# TD 6 : Processus, fork, signaux

**Objectifs pédagogiques :**

- **Communications IPC : shm, sem**
- **Tube et Tube nommé**

### Introduction

Dans ce TD, nous allons étudier l'API proposée par POSIX pour la communication entre processus. Il faudra donc un système POSIX (en particulier pas windows) pour compiler et exécuter nos programmes.

Le standard POSIX définit une API en C à bas niveau pour l'interaction entre les processus incluant des tubes, des segments de mémoire partagée, et des sémaphores.

## 2 Tube anonyme

**Question 1.** A l'aide de la primitive *pipe*, écrivez un programme $P$ qui crée deux fils $F1$ et $F2$, tous les processus doivent afficher le pid des trois processus avant de se terminer proprement.

Donc pour $P$ et $F2$, facile de connaitre les autres pid.

Le père P n'a qu'à stocker les pid de ses fils (obtenus par fork).

Le deuxième fils $F2$ peut hériter de son père le pid de $F1$ et utiliser e.g. *getppid*() pour trouver $P$.

Le premier fils $F1$, on a un souci, comment lui communiquer le pid de son petit frère ? On ne dispose pas d'API pour le retrouver (pas de *getsiblingpid*...).

Donc on utilise "pipe".

" *pipe*() creates a pipe, a unidirectional data channel that can be used for interprocess communication.

The array pipefd is used to return two file descriptors referring to the ends of the pipe.

pipefd[0] refers to the read end of the pipe. pipefd[1] refers to the write end of the pipe.

Data written to the write end of the pipe is buffered by the kernel until it is read from the read end of the pipe. "

Voir aussi "man 7 pipe".

Une fois qu'on a fait "pipe", on obtient deux descripteurs de fichiers (des int), avec lesquels on va utiliser l'API standard sur fd : read(fd, addresse, taille), write(fd, addresse, taille), et close(fd).

Pour être propre, on ferme toutes les extrémités que l'on n'utilise pas.

troisProc.cpp

```cpp
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>
#include <iostream>

int main3 () {
      pid_t P,F1,F2;

      P = getpid();

      int pdesc[2];
      if (pipe(pdesc) != 0) {
            perror("pipe error");
            return 1;
      }

```

```
      F1 = fork();
      if (F1 < 0) {
            perror("fork error");
            return 1;
      } else if (F1 == 0){
            F1 = getpid();
            // F1
            close(pdesc[1]);
            if (read(pdesc[0],&F2,sizeof(pid_t)) == sizeof(pid_t)) {
                  std::cout << "Je suis :" << getpid() << " P=" << P << " F1=" << F1 <<
                      " F2=" << F2 << std::endl;
            } else {
                  perror("erreur read");
                  return 1;
            }
            close(pdesc[0]);
            return 0;
      }

      F2 = fork();
      if (F2 < 0) {
            perror("fork error");
            return 1;
      } else if (F2 == 0){
            // F2
            F2 = getpid();
            close(pdesc[0]);
            if (write(pdesc[1],&F2,sizeof(pid_t)) == sizeof(pid_t)) {
                  std::cout << "Je suis :" << getpid() << " P=" << P << " F1=" << F1 <<
                      " F2=" << F2 << std::endl;
            } else {
                  perror("erreur write");
                  return 1;
            }
            close(pdesc[1]);
            return 0;
      }

      close(pdesc[0]);
      close(pdesc[1]);

      std::cout << "Je suis :" << getpid() << " P=" << P << " F1=" << F1 << " F2=" << F2
           << std::endl;

      wait(0);
      wait(0);

      return 0;
}
```

## 3 Tube nommé

**Question 1.** Ecrivez un programme *writer* qui prend un chemin en argument, y crée un tube nommé, puis attend que l'utilisateur saisisse du texte et le recopie dans le tube. Sur controle-C (SIGINT) le programme se ferme proprement sans laisser de traces.

**Question 2.** Ecrivez un programme *reader* qui prend un chemin en argument correspondant à un tube, puis lit le texte envoyé par writer dans le tube. Le programme se termine quand il n'y a plus d'écrivain sur le tube.

