IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

ANNEXE - Programmation système - Exemples

UE OSAP

Printemps 2024

Introduction

Ce document présente quelques *appels systèmes* Unix. Pour dire les choses simplement, un appel système est un moyen, pour un programme utilisateur, de solliciter un service auprès du système d'exploitation ; c'est une interface entre le *user space* et le *kernel space*.

Tout au long de ce document nous utiliserons le fil conducteur suivant : "Par quels moyens faire communiquer deux programmes ?". Pour cela nous évoquerons différentes techniques (plus ou moins heureuses) et les différents appels systèmes sous-jacents.

Notez que nous n'aborderons pas le mécanismes de sockets, qui fait l'objet d'un TP spécifique par ailleurs.

1 Utilisation des signaux Unix

Cet exemple montre comment manipuler les signaux Unix. Ce petit programme positionne tout d'abord un gestionnaire d'interruptions pour traiter la réception de certains signaux. Ensuite ce programme réalise ses activités courantes.

Ce programme a pour seule *activité courante* la charge d'afficher le caractère point `'.'` toutes les deux secondes. Ce délai de deux secondes est réalisé par la fonction `sleep(3)` (voir également `usleep(3)` et `nanosleep(2)`). Cette fonction peut également être interrompue à la réception d'un signal ; elle renvoie alors le nombre de secondes qu'il lui restait à dormir.

Un même *gestionnaire d'interruptions* est utilisé pour traiter différents signaux. On aurait pu évidemment positionner des gestionnaires différents pour chaque signal. Les signaux interceptés sont `SIGALRM`, `SIGUSR1`, `SIGHUP` et `SIGQUIT` ; le signal `SIGUSR2` est ignoré ; les autres signaux seront pris en compte par le gestionnaire par défaut qui provoque la terminaison brutale du processus.

Les signaux peuvent être envoyés à ce processus par la commande shell `kill(1)` (voir également la commande `killall(1)`). Ouvrir donc un premier terminal shell dans lequel vous exécutez le programme donné en exemple, et en parallèle ouvrir un second terminal shell dans lequel vous exécutez des commandes `kill`. Pour faciliter les choses, le programme donné en exemple commence par afficher sont numéro de processus (`pid`). Autrement, ce numéro pourrait être retrouvé par la commande shell `ps(1)`.

Vous êtes donc invités à essayer d'envoyer à ce programme différents signaux : `SIGALRM`, `SIGUSR1`, `SIGUSR2`, `SIGHUP`, `SIGQUIT`, `SIGTERM`, etc. Notez la réaction du programme de test.

Vous remarquerez également que le signal `SIGALRM` est envoyé d'une manière bien particulière. En effet, outre la manière classique d'envoyer des signaux avec `kill`, un processus à la possibilité de positionner une *alarme* qui demande au noyau de lui envoyer le signal `SIGALRM` dans x secondes. Ceci est obtenu par l'appel système `alarm(2)`.

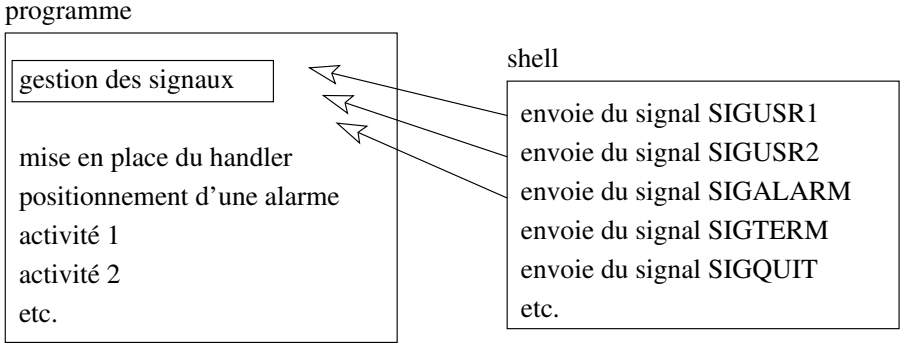

FIGURE 1 – Utilisation des signaux

Listing 1 – Utilisation des signaux

```c
#include <stdio.h>       /* Pour printf(3) et fflush(3) */
#include <signal.h>      /* Pour signal(2) */
#include <sys/types.h>   /* Pour le type pid_t */
#include <unistd.h>      /* Pour alarm(2) getpid(2) et sleep(3) */
#include <stdlib.h>      /* Pour exit(3) */

#define PERIOD  5


void sighandler(int signum) {
  printf("\nJe viens de recevoir le signal numero %d.\n", signum);
  switch (signum) {
  case SIGALRM:
  case SIGUSR1:
    printf("Alarme. La prochaine dans %d secondes.\n", PERIOD);
    alarm(PERIOD);
    break;
  case SIGHUP:
  case SIGQUIT:
    printf("Quit. Je me termine.\n");
    exit(0);
    break;
  default:
    printf("Aucun traitement prevu pour ce signal.\n");
  }
}


int main() {
  unsigned int t;
  pid_t pid;

  /* Positionnement du meme handler pour les signaux */
  /* SIGALRM, SIGUSR1, SIGHUP et SIGQUIT             */
  if ( signal(SIGALRM, sighandler) == SIG_ERR )
    fprintf(stderr, "Impossible de positionner le handler pour SIGALRM.\n");
  if ( signal(SIGUSR1, sighandler) == SIG_ERR )
    fprintf(stderr, "Impossible de positionner le handler pour SIGUSR1.\n");
  if ( signal(SIGHUP, sighandler) == SIG_ERR )
    fprintf(stderr, "Impossible de positionner le handler pour SIGUSR2.\n");
  if ( signal(SIGQUIT, sighandler) == SIG_ERR )
    fprintf(stderr, "Impossible de positionner le handler pour SIGQUIT.\n");
  /* J'ignore le signal SIGUSR2 */
  if ( signal(SIGUSR2, SIG_IGN) == SIG_ERR )
    fprintf(stderr, "Impossible d'ignorer SIGUSR2.\n");

  /* Utilisation de alarm : je demande au systeme de m'envoyer le signal */
  /* SIGALRM dans x secondes. */
  alarm(PERIOD);

  /* Affiche son pid pour aider l'utilisateur a faire des kill depuis */
  /* le shell. */
  pid = getpid();
  printf("Mon numero de processus est pid=%d.\n", pid);

  /* Maintenant que tout est en place, je peux commencer mes activites */
  while(1) {
    printf("."); fflush(NULL);
    t = sleep(2);
    if ( t != 0 )
      printf("Reveille alors qu'il me restait %d secondes de someil\n", t);
  }
}
```

Note. Dans cet exemple nous avons utilisé l'API ANSI C pour positionner nos gestionnaires d'interruptions. Cette API, relativement simple d'utilisation, peut voir son comportement varier suivant les systèmes Unix et les versions de la libc utilisés. On préférera donc utiliser l'API POSIX `sigaction(2)`. Il en sera ainsi dans les exemples suivants.

