone). C'est une interruption logicielle asynchrone qui a pour but d'informer de l'arrivée d'un événement (outil de base de notification d'évènement). La fonction kill() permet d'envoyer un signal à un processus.

- Ce mécanisme de communication permet à un processus de réagir à un événement sans être obligé de tester en permanence l'arrivée.

- Le signal peut-être envoyé par l'usager
  - SIGINT (CTRL/C), SIGQUIT ou SIGTSTP

- peut aussi être envoyé par le système.
  - SIGSEGV est généré en cas de débordement de la mémoire.
  - 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 (émis) au processus fautif.

- Le signal provoque l'exécution d'un traitant (*callback* ou *handler*) dans l'espace utilisateur.
  - Le traitant peut être considéré comme une routine de service implémenté par l'utilisateur (dans l'espace utilisateur).

- Le système d'exploitation gère un ensemble de signaux. Chaque signal a un nom, un numéro, un gestionnaire (handler) et est associé à un type d'événement (man 7 signal).

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

- Un processus peut également se mettre en attente de signaux (pause(), sigpause()).

Signaux (2)
Traitement

- Comme on ne peut pas prévoir à l'avance à quel moment un signal va arriver ni même s'il doit arriver, une fonction est enregistrée à l'avance pour chaque signal (gestionnaire de signal).

- Le système d'exploitation associe à chaque signal un traitement par défaut (gestionnaire par défaut du signal) :
  - abort (génération d'un fichier core et arrêt du processus);
  - exit (terminaison du processus sans génération d'un fichier core);
  - ignore (le signal est ignoré);
  - stop (suspension du processus);
  - continue (reprendre l'exécution si le processus est suspendu sinon le signal est ignoré).

- Par exemple, SIGUSR1 et SIGUSR2 tuent le processus, SIGCHLD est ignoré (man 7 signal pour plus de détails).

Les signaux (3)
Traitement

- Le système d'exploitation permet à un processus de redéfinir pour certains signaux leurs gestionnaires. Un processus peut donc indiquer au système ce qui doit se passer à la réception d'un signal :
  - ignorer le signal (certains signaux ne peuvent être ignorés),
  - le prendre en compte (en exécutant le traitement spécifié par le processus),
  - exécuter le traitement par défaut, ou
  - le bloquer (le différer).

- Si un processus choisit de prendre en compte un signal qui lui est destiné (capture d'un signal), il doit alors spécifier la procédure de gestion de signal.

- Par exemple, la touche d'interruption Ctrl+C génère le signal SIGINT. Par défaut, ce signal arrête le processus. Le processus peut associer à ce signal un autre gestionnaire de signal.

- Les signaux SIGKILL et SIGSTOP ne peuvent être ni capturés, ni ignorés, ni bloqués.

Les signaux (4)
Envoi d'un signal

- L'appel système qui permet d'envoyer un signal à un 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 du processus émetteur.

  Il retourne 0 en cas de succès et -1 en cas d'erreur.

- 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 le super-utilisateur.

- La fonction raise() permet à un processus de s'envoyer un signal.

```
#include <signal.h>
int raise (int sig);
```

- L'appel raise(sig) est équivalent à kill(getpid(), sig).

Les signaux (5)
Réception d'un signal

- Un processus vérifie s'il a reçu un signal aux transitions suivantes :
- quand il repasse du mode noyau au mode utilisateur c'est-à-dire quand il a terminé un appel système. Ce qui veut dire que le traitement du signal est différé tant que le processus destinataire n'est pas revenu en mode utilisateur
  - avant de passer dans un état bloqué
  - en sortant de l'état bloqué.
- L'adresse de retour de la fonction qui traite le signal est celle de l'instruction suivant celle qui a provoqué le déclenchement.
- Le traitement des signaux reçus se fait dans le contexte d'exécution du processus.
- C'est dans le bloc de contrôle (BCP) de chaque processus que l'on trouve la table de gestion des signaux, Cette table contient, pour chaque signal défini sur la machine, une structure *sigvec* suivante :

```
{
        bit pendant;
        void (*traitement)(int);
}
```

Le drapeau *pendant* indique que le processus a reçu un signal, mais n'a pas encore eu l'occasion de prendre en compte ce signal.

Les signaux (6)
Capture d'un signal

- La fonction signal().

#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.
  - SIG_DFL désigne l'action par défaut et
  - SIG_IGN indique au système que le processus appelant veut ignorer le signal.
  - signal retourne le gestionnaire précédent ou SIG_ERR en cas d'erreur.