Donc ce dont on a besoin :

- mkfifo : crée une fifo (tube nommé)
- open/close : comme si c'était un fichier
- read/write : idem
- unlink : efface du système de fichier

Les soucis à traiter :

- mkfifo + tester que ça marche + droits pour l'usager $S\_IRUSR \| S\_IWUSR$.
- open => $O\_RDONLY$ ou $O\_WRONLY$ selon l'extrémité
- lire ligne à ligne l'entrée standard = facile avec std::cin et une std::string
- Transmettre des chaînes de caractère = souci pour la taille de ce qui passe. On va écrire la taille de la chaîne (un size_t) puis son contenu au format chaîne du C.
- Symétriquement le lecteur va lire une taille puis "taille" octets.
- Il n'y a pas de boucles sur read ni write dans le corrigé, donc les grosses écritures vont mal se passer.
- On accroche un "unlink"+exit au signal Ctrl-C

writer.cpp

```cpp
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <iostream>
#include <string>
#include <csignal>

char * path;

void handleCtrlC (int sig) {
	unlink(path);
	exit(0);
}

int main4 (int argc, char ** argv) {
	if (argc < 2) {
		std::cerr << "Provide a name for the pipe." << std::endl;
	}

	if ( mkfifo(argv[1],S_IRUSR|S_IWUSR) != 0) {
		perror("mkfifo problem");
		return 1;
	}
	path = argv[1];

	int fd = open(argv[1],O_WRONLY);
	if (fd < 0) {
		perror("open problem");
		return 1;
	}

	signal(SIGINT, handleCtrlC);


```

```
      while (true) {
            std::string s;
            std::cin >> s;
            size_t sz = s.length()+1;
            if (sz > 1) {
                  write (fd, &sz, sizeof(sz));
                  write (fd, s.c_str(), sz);
            }
      }

      return 0;
}
```

reader.cpp

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <iostream>
#include <string>


int main2 (int argc, char ** argv) {
      if (argc < 2) {
            std::cerr << "Provide a name for the pipe." << std::endl;
      }

      int fd = open(argv[1],O_RDONLY);
      if (fd < 0) {
            perror("open problem");
            return 1;
      }

      while (true) {
            size_t sz;
            int rd = read(fd,&sz,sizeof(sz));
            if (rd == 0) {
                  break;
            } else if (rd != sizeof(sz)) {
                  perror("read error");
                  return 1;
            }
            char buff [sz];
            rd = read(fd,&buff,sz);
            if (rd == 0) {
                  break;
            } else if (rd != sz) {
                  perror("read error");
                  return 1;
            }
            std::cout << buff << std::endl;
      }

      return 0;
}
```

**Question 3.** Que se passe-t-il si on a plusieurs lecteurs ou plusieurs écrivains ?

Plusieurs écrivains = aucun problème, les écritures de petite taille (BUF_SIZE vaut 4096 sur ma machine) sont atomiques. MAIS avec le protocole tel qu'il est fait, on n'écrit pas atomiquement (deux appels à write). Pour que ça se passe bien, une seule écriture est nécessaire. On pourrait assez facilement le faire en préparant un buffer qui contienne la taille ET la string.

Plusieurs lecteurs, si on a forcé une seule écriture par message, ça devrait bien se passer, mais attention, un seul lecteur peut voir les messages à la fois la lecture est destructrice.

De plus on ne peut pas faire un seul read, car il faut lire la taille avant de lire le contenu. Donc on devrait passer vers un modèle où les messages sont de taille fixe pour permettre le mode multi-lecteur, tel quel on est obligé de faire deux lecture.

Morale : ça marchotte, mais ce n'est pas terrible, si on a des paquets de taille fixe, c'est plus envisageable d'avoir plusieurs lecteurs.

## 4 Sémaphore

**Question 1.** On considère une application constituée de deux processus P1 et P2. A l'aide de l'API sémaphore, réaliser une alternance, où le processus P1 affiche "Ping", et le processus P2 "Pong".

Donc il faut deux semaphores, avec un seul, on va s'auto-libérer en boucle.

P1 libère P2, qui libère P1 ...

On les laisse réfléchir, et on invite la discussion sur "un seul sémaphore, A fait P puis V, B fait V puis P". Non, ça ne marche pas.

Donc bien 2 sémaphores.

pingpong.cpp