2 Création de processus et signaux

Cet exemple montre comment manipuler les signaux pour réaliser la synchronisation de deux processus s'échangeant des données travers un fichier. Tout d'abord, le processus (père) crée un autre processus (fils). Ensuite, les deux processus se synchronisent en utilisant les signaux de manière à se coordonner pour lire et écrire dans un fichier. Pensez observer ce qu'il se passe partir de commandes lancées dans le SHELL.

2.1 Création de processus

Le programme principal lance un processus fils. Ces deux processus sont sensibles au signal `SIGUSR1`. Ils affichent un message avant de se tuer la réception du signal `SIGUSR1`. Le fils fait une boucle infinie. Le père après une attente de 10 secondes (`sleep(3)`) envoie un signal son fils et fait ensuite une boucle infinie. Pour arrêter le père envoyez partir du SHELL un signal `SIGUSR1`.

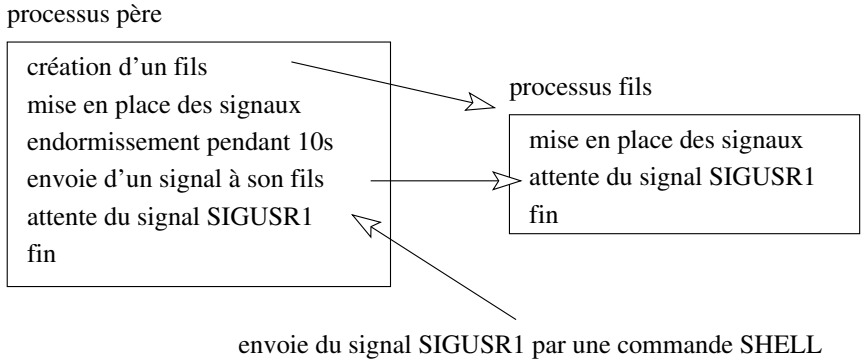

FIGURE 2 – Création d'un processus fils et synchronisation

Listing 2 – Création d'un processus fils et synchronisation

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <errno.h>
#include <signal.h>
#include <stdlib.h>

void handler_fils(int sig) {
  printf("Le␣fils␣recoit␣le␣signal␣SIGUSR1␣(%d)\n", sig);
  exit(0);
}

void handler_pere(int sig) {
  printf("Le␣pere␣recoit␣le␣signal␣SIGUSR1␣(%d)\n", sig);
  exit(0);
}

void handler_pere_fin_fils(int sig) {
  int status;
  pid_t fils;

  printf("Le␣pere␣recoit␣le␣signal␣SIGCHLD␣(%d)\n", sig);
  fils = wait(&status);
  if ( WIFEXITED(status) )
    printf("Le␣fils␣(%d)␣s'est␣termine␣normalement␣avec␣la␣valeur␣%d\n",
      fils, WEXITSTATUS(status) );
```

```
  if ( WIFSIGNALED(status) )
    printf("Le fils (%d) s'est termine sur le signal %d\n",
      fils , WTERMSIG(status) );
}


void pere(pid_t pid_fils) {
  sigset_t masque;
  struct sigaction action;

  /* Mise en place du masque de signaux */
  /* Tous les signaux sont ignores */
  if ( sigfillset(&masque) ) {
    perror("Le pere");
    exit(-1);
  }
  /* on devient sensible a SIGUSR1 (10) */
  if ( sigdelset(&masque, SIGUSR1) ) {
    perror("Le pere");
    exit(-1);
  }
  /* on devient sensible a SIGCHLD (17) */
  if ( sigdelset(&masque, SIGCHLD) ) {
    perror("Le pere");
    exit(-1);
  }
  /* prise en compte du masque */
  if ( sigprocmask(SIG_SETMASK, &masque, NULL) ) {
    perror("Le pere");
    exit(-1);
  }

  /* mise en place des handlers */
  action.sa_handler = handler_pere;
  action.sa_flags = SA_RESTART; /* SA_RESTART relance a la fin du handler */
                        /* les appels systemes interrompus par le signal */
  if ( sigaction(SIGUSR1, &action, NULL) ) {
    perror("Le pere");
    exit(-1);
  }
  action.sa_handler = handler_pere_fin_fils;
  if (sigaction(SIGCHLD, &action, NULL)) {
    perror("Le pere");
    exit(-1);
  }

  sleep(10);
  printf("Le pere (%d) envoie le signal SIGUSR1 a son fils (%d)\n",
          getpid(), pid_fils);
  kill(pid_fils, SIGUSR1);
  for(;;);
}


void fils() {
  sigset_t masque;
  struct sigaction action;

  /* Mise en place du masque de signaux */
  /* Tous les signaux sont ignores */
  if ( sigfillset(&masque) ) {
    perror("Le fils");
    exit(-1);
  }
  /* on devient sensible a SIGUSR1 (10) */
  if ( sigdelset(&masque, SIGUSR1) ) {
```

```
    perror("Le␣fils");
    exit(-1);
  }

  /* mise en place du handler */
  action.sa_handler = handler_fils;
  action.sa_flags = SA_RESTART;
  if (sigaction(SIGUSR1, &action, NULL)) {
    perror("Le␣fils");
    exit(-1);
  }

  for(;;);
}


int main() {
   pid_t pid;

   switch ( pid = fork() ) {
   case 0 :
     fils();
     break;
   case -1 :
     perror(NULL);
     break;
   default :
     pere(pid);
     break;
   }