Les signaux (7)
Attente d'un signal

- L'appel système pause suspend l'appelant jusqu'au prochain signal.

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

- 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 );
```

- L'appel système alarm(sec) programme l'envoi par le système du signal SIGALRM au processus appelant dans sec secondes. La valeur retournée est le temps restant dans l'horloge. S'il y a deux appels successifs l'horloge est réinitialisée

```
int alarm (int sec);
```

Les signaux (8)
Exemple 1 : 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 (9)
Exemple 2

```
// 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();
}
```

```
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
4637 pts/2 00:00:00 bash
4664 pts/2 00:00:00 test-signaux
4665 pts/2 00:00:00 ps
bash-2.05b$ kill -SIGTERM 4664
bash-2.05b$ ps
PID TTY TIME CMD
4637 pts/2 00:00:00 bash
4664 pts/2 00:00:00 test-signaux
4666 pts/2 00:00:00 ps
bash-2.05b$ kill -SIGUSR2 4664
bash-2.05b$ On peut maintenant m'eliminer
bash-2.05b$ ps
PID TTY TIME CMD
4637 pts/2 00:00:00 bash
4664 pts/2 00:00:00 test-signaux
4667 pts/2 00:00:00 ps
bash-2.05b$ kill -SIGTERM 4664
bash-2.05b$ ps
PID TTY TIME CMD
4637 pts/2 00:00:00 bash
4668 pts/2 00:00:00 ps
[1]+ Terminated ./test-signaux
bash-2.05b$
```

Les signaux (19)
Limitations

- Les signaux ont néanmoins des limitations :
  - Ils coûtent chers car ils sont lancés par un appel système, le récepteur est interrompu, sa pile à l'exécution est modifiée, le gestionnaire prend la main, la pile à l'exécution est restaurée.
  - Ils sont en nombre limité (environ une trentaine dont seulement deux signaux ne sont pas utilisés par le système lui même).
  - Ils sont strictement limités au rôle de notification d'évènement et ne véhiculent pas d'information (même pas l'identité de l'émetteur).

Files de messages (message queues)

- permet d'échanger des messages entre processus (plutôt qu'une séquence d'octets, comme avec les tubes)
- on peut associer un type (= entier) à un message; le type peut être utilisé lors de la commande de réception (ordre FIFO par défaut)
- désignation
  - externe: *clé* (désignation utilisée lors de la création/ouverture d'une file de messages; permet à deux processus, en utilisant la même clé, de désigner la même file de messages)
  - interne: *identificateur* (désignation utilisée par un processus après la création/ouverture d'une file de message par un processus)
- protection: basée sur UID et GID (comme pour les fichiers)
- mêmes mécanismes de désignation et protection que pour les *sémaphores* et les *segments de mémoire partagée* (mais 3 espaces de désignation distincts)

NB : les files de message n'existent pas dans POSIX.1 existent sous une forme différente dans POSIX.4

La gestion des clés

- La primitive ftok construit une clé à partir d'une référence de fichier existant et d'un nombre entier : attention si le fichier est déplacé, la clé sera modifiée ! ! !

  #include <sys/ipc.h>

  key_t ftok(const char *référence,int nombre) ;

Exemple

La référence de fichier et le nombre entier sont passés en arguments, sur la ligne de commande.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
main(int argc,char *argv[])
      { key_t cle;
if(argc != 3) {
   fprintf(stderr,"%s: syntaxe:%s <ref_cle>
  <nbr_cle>\n",argv[0],argv[0]);
exit(1);
}
if((cle = ftok(argv[1],atoi(argv[2]))) == -1) {
   fprintf(stderr,"Probleme de cle\n");
   exit(2);
}}
```

Création d'une file de messages

int msgget (key_t Key, int option)

- **retourne** l'identificateur de la file, que le processus utilisera par la suite pour manipuler la file de messages **(ou -1 en cas d'erreur).**
- La clé **key** peut être obtenue grâce à la fonction ftok( ). Le type key_t est le type int.
- option : combinaison, avec l'opérateur | de droits d'accès et de constantes : **IPC_CREAT, IPC_EXCL** …

  – Une nouvelle file de messages est créée si **key** a la valeur **IPC_PRIVATE** ou si aucune file de message n'est associée à key, et si la valeur **IPC_CREAT** a été introduite dans option.

  – La fonction **msgget** échouera si option indique à la fois **IPC_CREAT** et **IPC_EXCL** et si une file de messages existe déjà associée à key.