```cpp
#include <fcntl.h> /* For O_* constants */
#include <sys/stat.h> /* For mode constants */
#include <semaphore.h>
#include <sys/wait.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <iostream>


int main() {

      // create mutex initialisé 0
      sem_t * smutex1 = sem_open("/monsem1", O_CREAT |O_EXCL | O_RDWR , 0666, 0);
      if (smutex1==SEM_FAILED) {
            perror("sem create");
            exit(1);
      }
      // create mutex initialisé 0
      sem_t * smutex2 = sem_open("/monsem2", O_CREAT |O_EXCL | O_RDWR , 0666, 0);
      if (smutex2==SEM_FAILED) {
            perror("sem create");
            exit(1);
      }

      pid_t f = fork();
      if (f==0) {
            // fils
```

```
            for (int i=0; i < 10; i++) {
                  sem_wait(smutex1);
                  std::cout << "Ping" << std::endl;
                  sem_post(smutex2);
            }
            sem_close(smutex1);
            sem_close(smutex2);
      } else {
            // pere
            for (int i=0; i < 10; i++) {
                  sem_post(smutex1);
                  sem_wait(smutex2);
                  std::cout << "Pong" << std::endl;
            }
            sem_close(smutex1);
            sem_close(smutex2);

            // on nettoie
            sem_unlink("/monsem1");
            sem_unlink("/monsem2");
            wait(nullptr);
      }
      return 0;
}
```

On va traiter ici directement avec l'API pour créer des sémaphores (nommés), sans déployer un segment partagé. Cela signifie que l'on crée avec *sem_open* et un nom de fichier (qui DOIT commencer par / puis ne pas contenir de slash, et n'être pas trop long), comme on ouvrirait une inode dans le filesystem. On crée un objet dans le système, qu'il faut nettoyer à l'aide de unlink.

**Question 2.** On considère à présent $N$ processus qui doivent s'alterner (circulairement). Combien de sémaphores faut-il introduire ? Proposez une réalisation de ce comportement.

Donc il faut $N$ semaphores, et cadrer tout ça avec des boucles. Pour plus d'homogénéité, je laisse les $N$ fils jouer entre eux, la version a deux processus c'était père vs fils.

On pourrait ajouter un numéro Ping1, Ping2... pour mieux constater l'alternance.

On crée un nouveau nom unique par semaphore.

pingNpong.cpp

```
#include <fcntl.h> /* For O_* constants */
#include <sys/stat.h> /* For mode constants */
#include <semaphore.h>
#include <sys/wait.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <iostream>


#define N 4

int main() {

      sem_t * mutex [N];

      for (int i=0; i < N ; i++) {
```

```
            // create mutex initialisé 0 sauf le premier
            int semval = 0;
            if (i==0) {
                  semval = 1;
            }
            std::string semname = "/monsem"+std::to_string(i);
            mutex[i] = sem_open(semname.c_str(), O_CREAT |O_EXCL | O_RDWR , 0666, semval
                );
            if (mutex[i]==SEM_FAILED) {
                  perror("sem create");
                  exit(1);
            }
      }

      for (int i=0; i < N ; i++) {
            pid_t f = fork();
            if (f==0) {
                  // fils
                  for (int round=0; round < 10; round++) {
                        sem_wait(mutex[i]);
                        std::cout << "Ping" << i << std::endl;
                        sem_post(mutex[(i+1)%N]);
                  }
                  return 0;
            }
      }

      for (int i=0; i < N ; i++) {
            wait(nullptr);
      }

      // on nettoie
      for (int i=0; i < N ; i++) {
            sem_unlink( ("/monsem"+std::to_string(i)).c_str() );
      }
      return 0;
}
```

Remarque importante : Si on réfléchit en termes de l'API thread, il nous faut N condition, donc aussi N mutex pour obtenir ce comportement. Le point important est que A fait P et UN AUTRE processus B fait le V, ce qui n'est pas possible avec un mutex. Donc on a des sémaphores dont le compteur oscille entre 0 et 1, mais ce ne sont pas des équivalents à des mutex ici.

## 5 Mémoire partagée

On considère un programme qui manipule une pile partagée entre plusieurs processus.
On fournit la base de code suivante :

Stack.h