   exit(0);
}
```

2.2 Échange d'information entre les 2 processus

Nous complétons le programme précédent afin de réaliser la communication de données par utilisation d'un fichier entre le processus père et le processus fils. Les deux processus réalisent un certain nombre de *ping-pong*.

Le père commence par écrire des données dans un fichier, il prévient ensuite son fils par un signal que les données sont prêtes. Le fils, qui s'était mis en attente d'un signal au moyen de l'appel système `pause(2)`, se débloque. Il lit les données et écrit ses propres données dans le fichier. Il prévient alors son père qu'il peut venir lire les données dans le fichier. Le père se débloque et vient lire les données du fils dans le fichier tampon et ainsi de suite.

Le père et le fils s'échangent un entier différent chaque communication. Une boucle réalise 1000 ping-pong. Nous mesurons le débit (d en Mb/s) en mesurant la date (appel système `time(2)`) avant la première écriture du père (t_d) et après la dernière lecture du père (t_f). La taille des données est la taille d'un entier soit 4 octets.
$d = \frac{(1000 \times 2 \times 4 \times 8) \times 10^{-6}}{t_f - t_d}$

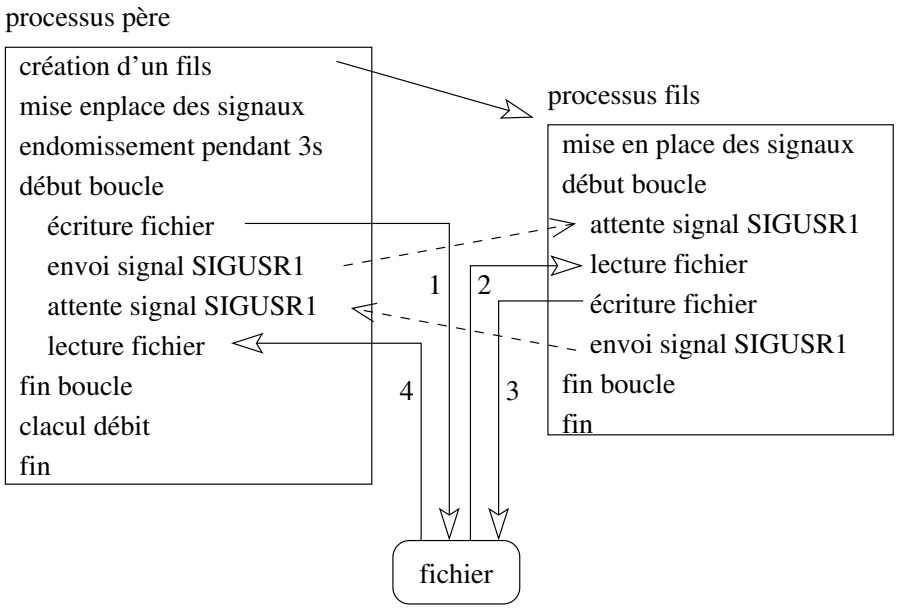

FIGURE 3 – Communication par fichier

Listing 3 – Communication par fichier

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <errno.h>
#include <signal.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <time.h>

#define MAX_PING          1000
#define PING_NAME         "prog3.dat"

pid_t pid_moi;
int i;

void handler_pere(int sig) {
  printf("Le pere (%d) est prevenu\n", pid_moi);
}

void handler_fils(int sig) {
  printf("Le fils (%d) est prevenu\n", pid_moi);
}

void handler_mortfils(int sig) {
  printf("Le pere (%d) apprend que son fils est mort\n", pid_moi);
  if ( i < MAX_PING-1 ) {
    printf("Le pere a fait %d pings\n", i);
    exit(-1);
  }
}


int main() {
  pid_t pid_fils, pid_pere;
  sigset_t masque;
  int offset;
  int ping_file;
  struct sigaction act;
```

```
time_t debut, fin;

/* mise en place des signaux */
if ( sigfillset(&masque) ) { perror(NULL); exit(-1); }
if ( sigdelset(&masque, SIGUSR1) ) { perror(NULL); exit(-1); }
if ( sigdelset(&masque, SIGCHLD) ) { perror(NULL); exit(-1); }
if ( sigprocmask(SIG_SETMASK, &masque, NULL) ) { perror(NULL); exit(-1); }

/* ouverture du fichier */
ping_file = open(PING_NAME, O_RDWR|O_CREAT|O_SYNC, 0666);
if ( ping_file == -1 ) { perror(NULL); exit(-1); }

/* fork */
switch ( pid_fils = fork() ) {
case 0 :                /* fils */
  pid_pere = getppid();
  pid_moi = getpid();
  act.sa_handler = handler_fils;
  act.sa_flags = SA_RESTART;
  if ( sigaction(SIGUSR1, &act, NULL) ) { perror("handler_fils"); exit(-1); }
  printf("Le fils %d a pour pere %d\n", pid_moi, pid_pere);
  for (i = 0; i < MAX_PING; i++) {
    printf("Le fils (%d) attend un signal\n", pid_moi);
    pause();  /* Je me met en attente d'un signal */
    lseek(ping_file, -sizeof(offset), SEEK_CUR);
    read(ping_file, (char *)&offset, sizeof(offset));
    printf("Le fils (%d) lit %d\n", pid_moi, offset);
    write(ping_file, (char *)&offset, sizeof(offset));
    printf("Le fils (%d) ecrit %d et previent le pere (%d)\n",
        pid_moi, offset, pid_pere);
    kill(pid_pere, SIGUSR1);
  }
  printf("Le fils a fini\n");
  break;
case -1 :      /* erreur */
  perror(NULL);
  exit(-1);
  break;
default :              /* pere */
  pid_moi = getpid();
  act.sa_handler = handler_pere;
  act.sa_flags = SA_RESTART;
  if ( sigaction(SIGUSR1, &act, NULL) ) { perror("handler_pere"); exit(-1); }
  act.sa_handler = handler_mortfils;
  if ( sigaction(SIGCHLD, &act, NULL) ) { perror("handler_mortfils"); exit(-1); }
  printf("Le pere %d a pour fils %d\n", pid_moi, pid_fils);
  sleep(3);
  offset = 0;
  debut = time(NULL);
  for (i = 0; i < MAX_PING; i++) {
    write(ping_file, (char *)&offset, sizeof(offset));
    printf("Le pere (%d) ecrit %d et previent le fils (%d) \n",
        pid_moi, offset, pid_fils);
    kill(pid_fils, SIGUSR1);
    printf("Le pere (%d) attend un signal\n", pid_moi);
    pause();
    lseek(ping_file, -sizeof(offset), SEEK_CUR);
    read(ping_file, (char *)&offset, sizeof(offset));
    printf("Le pere (%d) lit %d\n", pid_moi, offset);
    offset++;
  }
  fin = time(NULL);
  wait(NULL);
  printf("Le pere voit que le fils a termine\n");
  printf("Debit %fMb/s\n", ( MAX_PING * 2 * sizeof(offset) * 8 ) * 1E-6 / (fin-debut) );
  if ( unlink(PING_NAME) ) { perror(NULL); exit(-1); }
```

```
    break;
  }

  if ( close(ping_file) ) { perror(NULL); exit(-1); }
  exit(0);
}
```

Note : Ce programme comporte un énorme bug (pédagogique), bien que l'algorithme semble intuitivement "correct". En effet, lorsque l'on se met en attente d'un signal `SIGUSR1` pour être débloqué (à l'aide de l'appel système `pause()`), on attend un signal qui sera envoyé dans le futur. Cependant, il se peut que le partenaire l'ait envoyé avant même que l'on se mette en attente... Considérons par exemple le scénario suivant : le père envoit un signal au fils pour le débloquer, le fils fait tout son travail et envoie à son tour un signal au père pour le débloquer, qui malheureusement ne s'est pas encore mis en attente, ayant été suspendu par le scheduleur entre le `kill()` et le `wait()`. Les deux processus se retrouvent chacun en attente de l'autre : un inter-blocage. On peut exécuter ce programme des centaines de fois sans que ce bug ne survienne, mais il est bien présent.

3 Communication par tube

Nous reprenons le principe de l'exemple précédent. Cependant la méthode de communication est différente ici : nous allons utiliser les tubes.