  – Option : séquence de bits; pour la création on doit avoir:
    option = bit IPC_CREATE à 1 + droits d'accès.
    Exemple: id = msgget(clé, 0666 | IPC_CREATE)

NB: la commande ipcs permet de consulter les tables IPC du système et de visualiser l'ensemble des files de messages, de sémaphores et de segments de mémoire partagée.

Table des IPC (fonction ipcs)

commande ipcs sans paramètre

```
--> ipcs↵
IPC status from /dev/kmem as of Sun Jul  7 08:41:31 2002
T      ID     KEY        MODE        OWNER    GROUP
Message Queues:
q       0 0x3c200834 -Rrw--w--w-      root      root
q       1 0x3e200834 --rw-r--r--      root      root
Shared Memory:
m       0 0x2f1c0002 --rw-------      root       sys
m       1 0x41200679 --rw-rw-rw-      root      root
m   48005 0x00000000 --rw-rw-rw-   rifflet     users
m   43409 0x41440014 --rw-rw-rw-   rifflet     users
Semaphores:
s       0 0x2f1c0002 --ra-ra-ra-      root       sys
s       1 0x41200679 --ra-ra-ra-      root      root
s   18192 0x41440014 --ra-ra-ra-   rifflet     users
-->
```

IPC anonyme créé avec la clé IPC_PRIVATE

Envoi d'un message

int msgsnd (int id, void *pt_msg, int long, int option)

- id: identificateur de la file de message
- pt_msg: pointeur sur le message à envoyer
- long: longeur du message
- option: permet par exemple de ne pas bloquer l'émetteur si la file de messages est pleine
- l'appel retourne 0 si ok

Réception d'un message

int msgrcv (int id, void *pt_msg, int longmax, int option, long type)

- id: identificateur de la file de messages
- pt_msg: adresse pour stocker le message reçu
- longmax: taille max de l'emplacement de stockage du message
- option: permet par exemple de ne pas bloquer si la file est vide
- type:
  - type = 0: le message en tête de la file est reçu
  - type > 0: le premier message dans la file du type donné est reçu
  - type < 0: le premier message dans la file du plus petit type T t.q. T ≤ |type| est reçu
- l'appel retourne la longueur du message reçu

Réception d'un message

Exemple d'utilisation du type d'un message
On considère 3 types:

- type = 1: messages les plus prioritaires
- type = 2: messages de priorité moyenne
- type = 3: messages les moins prioritaires
- msgrcv(, , , , 1) permet de recevoir un message de la priorité max
- msgrcv(, , , , -2) permet de recevoir un message de priorité 1 (si disponible), sinon un message de priorité 2
- msgrcv(, , , , -3) permet de recevoir un message de priorité 1 (si disponible), sinon un message de priorité 2 (si disponible), sinon un message de priorité 3

Destruction d'une file de messages

int msgctl (int id, int cmd, … arg)

- msgctl: fournit plusieurs commandes de contrôle, dont une commande permettant de détruire une file de messages
- id: identificateur de la file de messages;
- cmd: commande; pour détruire, la commande est IPC_RMID
- arg: arguments pour certaines commandes (pour détruire: NULL)

Exemple client/serveur

- Dans cet exemple, chaque client met son numéro de processus (fonction getpid) dans le champ type des messages de requête ce qui permet au serveur de l'identifier. Le serveur copie ce champ dans la réponse, ce qui permet au client de récupérer dans la queue des réponses les seuls messages qui le concernent.

Exemple processus client

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define FICHIER1 "client"
#define FICHIER2 "serveur"
#define PROJET 1
#define LG_MAX   512
struct msgform {
    long mtype;
    char mtext[ LG_MAX ];
    }  msg;

int main(void) {
    int res;
    int frequete, freponse;
    key_t clef_requetes, clef_reponses ;
clef_requetes= ftok(FICHIER1, PROJET);

if (clef_requetes == -1){
perror("Problème pour obtenir la clé");
exit(EXIT_FAILURE);
}
clef_reponses= ftok(FICHIER2, PROJET);
if (clef_reponses == -1){
perror("Problème pour obtenir la clé");
exit(EXIT_FAILURE);
}
    frequete = msgget(clef_requetes, 0700 | IPC_CREAT);
    if (frequete == -1) { perror("msgget"); return
(EXIT_FAILURE); }
    freponse = msgget(clef_reponses, 0700 | IPC_CREAT);
    if (freponse == -1) { perror("msgget"); return
(EXIT_FAILURE); }
    msg.mtype = getpid();
    strcpy(msg.mtext, "Hello");
    res = msgsnd(frequete, & msg, strlen(msg.mtext) + 1, 0);
    if (res == -1) { perror("msgsnd"); exit(0); }