```
#pragma once

#include <cstring> // size_t,memset

namespace pr {

#define STACKSIZE 100
```

```

template<typename T>
class Stack {
      T tab [STACKSIZE];
      size_t sz;
public :
      Stack () : sz(0) { memset(tab,0,sizeof tab) ;}

      T pop () {
            // bloquer si vide
            T toret = tab[--sz];
            return toret;
      }

      void push(T elt) {
            //bloquer si plein
            tab[sz++] = elt;
      }
};

}
```

prodcons.cpp

```
#include "Stack.h"
#include <iostream>
#include <unistd.h>
#include <sys/wait.h>
#include <vector>


using namespace std;
using namespace pr;

void producteur (Stack<char> * stack) {
      char c ;
      while (cin.get(c)) {
            stack->push(c);
      }
}

void consomateur (Stack<char> * stack) {
      while (true) {
            char c = stack->pop();
            cout << c << flush ;
      }
}

int main () {
      Stack<char> * s = new Stack<char>();

      pid_t pp = fork();
      if (pp==0) {
            producteur(s);
            return 0;
      }

      pid_t pc = fork();
      if (pc==0) {
            consomateur(s);
            return 0;
```

```
    }

    wait(0);
    wait(0);

    delete s;
    return 0;
}
```

**Question 1.** En se limitant aux seuls semaphores, modifier le Stack pour que : pop bloque si vide, push bloque si plein, et que les données partagées *sz* et *tab* soit protégées des accès concurrents. On suppose que le Stack est alloué dans un segment de mémoire partagée, et qu'il sera partagé entre processus.

On les laisse réfléchir, on a ici une vraie question du type que l'on poserait volontiers en examen.

Les précisions sur le contexte "mémoire partagée" et "entre processus" affecte la façon de créer et initialiser les semaphores. Dans ce contexte, on sait qu'on peut jouer la carte OO, et loger des semaphores dans la classe elle même. On a ici un modèle de classe C++ dont les instances sont multi-process safe.

On se contente ici de *sem_init* auquel on passe 1 en deuxième arg pour préciser que c'est des processus. Réciproquement, le destructeur va utiliser *sem_destroy* pour nettoyer ces objets.

On note au passage les différences avec l'exo précédent: on y a utilisé *sem_open*, *sem_unlink*. Nos sémaphores sont ici "anonymes" au sens où ils n'ont pas d'entrée spécifique (inode) dans le système de fichier.

Donc on a besoin de trois semaphores :

- un sem qui joue le role de mutex, il protège sz et tab. Initialisé à 1, les accès aux attibuts sont des sections critiques, chacun fait un $P$ avant et un $V$ après avoir lu/écrit *tab* ou *sz*.
- deux sem qui jouent le rôle de mécanisme de notification et de contrôle anti débordement.
  - sempop : initialisé à 0, on P dessus avant de pop, on V dessus après un push. Donc au départ 0, c'est vide, la valeur du compteur reflète size.
  - sempush : initialisé à MAX, on P dessus avant de push, on V dessus après un pop. Donc au départ MAX, c'est vide on peut fair MAX push. La valeur du compteur reflète $MAX - size$.

Stack.h

```
#ifndef __STACK_HH__
#define __STACK_HH__

#include <cstring> // size_t
#include <semaphore.h>
#include <iostream>

namespace pr {

#define STACKSIZE 100

// T = un type numerique (contrat de valarray)
template<typename T>
class Stack {
      T tab [STACKSIZE];
      size_t sz;
      sem_t mutex;
      sem_t sempop;
      sem_t sempush;
public :
      Stack () : sz(0) {
            sem_init(&mutex,1,1);
            sem_init(&sempop,1,0);
```