Chaque processus renvoie le débit calculé. Notez que les tubes sont monodirectionnels et nous utilisons donc deux tubes : un pour envoyer et un pour recevoir. Afin de ne pas être tenté d'utiliser un tube dans les 2 sens de communication, nous fermerons les descripteurs ne vous servant pas.

3.1 Communication par tube ordinaire

Nous reprenons le principe du ping-pong en utilisant comme méthode de communication et de synchronisation uniquement le tube non nommé. Ceci est possible grâce au lien de parenté entre les deux processus. Les tubes simples sont créés par l'appel système `pipe(2)`.

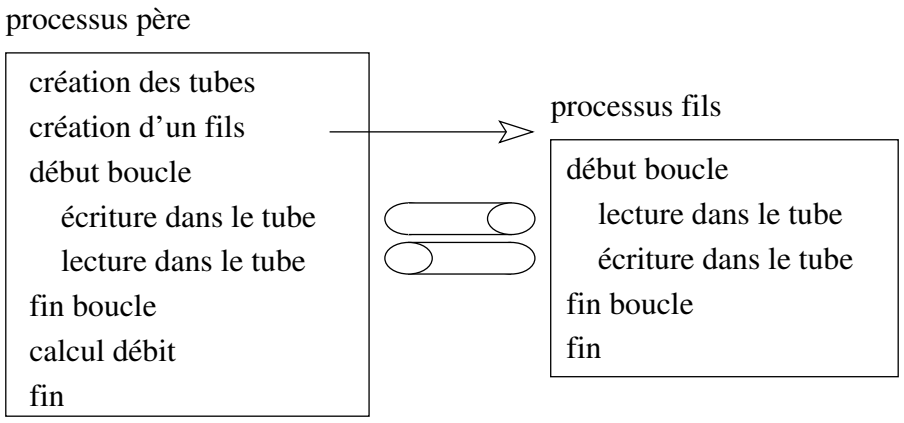

FIGURE 4 – Communication par tubes ordinaires

Listing 4 – Communication par tubes ordinaires

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <stdlib.h>
#include <time.h>

#define MAX_PING        1000

int main() {
  pid_t pid;
  int p2f[2];    /* pipe pere -> fils */
```

```
  int f2p[2];    /* pipe fils -> pere */
  time_t debut, fin;
  unsigned int i;
  unsigned int lu;

  /* creation des deux tubes */
  if ( pipe(p2f) ) { perror("p2f"); exit(-1); }
  if ( pipe(f2p) ) { perror("f2p"); exit(-1); }

  switch ( pid = fork() ) {
  case 0 :                /* fils */
    close(f2p[0]); close(p2f[1]);  /* on ferme les extremites inutiles */
    debut = time(NULL);
    for (i = 0; i < MAX_PING; i++) {
      read(p2f[0], &lu, sizeof(unsigned int));
      printf("Le fils lit %d\n", lu);
      printf("Le fils ecrit %d\n", i);
      write(f2p[1], &i, sizeof(unsigned int));
    }
    fin = time(NULL);
    printf("Debit cote fils : %f\n", (i * 4 * 8) * 1E-6 / (fin-debut) );
    close(p2f[0]); close(f2p[1]);
    break;
  case -1 :      /* erreur */
    perror(NULL);
    exit(-1);
    break;
  default :               /* pere */
    close(p2f[0]); close(f2p[1]);
    debut = time(NULL);
    for (i = 0; i < MAX_PING; i++) {
      printf("Le pere ecrit %d\n", i);
      write(p2f[1], &i, sizeof(unsigned int));
      read(f2p[0], &lu, sizeof(unsigned int));
      printf("Le pere lit %d\n", lu);
    }
    fin = time(NULL);
    wait(NULL);
    printf("Debit cote pere : %f\n", (i * 4 * 8) * 1E-6 / (fin-debut) );
    close(f2p[0]); close(p2f[1]);
  }
  exit(0);
}
```

3.2 Communication par tube nommé

On désire faire communiquer deux processus indépendants, sans relation de filiation. On utilise ici des tubes nommés qui permettent cela. Les tubes nommés sont des fichiers *spéciaux* et sont créés par la fonction `mkfifo(3)`.

Il nous faut écrire soit deux programmes ou bien, ce qui est mieux, un seul programme qui se comportera soit comme le processus A soit comme le processus B en fonction d'un attribut passé en ligne de commande. Vous noterez au passage l'utilisation de la fonction `getopt(3)` pour nous aider à parser la ligne de commande de manière élégante. Ouvrez deux terminaux shell, dans l'un fous exécuterez le programme avec l'option `-a`, puis dans l'autre vous exécuterez le même programme mais avec l'option `-b`.

Il faut faire attention au problème de synchronisation initiale. Normalement, l'ouverture d'une file FIFO est bloquante jusqu'à ce que l'autre côté soit aussi ouvert. Il faut donc faire attention à ce que le premier tube nommé ait ses extrémités ouvertes en lecture d'une part et en lecture d'autre par chacun des deux processus, avant de traiter le second tube. Si l'on croise (un processus ouvre le premier tube alors que l'autre processus ouvre l'autre tube) on va s'interbloquer, si on utilise bien cette ouverture bloquante, cela nous permet de synchroniser le démarrage de nos deux processus. Pour plus de détails, lire le man de `fifo(4)`.

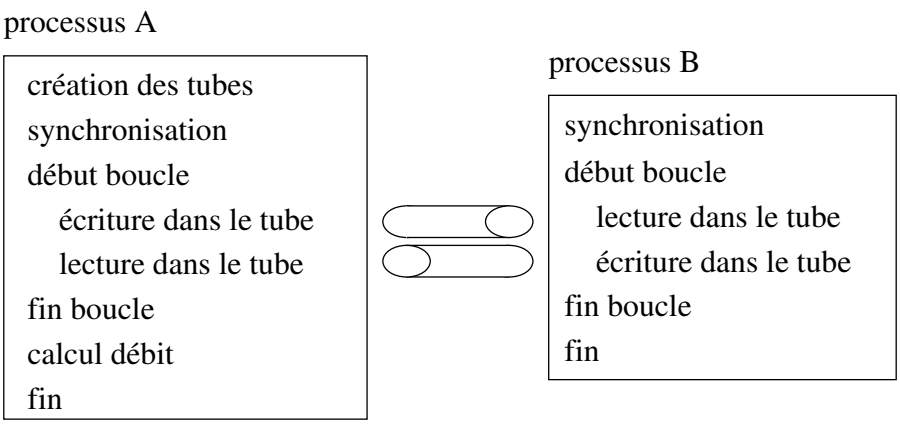

FIGURE 5 – Communication par tubes nommés

Listing 5 – Communication par tubes nommés

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <time.h>

#define MAX_PING        1000
#define A2B             "pipe.a2b"
#define B2A             "pipe.b2a"

enum behavior {A, B, undef};

int main(int argc, char *argv[]) {
  int a2b;      /* FIFO A -> B */
  int b2a;      /* FIFO B -> A */
  time_t debut, fin;
  unsigned int i;
  unsigned int lu;
  int c;
  enum behavior moi = undef;

  /* parser les arguments sur la ligne de commande */
  while ( ( c = getopt(argc, argv, "ab") ) != EOF )
    switch (c) {
      case 'a' : moi = A; break;
      case 'b' : moi = B; break;
    }
  for (c = optind; c != argc; c++)
    fprintf(stderr, "Warning: Unknown option %s\n", argv[c]);

  /* Actions suivant le comportement de A ou de B */
  switch ( moi ) {
  case A :              /* programme A */
    printf("Je suis le programme A\n");
    /* creation des deux tubes nommes (FIFO) */
    if ( mkfifo(A2B, S_IRUSR|S_IWUSR) != 0) { perror(A2B); exit(-1); }
    if ( mkfifo(B2A, S_IRUSR|S_IWUSR) != 0 ) { perror(B2A); exit(-1); }
    a2b = open(A2B, O_WRONLY);
    if ( a2b == -1 ) { perror(A2B); exit(-1); }
    b2a = open(B2A, O_RDONLY);
    if ( b2a == -1 ) { perror(B2A); exit(-1); }
    debut = time(NULL);
    for (i = 0; i < MAX_PING; i++) {
      printf("Le programme A ecrit %d\n", i);
      write(a2b, &i, sizeof(unsigned int));
```

```
      read(b2a, &lu, sizeof(unsigned int));
      printf("Le programme A lit %d\n", lu);
    }
    fin = time(NULL);
    printf("Debit cote A : %f\n", (i * 4 * 8) * 1E-6 / (fin-debut) );
    if ( unlink(A2B) ) { perror(NULL); exit(-1); }
    if ( unlink(B2A) ) { perror(NULL); exit(-1); }
    break;
  case B :                /* programme B */
    printf("Je suis le programme B\n");
    a2b = open(A2B, O_RDONLY);
    if ( a2b == -1 ) { perror(A2B); exit(-1); }
    b2a = open(B2A, O_WRONLY);
    if ( b2a == -1 ) { perror(B2A); exit(-1); }
    debut = time(NULL);
    for (i = 0; i < MAX_PING; i++) {
      read(a2b, &lu, sizeof(unsigned int));
      printf("Le programme B lit %d\n", lu);
      printf("Le programme B ecrit %d\n", i);
      write(b2a, &i, sizeof(unsigned int));
    }
    fin = time(NULL);
    printf("Debit cote B : %f\n", (i * 4 * 8) * 1E-6 / (fin-debut) );
    break;
  case undef :
    fprintf(stderr, "Error: vous devez choisir l'option -a ou -b\n");
    exit(-1);
  }
  close(b2a);
  close(a2b);
  exit(0);
}
```