    res = msgrcv(freponse, & msg, LG_MAX, getpid(), 0);
    if (res == -1) { perror("msgrcv"); exit(0); }

    printf("result : %s\n", msg.mtext);
    return (EXIT_SUCCESS);
    }
```

Exemple (suite) serveur

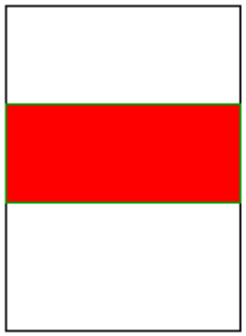
```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define FICHIER1 "client"
#define FICHIER2 "serveur"
#define PROJET 1
#define LG_MAX 512

struct msgform {
   long mtype;
   char mtext[ LG_MAX ];
   }  msg;
int main(void) {
   int res, i;
   key_t clef_requetes, clef_reponses ;
   int frequete, freponse;
clef_requetes= ftok(FICHIER1, PROJET);

if (clef_requetes == -1){
perror("Problème pour obtenir la clé");
exit(EXIT_FAILURE);
}
clef_reponses= ftok(FICHIER2, PROJET);
if (clef_reponses == -1){
perror("Problème pour obtenir la clé");
exit(EXIT_FAILURE);
}
 frequete = msgget(clef_requetes, 0700 | IPC_CREAT);
   if (frequete == -1) { perror("msgget"); return (EXIT_FAILURE);
}
   freponse = msgget(clef_reponses, 0700 | IPC_CREAT);
   if (freponse == -1) { perror("msgget"); return
(EXIT_FAILURE); }
   printf("Waiting a request...\n");
   res = msgrcv(frequete, & msg, LG_MAX, 0, 0);
   if (res == -1) { perror("msgrcv"); exit(0); }