```
            sem_init(&sempush,1,STACKSIZE);
            memset(tab,0,sizeof tab) ;
      }

      ~Stack() {
            sem_destroy(&mutex);
            sem_destroy(&sempop);
            sem_destroy(&sempush);
      }

      T pop () {
            sem_wait(&sempop);
            sem_wait(&mutex);
            T toret = tab[--sz];
            sem_post(&mutex);
            sem_post(&sempush);
            return toret;
      }

      void push(T elt) {
            sem_wait(&sempush);
            sem_wait(&mutex);
            tab[sz++] = elt;
            sem_post(&mutex);
            sem_post(&sempop);
      }
};

}


#endif
```

Attention à bien initialiser dans le constructeur, et aussi à les libérer à la destruction. On veut s'en servir entre *processus* (et non entre thread) du coup on passe 1 en deuxième argument au *sem_init*.

**Question 2.** Les processus consommateur et producteur ne peuvent pas communiquer à travers le stack actuellement, car il n'est pas dans une zone partagée. Proposez une correction du code pour que le Stack soit dans un segment de mémoire partagée anonyme. Comparez le code à une version dans un segment nommé.

**NB : In place new.** Le C++ permet de construire un objet avec new, mais dans un endroit arbitraire, supposé déjà alloué à une taille suffisante. Pour une classe *MyClass*, on peut écrire :

```
void * zone = /*une adresse vers une zone pré-allouée */ ;
MyClass * mc = new (zone) MyClass(arg1,arg2);
```

Avec cette syntaxe, le constructeur de la classe est invoqué, mais il n'y pas d'allocation. On peut s'en servir pour initialiser des objets C++ dans un segment de mémoire partagée. Attention dans ce cas à la destruction : il faut invoquer le destructeur explicitement `mc->~MyClass();`, mais sans faire un `delete` qui ferait aussi une désallocation.

Donc deux versions, la version anonyme est plus facile, en gros on dirait un malloc, mais on ne peut s'en servir qu'au sein d'une arborescence de processus. L'astuce donc on ajoute "MAP_ANONYMOUS" aux flags de *mmap*, et on passe $-1$ pour le filedescriptor.

La version nommée, on commence par "shm_open" (qui crée une inode pour ce shm), puis "ftruncate" pour lui allouer une taille (l'inode par défaut, c'est vide, comme si on "touch" un fichier). Ensuite on passe le filedescriptor obtenu à l'invocation à mmap. Cette invocation monte les pages en mémoire de

l'inode, de façon à garantir que les écritures sont vues immédiatement par les lecteurs, l'ensemble est *memory mapped*.
Et on ajoute un "shm_unlink" à la fin du code pour nettoyer l'inode créée par le *shm_open*.

prodcons.cpp

```
#include <iostream>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <fcntl.h>
#include "Stack_cor.h"
#include <vector>


using namespace std;
using namespace pr;

void producteur (Stack<char> * stack) {
      char c ;
      while (cin >> c) {
            stack->push(c);
      }
}

void consomateur (Stack<char> * stack) {
      while (true) {
            char c = stack->pop();
            cout << c << std::flush ;
      }
}

std::vector<pid_t> tokill;

void killem (int) {
      for (auto p : tokill) {
            kill(p,SIGINT);
      }
}

int main () {
      size_t shmsize = sizeof(Stack<char>);
      std::cout << "Allocating segment size "<<shmsize << std::endl;

      int fd;
      void * addr;

      bool useAnonymous = false;
      if (useAnonymous) {
            addr = mmap(nullptr, shmsize, PROT_READ | PROT_WRITE, MAP_SHARED |
                MAP_ANONYMOUS, -1, 0);
            if (addr == MAP_FAILED) {
                  perror("mmap anonymous");
                  exit(1);
            }
      } else {
            fd = shm_open("/myshm",O_CREAT|O_EXCL|O_RDWR,0666);
            if (fd < 0) {
                  perror("shm_open");
```