4 Communication par file de messages IPC

Cet exemple reprend le principe des exemples précédents. C'est simplement la méthode de communication qui est différente. Ici, on va utiliser les files de messages IPC. Chaque processus renvoie le débit calculé.

4.1 Communication par messages

On désire faire communiquer deux processus indépendants, sans relation de filiation. On utilise ici les files de messages IPC qui permettent cela. Il s'agit d'un service de messagerie (texte) offert par les *Inter Processus Communication* (IPC).

Comme précédemment nous écrirons un seul programme mais qui adoptera soit le comportement du processus A soit celui du processus B en fonction d'un attribut passé en ligne de commande.

Nous utiliserons une seule file de messages, mais les messages seront typés de manière à indiquer s'ils sont de A pour B ou bien de B pour A. Chaque processus récupère par l'appel système `msgget(2)` un identifiant de messagerie à partir d'un numéro de clef commun. Une fois la file de message créé avec l'appel système `msgget(2)`, les messages sont postés avec `msgsnd(2)`, et reçus grâce à l'appel système `msgrcv(2)`. L'appel système `msgctl(2)` permet de contrôler la file de message, positionner des permissions, lire des informations comme le pid ou la date du dernier lecteur ou écrivain, ou encore pour supprimer la file.

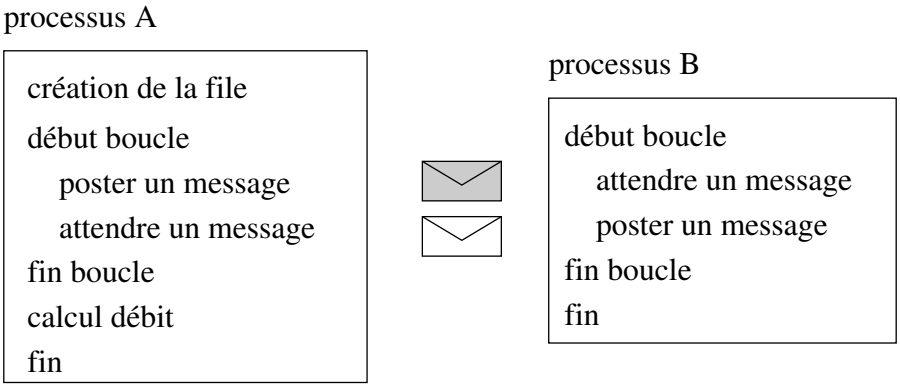

FIGURE 6 – Communication par messages

Listing 6 – Communication par messages

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>

#define MAX_PING        1000
#define MSGSIZE         4
#define MSG_A2B_t       1
#define MSG_B2A_t       2

struct msgbuf {
  long  mtype;          /* type de message ( > 0 ) */
  char  mtext[MSGSIZE]; /* contenu du message      */
};

enum behavior {A, B, undef};

int main(int argc, char *argv[]) {
  struct msgbuf msg_ecrit, msg_lu;
  int msgqid;
  time_t debut, fin;
  unsigned int i;
  int c;
  enum behavior moi = undef;
  key_t key = 20;
  char buff[10]; /* utilise pour la conversion int -> string[MSGSIZE] */

  /* parser les arguments sur la ligne de commande */
  while ( ( c = getopt(argc, argv, "abk:") ) != EOF )
    switch (c) {
      case 'a' : moi = A; break;
      case 'b' : moi = B; break;
      case 'k' : key = atol(optarg); break;
    }
  for (c = optind; c != argc; c++)
    fprintf(stderr, "Warning: Unknown option %s\n", argv[c]);

  /* Actions suivant le comportement de A ou de B */
  switch ( moi ) {
  case A :              /* programme A */
    /* recupere un identifiant sur ma file de messages */
    msgqid = msgget(key, IPC_CREAT|0660);
    if ( msgqid == -1 ) { perror("A msgget"); exit(-1); }
    printf("Je suis le programme A, j'utilise la clef 0x%x identifiant %d\n",
```