   for(i=0; i < strlen(msg.mtext); i++)
      msg.mtext[i] = toupper(msg.mtext[i]);
   res = msgsnd(freponse, & msg, strlen(msg.mtext) + 1, 0);
   if (res == -1) { perror("msgsnd"); exit(0); }
return (EXIT_SUCCESS);
   }
```

Files de messages Posix.4 (message queues)

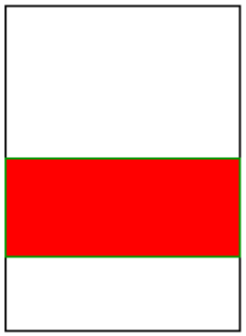
- Une file de messages Posix.4 est désignée comme un fichier (même si une file de messages ne se trouve pas dans l'espace des noms des fichiers, c-à-d n'est pas forcément visible par la commande ls);
- le nom doit commencer par /
- création (si n'existe pas) et ouverture : mqd_t mq_open(const char *nom, int option, ...);
- envoi d'un message: mq_send
- réception d'un message: mq_receive
- fermeture: mq_close
- destruction: mq_unlink
- réception asynchrone: mq_notify
  Si la file est vide, permet à un processus d'être notifié de la réception d'un message. Utilise le mécanisme des signaux.

Mémoire partagée

- il est possible de définir une zone de mémoire partagée entre plusieurs processus
- l'unité de partage est appelée segment*:* un segment peut faire partie de l'espace d'adressage de plusieurs processus



espace d'adressage de P1



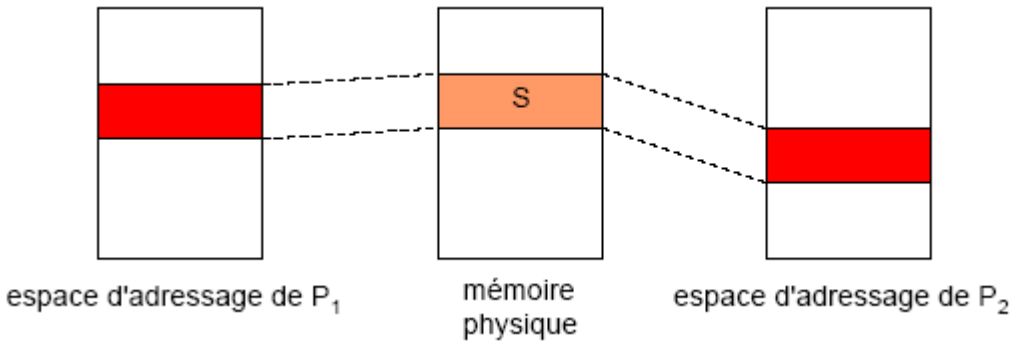
espace d'adressage de P2

en rouge: zone de mémoire partagée entre P1 et P2 (c'est physiquement le même emplacement mémoire; possible grâce au mécanisme de pagination de la mémoire).

NB: Les appels liés à la mémoire partagée n'existent pas dans POSIX.1.
Ils existent sous une forme différente dans POSIX.4

Mémoire partagée

- Pour partager une zone de mémoire entre P1 et P2 il faut procéder ainsi:
  - P1 ou P2 doit créer un segment de mémoire partagé S
  - P1 doit appliquer S dans son espace d'adressage
  - P2 doit appliquer S dans son espace d'adressage
  - à partir de là, le segment S est partagé (c'est une même zone en mémoire physique)



Création d'un segment de mémoire partagée

int shmget (key_t clé, int taille, int option);

- crée le segment de mémoire de nom clé si celui-ci n'existe pas; s'il existe déjà, permet d'obtenir l'identificateur du segment de mémoire, que le processus utilisera par la suite pour manipuler le segment de mémoire
- taille*:* taille du segment de mémoire
- option: permet par exemple de spécifier les droits d'accès au segment de mémoire
- l'appel retourne un identificateur permettant par la suite au processus de désigner le segment de mémoire

Attachement d'un segment de mémoire partagée

void *shmat(int id, void *adr, int option)

- l'appel applique (ou attache) un segment de mémoire partagée dans l'espace d'adressage du processus
- à partir de cette opération, toute lecture/écriture dans la zone dans laquelle le segment a été attaché est une lecture/écriture d'une zone de mémoire partagée
- id: identificateur retourné par l'appel shmget
- adr*:* adresse du début de la zone dans laquelle le segment est appliqué/attaché; si adr = NULL, le système choisit la première adresse disponible
- option: permet par exemple de spécifier la protection souhaitée (droits d'accès en lecture/écriture/exécution)
- l'appel retourne en résultat l'*adresse d'attachement*

Détachement d'un segment de mémoire partagée

int shmdt (void *adr)

- Supprimer la visibilité d'un processus sur un segment. Retourne zéro ou -1. Le paramètre adr est l'adresse d'attachement du segment, c'est-à-dire la valeur renvoyée par la primitive shmat.

Détachement et suppression

int msgctl (int id, int cmd, … arg)

- Le détachement n'affecte que la visibilité du segment depuis un processus et pas du tout son existence.
- Un segment pouvant être attaché à plusieurs processus, une suppression n'aura vraiment lieu que lorsque l'opération de contrôle *SHM_RMID* réalisé avec la primitive *shmctl* soit appelée.

Exemple

communication au moyen de segment partagé :

- Les deux programmes suivants communiquent au moyen d'un segment créé par le premier. Le segment de données est de clé 5. Seuls les processus du groupe peuvent y accéder. L'accès est exclusif.
- Le premier programme attache le segment créé à son espace de données 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 de données puis accède en lecture au segment. Ensuite, il détache le segment de son espace d'adressage.

Segments de données communs

Exemple

```
// programme shm1.cpp
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <iostream.h>
int main ( )
{ char * add;
int status, cle = 5;
if( (status = shmget(cle, sizeof(int), IPC_CREAT | IPC_EXCL | 0600))==-
   1)
exit(1);
cout << "status "<< status<< endl;
if((add =(char *) shmat(status,NULL,0)) == (char *)-1)
exit(2);
int* entier= (int *) add;
*entier = 1190;
sleep(2);
if( shmctl(status, IPC_RMID, NULL) == -1)
exit(3);
exit(0);
}
```

Segments de données communs
Exemple (suite)

```
// programme shm2.cpp
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <iostream.h>
int main ( )
{ char * add;
int status, cle = 5;
if( (status = shmget(cle, sizeof(int), 0))==-1)
exit(1);
cout << "status "<< status<< endl;
if((add =(char*) shmat(status,NULL,0)) == (char *) -1)
exit(2);
int* entier= (int *) add;
cout << "entier = " << *entier << endl;
if( shmctl(status, IPC_RMID, NULL) == -1)
exit(3);
exit(0);
}
```

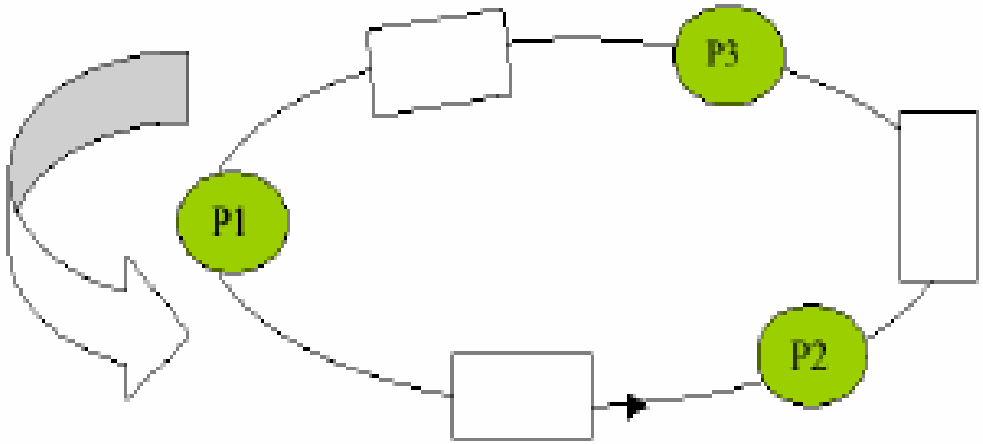
```
pascal> shm1 & shm2 &
[4] 10055
[5] 10056
status 788889600
status 788889600
entier = 1190
```

Mémoire partagée Posix.4

- Un segment partagé de mémoire est désigné sous Posix.4 comme un fichier (même si le segment ne se trouve pas dans l'espace des noms de fichiers, c-à-d n'est pas forcément visible par la commande ls)
- le nom doit commencer par /
- création (si n'existe pas) et ouverture: shm_open (retourne un descripteur de fichier)
- application dans l'espace virtuel du processus: mmap (cf slides suivants)
- fermeture:close
- libération: munmap
- destruction d'un segment de mémoire partagé: shm_unlink

Exercice

On veut établir, en utilisant les tubes anonymes (pipes), une communication de type anneau unidirectionnel entre trois processus fils. Pour ce faire, la sortie standard de l'un doit être redirigée vers l'entrée standard d'un autre, selon le schéma suivant :



Complétez le programme suivant en y ajoutant le code permettant de réaliser les redirections nécessaires à la création d'un tel anneau.

Corrections

```
int main ()
{
        /*1*/


    if (fork())  // création du premier processus
    {
        if(fork())
        {
            /*2*/
            if(fork())
            {       /*3*/

                    while (wait(NULL)>0);

                    /*4*/
            } else
            {     // processus P3
                    /*5*/
                                                    execlp("program3", "program3",NULL);

                                                    /*6*/
                                                }
                                            } else
                                            {     // processus  P2

                                                    /*7*/


                                                execlp("program2", "program2",NULL);

                                                    /*8*/
                                            }
        {   //processus P1

                    /*9*/



                execlp("program1","program1", NULL);

                    /*10*/
        }
            /*11*/
}
```

Exercice

Considérez le programme suivant qui a en entrée trois paramètres : deux fichiers exécutables et un nom de fichier. Ce programme crée deux processus pour exécuter les deux fichiers exécutables.

Complétez le code de manière à exécuter, l'un après l'autre, les deux fichiers exécutables et à rediriger les sorties standards des deux exécutables vers le fichier spécifié comme troisième paramètre. On récupérera ainsi dans ce fichier les résultats du premier exécutable suivis de ceux du deuxième.

```
int main(int argc, char* argv[])
{               /*0*/
if (fork()==0)
{               /*1*/
                execvp(argv[1], &argv[1]);
                /*2*/
}
/*3*/
if (fork()==0)
{               /*4*/
                execvp(argv[2], &argv[2]);
                /*5*/
}
/*6*/
}
```

Exercice 1

Le signal SIGCHLD est un signal qui est automatiquement envoyé par le fils à son père lorsque le fils se termine (par un exit, un return, ou autre).
Ajoutez une fonction et **le code nécessaire** pour que le père n'attende jamais son fils de façon bloquante et le fils ne devienne pas zombie.

```
/*0*/
int main(int argc, char *argv[])
{       /*1*/
        if (!fork())
        {               /*2*/
                        for (int i = 0 ; i <10 ; i++) ; //simule un petit calcul
                        /*3*/
                        exit(1) ;
                        /*4*/
        }/*5*/
        while(1) ; //Simule un calcul infini
                        /*6*/
}
```