```
        key, msgqid);
    msg_ecrit.mtype = MSG_A2B_t;
    debut = time(NULL);
    for (i = 0; i < MAX_PING; i++) {
      sprintf(buff, "%03d", i);                    /* conversion int -> str */
      strncpy(msg_ecrit.mtext,buff,MSGSIZE-1);     /* on limite a MSGSIZE */
      msg_ecrit.mtext[MSGSIZE-1] = 0;              /* ca ralenti un peu */
      printf("Le programme A ecrit \"%s\"\n", msg_ecrit.mtext);
      if ( msgsnd(msgqid, &msg_ecrit, MSGSIZE, 0) == -1 )
          { perror("A msgsnd"); msgctl(msgqid, IPC_RMID, NULL); exit(-1); }
      msgrcv(msgqid, &msg_lu, MSGSIZE, MSG_B2A_t, 0);
      printf("Le programme A lit \"%s\"\n", msg_lu.mtext);
    }
    fin = time(NULL);
    printf("Debit cote A : %f\n", (i * MSGSIZE * 8) * 1E-6 / (fin-debut) );
    msgctl(msgqid, IPC_RMID, NULL);
    break;
  case B :                /* programme B */
    msgqid = msgget (key, IPC_CREAT|0660);
    if ( msgqid == -1 ) { perror("B msgget"); exit(-1); }
    printf("Je suis le programme B, j'utilise la clef 0x%x identifiant %d\n",
        key, msgqid);
    msg_ecrit.mtype = MSG_B2A_t;
    debut = time(NULL);
    for (i = 0; i < MAX_PING; i++) {
      msgrcv(msgqid, &msg_lu, MSGSIZE, MSG_A2B_t, 0);
      printf("Le programme B lit \"%s\"\n", msg_lu.mtext);
      sprintf(buff, "%03d", i);
      strncpy(msg_ecrit.mtext, buff, MSGSIZE-1);
      msg_ecrit.mtext[MSGSIZE-1] = 0;
      printf("Le programme B ecrit \"%s\"\n", msg_ecrit.mtext);
      if ( msgsnd(msgqid, &msg_ecrit, MSGSIZE, 0) == -1 )
          { perror("B msgsnd"); msgctl(msgqid, IPC_RMID, NULL); exit(-1); }
    }
    fin = time(NULL);
    printf("Debit cote B : %f\n", (i * MSGSIZE * 8) * 1E-6 / (fin-debut) );
    break;
  case undef :
    fprintf(stderr, "Error: vous devez choisir l'option -a ou -b\n"
          "Vous pouvez egalement choisir une clef avec l'option -k <id>\n");
    exit(-1);
  }
  exit(0);
}
```

Note : Messages texte. Les messages sont à priori de nature textuelle. Ce qui explique les conversions mise en œuvre dans l'exemple. Cependant, rien n'interdit formellement de tenter de s'échanger des informations binaires.

Note : File de messages POSIX <mqueue.h>. La norme POSIX définie également un système de messagerie interprocessus, à ne pas confondre avec le système de messagerie IPC. Les messages POSIX apportent le même type de service si ce n'est qu'il s'agit de files de messages *nommées* (comme des fichiers), et que nous avons en plus une notion de priorité. Les fonctions sur les files de messages POSIX sont : `mq_open(3RT)`, `mq_close(3RT)`, `mq_unlink(3RT)`, `mq_send(3RT)`, `mq_receive(3RT)`, `mq_notify(3RT)`, `mq_setattr(3RT)`. Les programmes doivent être compilés avec la librairie `librt`.

5 Communication par mémoire partagée et synchronisation par sémaphore

Cet exemple reprend le principe du ping-pong des exemples précédent. C'est simplement la méthode de communication qui est différente. Ici, on va utiliser de la mémoire partagée et des sémaphores pour la synchronisation. Chaque processus renvoie le débit calculé.

Même si nous utilisons qu'un seul bloc de mémoire partagée (une seule section critique), nous devons nous assurer que nos deux processus vont y accéder alternativement. Pour cela, nous avons besoin de deux sémaphores.

Selon la philosophie des sémaphores IPC, on associe un tableau de sémaphores à une clef. Nous utilisons donc un tableau de deux sémaphores. Nous aurions pu utiliser également deux tableaux de un sémaphore. Ceci n'aurait pas eu d'impact sur l'algorithme que nous avons. Par ailleurs nous utilisons nos sémaphores comme de simples mutex.

Après avoir créé un segment de mémoire partagé avec `shmget(2)`, chaque processus doit l'attacher dans son espace d'adressage avec `shmat(2)`. Après utilisation, ils devront le détacher avec `shmdt(2)` avant de supprimer le segment avec `shmctl(2)`. Les sémaphores sont créés avec `semget(2)`. Les processus réalisent une ou plusieurs opérations de manière atomique sur un tableau de sémaphore avec `semop(2)`, et sont mis en attente ou réveillés suivant les besoins. L'appel système `semctl(2)` permet d'initialiser la valeur d'un sémaphore, de la lire, de détruire un ensemble de sémaphore, etc.

Vous noterez au passage l'utilisation de `atexit(3)` qui permet d'enregistrer une procédure pour être exécutée à la terminaison du processus.

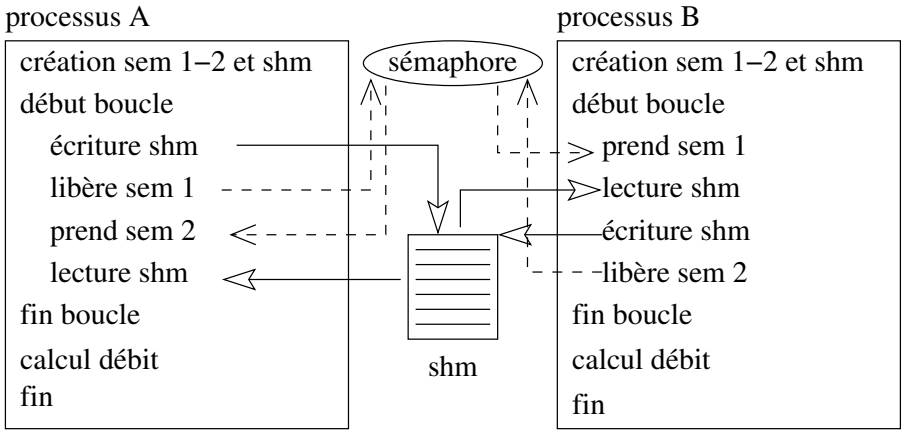

FIGURE 7 – Communication par mémoire partagée

Listing 7 – Communication par mémoire partagée

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <time.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>

#define MAX_PING        1000

enum behavior {A, B, undef};


#if defined(__GNU_LIBRARY__) && !defined(_SEM_SEMUN_UNDEFINED)
/* l'union semun est definie par l'inclusion de <sys/sem.h> */
#else
/* d'apres X/OPEN il faut la definir nous-memes */
union semun {
       int val;                    /* valeur pour SETVAL */
       struct semid_ds *buf;       /* buffer pour IPC_STAT, IPC_SET */
       unsigned short *array;      /* table  pour GETALL, SETALL */
                                   /* Specificite Linux : */
       struct seminfo *__buf;      /* buffer pour IPC_INFO */
};
#endif


int *shm; /* pointeur sur le bloc de memoire partagee */
```

```
int semid, shmid;

/* procedure executee a la terminaison des programmes */
void detruit_shm_ptr() {
  if ( shmdt(shm) == -1 ) perror("shmdt");
}

/* procedure executee a la terminaison du programme A seulement */
void detruit_shm() {
  if ( shmctl(shmid, IPC_RMID, NULL) == -1 ) perror("shmctl IPC_RMID");
}

/* procedure executee a la terminaison du programme A seulement */
void detruit_sem() {
  if ( semctl(semid, 0, IPC_RMID) == -1 ) perror("semctl IPC_RMID");
}


int main(int argc, char *argv[]) {
  time_t debut, fin;
  unsigned int i;
  int c;
  enum behavior moi = undef;
  key_t key_shm = 20;
  key_t key_sem = 20;
  struct sembuf sopsPV[2], sopsVP[2]; /* operations sur nos semaphores */
  union semun arg; /* argument pour semctl */


  /* parser les arguments sur la ligne de commande */
  while ( ( c = getopt(argc, argv, "abm:s:") ) != EOF )
    switch (c) {
      case 'a' : moi = A; break;
      case 'b' : moi = B; break;
      case 'm' : key_shm = atol(optarg); break;
      case 's' : key_sem = atol(optarg); break;
    }
  for (c = optind; c != argc; c++)
    fprintf(stderr, "Warning: Unknown option %s\n", argv[c]);

  /* Deux types d'operations sur nos semaphores */
  /* Operations P() et V() des mutex de Dijkstra */
  /* Nous ferons de maniere atomique P(sem0) et V(sem1) */
  /* Ainsi que V(sem0) et P(sem1) */
  /* sem0 controle la bascule A->B, et sem1 la bascule B->A */
  sopsPV[0].sem_num = 0; /* operation sur sem0 */
  sopsPV[0].sem_op = -1; /* operation P() pour prendre la ressource */
  sopsPV[0].sem_flg = 0;
  sopsPV[1].sem_num = 1; /* operation sur sem1 */
  sopsPV[1].sem_op = +1; /* operation V() pour liberer la ressource */
  sopsPV[1].sem_flg = 0;
  sopsVP[0].sem_num = 0; /* operation sur sem0 */
  sopsVP[0].sem_op = +1; /* operation V() pour liberer la ressource */
  sopsVP[0].sem_flg = 0;
  sopsVP[1].sem_num = 1; /* operation sur sem1 */
  sopsVP[1].sem_op = -1; /* operation P() pour prendre la ressource */
  sopsVP[1].sem_flg = 0;

  /* Actions suivant le comportement de A ou de B */
  switch ( moi ) {
  case A :                /* programme A */
    /* Creer le tableau de 2 semaphores */
    semid = semget(key_sem, 2, IPC_CREAT|0660);
    if ( semid == -1 ) { perror("A semget"); exit(-1); }
    atexit(detruit_sem);
```

```
    /* positionner les valeurs initiales des semaphores */
    /* seul le programme A le fait */
    arg.val = 0; /* au debut la ressource A->B est prise par A */
    if ( semctl(semid, 0, SETVAL, arg) == -1 ) {perror("sem0=0"); exit(-1); }
    arg.val = 1; /* au debut la ressource A->B est libre */
    if ( semctl(semid, 1, SETVAL, arg) == -1 ) {perror("sem1=1"); exit(-1); }

    /* creer le segment de memoire partagee */
    shmid = shmget(key_shm, sizeof(int), IPC_CREAT|0660);
    if ( shmid == -1 ) { perror("A shmget"); exit(-1); }
    atexit(detruit_shm);
    shm = shmat(shmid, NULL, 0);
    if ( shm  == -1 ) { perror("A shmat"); exit(-1); }
    atexit(detruit_shm_ptr);

    printf("Je suis le programme A; sem: key=0x%x id=%d; shm: key=0x%x id=%d\n",
        key_sem, semid, key_shm, shmid);

    debut = time(NULL);
    for (i = 0; i < MAX_PING; i++) {
      printf("Le programme A ecrit %d\n", *shm = i);
      /* libere la ressource et fait la bascule A->B */
      /* en meme temps demande la bascule B->A pour acceder a la ressource */
      if ( semop(semid, sopsPV, 2) == -1 ) { perror("P(0)+V(1)"); exit(-1); }
      printf("Le programme A lit %d\n", *shm);
    }
    fin = time(NULL);

    printf("Debit cote A : %f\n", (i * 4 * 8) * 1E-6 / (fin-debut) );
    break;

  case B :                /* programme B */
    semid = semget(key_sem, 2, IPC_CREAT|0660);
    if ( semid == -1 ) { perror("B semget"); exit(-1); }

    shmid = shmget(key_shm, sizeof(int), IPC_CREAT|0660);
    if ( shmid == -1 ) { perror("B shmget"); exit(-1); }
    shm = shmat(shmid, NULL, 0);
    if ( shm  == -1 ) { perror("B shmat"); exit(-1); }
    atexit(detruit_shm_ptr);

    printf("Je suis le programme B; sem: key=0x%x id=%d; shm: key=0x%x id=%d\n",
        key_sem, semid, key_shm, shmid);

    debut = time(NULL);
    for (i = 0; i < MAX_PING; i++) {
      /* libere la ressource et fait la bascule B->A */
      /* en meme temps demande la bascule A->B pour acceder a la ressource */
      if ( semop(semid, sopsVP, 2) == -1 ) { perror("V(0)+P(1)"); exit(-1); }
      printf("Le programme B lit %d\n", *shm);
      printf("Le programme B ecrit %d\n", *shm = i);
    }
    fin = time(NULL);

    printf("Debit cote B : %f\n", (i * 4 * 8) * 1E-6 / (fin-debut) );
    break;

  case undef :
    fprintf(stderr, "Error: vous devez choisir l'option -a ou -b\n");
    exit(-1);
  }
  exit(0);
}
```

Note : Mémoire partagée POSIX. La norme POSIX permet également de partager de la mémoire entre processus. Il s'agit essentiellement d'une variation de la technique de projection en mémoire des fichiers avec `mmap(2)` : on utilise alors `shm_open(3)` et `shm_unlink(3)` à la place de `open()` et `close()`.

Note : Sémaphores POSIX `<semaphore.h>`. La norme POSIX définie également un mécanisme de synchronisation interprocessus par sémaphores. Au contraire des sémaphores IPC, une structure de sémaphore POSIX définie un simple et unique sémaphore, et non pas un tableau de sémaphores. Par ailleurs, selon la philosophie POSIX les sémaphores sont identifiés par un nom plutôt que par un numéro de clef. Les opérations sur les sémaphores POSIX sont : `sem_open(3RT)`, `sem_init(3RT)`, `sem_close(3RT)`, `sem_unlink(3RT)`, `sem_destroy(3RT)`, `sem_getvalue(3RT)`, `sem_wait(3RT)`, `sem_trywait(3RT)`, `sem_post(3RT)`.

6 Multi-threading

Cet exemple présente la programmation des threads. Nous reprenons le programme précédent, mais cette fois-ci les threads vont réaliser un ping-pong en utilisant la mémoire locale. La synchronisation se fera en utilisant les mutex dédiés aux threads. Nous calculerons aussi le débit.

Un thread est créé par `pthread_create(3)`. Il s'achève avec un `pthread_exit(3)`. Un thread peut être rejoint par un autre (i.e. on attend sa terminaison) par `pthread_join(3)`.

Un mutex est créé par `pthread_mutex_init(3)` et détruit par `pthread_mutex_destroy(3)`. Un mutex peut être pris par un thread avec `pthread_mutex_lock(3)` ou `pthread_mutex_trylock(3)`, et relâché avec `pthread_mutex_unlock(3)`.

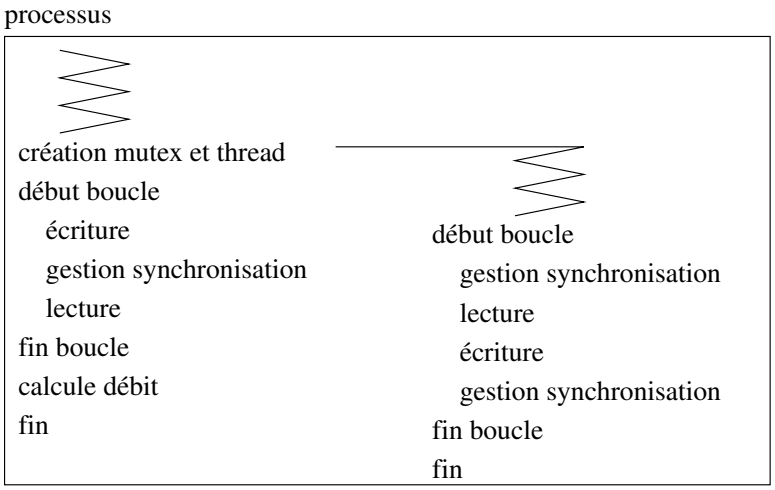

FIGURE 8 – Communication par mémoire locale entre deux threads

Listing 8 – Communication par mémoire locale entre deux threads

```
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <time.h>

#define MAX_PING          1000


pthread_mutex_t mutex_a2b, mutex_b2a; /* les mutex */
int shm; /* la memoire partagee entre les deux threads */


/* Le thread du second flot d'execution B */
void le_thread() {
  unsigned int i;
  for (i = 0; i < MAX_PING; i++) {
    /* demande la ressource et attend la bacule A->B */
    if ( pthread_mutex_lock(&mutex_a2b) )
```

```
        { perror("lock mutex_a2b"); exit(-1); }
    printf("Le thread B lit %d\n", shm);
    printf("Le thread B ecrit %d\n", shm = i);
    /* libere la ressource et fait la bascule B->A */
    if ( pthread_mutex_unlock(&mutex_b2a) )
        { perror("unlock mutex_b2a"); exit(-1); }
  }
  pthread_exit(NULL);
}


int main() {
  time_t debut, fin;
  unsigned int i;
  pthread_t tid;

  /* initialisation des mutex */
  pthread_mutex_init(&mutex_a2b, NULL);
  pthread_mutex_init(&mutex_b2a, NULL);
  if ( pthread_mutex_lock(&mutex_b2a) )
      { perror("lock mutex_b2a"); exit(-1); }

  /* creation du thread pour lancer un second flot d'execution */
  if ( pthread_create(&tid, NULL, le_thread, NULL) )
      { perror("pthread_create"); exit(-1); }

  /* Le flot d'execution principal A */
  debut = time(NULL);
  for (i = 0; i < MAX_PING; i++) {
    printf("Le thread A ecrit %d\n", shm = i);
    /* libere la ressource et fait la bascule A->B */
        if ( pthread_mutex_unlock(&mutex_a2b) )
        { perror("unlock mutex_a2b"); exit(-1); }
    /* demande la ressource et attend la bascule B->A */
    if ( pthread_mutex_lock(&mutex_b2a) )
        { perror("lock mutex_b2a"); exit(-1); }
    printf("Le thread A lit %d\n", shm);
  }
  fin = time(NULL);
  if ( pthread_join(tid, NULL) )
      { perror("join"); exit(-1); }

  printf("Debit cote A : %f\n", (i * 4 * 8) * 1E-6 / (fin-debut) );

  exit(0);
}
```

Note : threads Solaris `<thread.h>`. Solaris, grand précurseur dans le domaine, avait également défini son propre système de gestion des threads. Notons qu'aujourd'hui les threads POSIX sont également disponibles sous Solaris. Pour des raisons de probabilité nous préférerons donc utiliser les threads POSIX.

7 Une curiosité

Nous avons abordé différentes techniques pour partager de l'information entre deux processus (sans aborder les sockets qui font l'objet d'un autre TP). Pour être exhaustif sur le sujet, et bien que ce soit tiré par les cheveux, évoquons le cas de `ptrace()`.

Une différence notable entre les processus légers (threads) et les processus lourds, c'est que les premiers partagent le même espace mémoire, alors que les seconds voient leurs espaces mémoires bien cloisonnés... mais pas toujours. L'exception vient avec l'appel système `ptrace()` (voir le `man 2 ptrace`) qui permet à un processus de s'attacher à un autre (tout en devenant son père) : cela lui permet de contrôler son exécution et d'accéder à son espace mémoire.

Cela est utilisé à des fins de débuggage (outils `gdb strace` etc.), voir pour du reverse engineering [1], mais

1. Nicolas Bareil, *"Playing with ptrace() for fun and profit - Injection de code sous Linux"*, SSTIC 2006, `http://actes.sstic.`

rârement pour de la communication entre processus (d'ailleurs, cet appel système pose des problèmes de probabilité).

8 Lectures conseillées

- *Advanced Linux Programming*

  By Mark Mitchell, Jeffrey Oldham, and Alex Samuel, of CodeSourcery LLC

  Published by New Riders Publishing

  ISBN 0-7357-1043-0

  First Edition, June 2001

  `http://www.advancedlinuxprogramming.com`

- *Linux Device Drivers*

  By Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman

  Published by O'Reilly

  ISBN 0-596-00590-3

  Third Edition February 2005

  `http://lwn.net/Kernel/LDD3/`

`org/SSTIC06/Playing_with_ptrace/`