```
                return 1;
        }
        if (ftruncate(fd,shmsize) != 0) {
                perror("ftruncate");
                return 1;
        }
        addr = mmap(nullptr, shmsize, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
        if (addr == MAP_FAILED) {
                perror("mmap anonymous");
                exit(1);
        }
    }

    // In place new : faire le new à l'addresse fournie, supposée assez grande.
    Stack<char> * s = new (addr) Stack<char>();

    pid_t pp = fork();
    if (pp==0) {
        producteur(s);
        return 0;
    }

    pid_t pc = fork();
    if (pc==0) {
        consomateur(s);
        return 0;
    } else {
        tokill.push_back(pc);
    }

    signal(SIGINT, killem);

    wait(0);
    wait(0);

    // invoque le destructeur, mais pas delete
    s->~Stack();
    if (munmap(addr,shmsize) != 0) {
        perror("munmap");
        exit(1);
    }
    if (! useAnonymous) {
        if (shm_unlink("/myshm") != 0) {
            perror("sem unlink");
        }
    }
}
```

On fait effectivement un "in place new" pour le Stack. Attention le delete à la fin saute par contre, en faveur d'une invocation explicite au dtor.

**Question 3.** Le programme ne se termine pas actuellement, ajoutez une gestion de signal pour que le main interrompe le(s) consommateur(s) si on lui envoie un Ctrl-C (SIGINT).

Ok, donc on revient un peu sur les signaux. Je stocke les pid des consommateurs (qui vont finir bloqués), le père quand il se prend Ctrl-C transmet le signal à ses fils, qui en mourront (comportement par défaut). J'ai fait une var globale (vecteur des pid des fils) pour éviter les soucis de conversion d'une lambda avec

> capture vers un pointeur de fonction “void () (int sig)”.
>
> Seuls les consommateurs sont bloqués; le code est prévu pour facilement généraliser à plusieurs consommateurs, i.e. on utilise un vector pour stocker un seul pid.

**Question 4.** Généraliser le programme à plusieurs producteurs et/ou consommateurs.

> Rien de bien méchant, mais pas de corrigé ici désolé.
>
> On remarque simplement que les synchros sur le stack et le comportement global de notre programme permet effectivement de supporter un nombre arbitraire de consommateurs ou producteurs.

# TME 6 : Mémoire Partagée, Sémaphores

**Objectifs pédagogiques :**

- **shared memory**
- **semaphores**

## 0.1 Github pour le TME

Les fichiers fournis sont a récupérer suivant la même procédure qu'au TME5 sous authentification github : https://classroom.github.com/a/PRLSwEwI.

Vous ferez un push de vos résultats sur ce dépôt pour soumettre votre travail.

Attention à activer les flags $-pthread$ mais aussi $-lrt$ au link pour avoir les sémaphores.

# 1 Fork, exec, pipe

**Question 1.** On souhaite simuler le comportement d'un shell pour chaîner deux commandes : la sortie de la première commande doit alimenter l'entrée de la deuxième commande.

Ecrire un programme "pipe" qui a ce comportement. On utilisera :

- fork et exec, on suggère d'utiliser la version execv,
- pipe pour créer un tube,
- dup2(fd1,fd2) pour remplacer fd2 par fd1, de façon à substituer aux entrées sorties "normales" les extrémités du pipe

On pourra tester avec par exemple (le backslash protège l'interprétation du pipe par le shell) :

```
pipe /bin/cat pipe.cpp \| /bin/wc -l
```

**NB 1 :** Il faut itérer sur les arguments $argc, argv$ passés au main pour chercher le symbole pipe '|', et construire deux tableaux d'arguments (des $const\ char*$) terminés par $nullptr$ pour invoquer $execv$.

**NB 2 :** le cours comporte un exemple très proche, en page 57 du cours 6.

Donc une boucle un peu moche/bancale, on suppose que les deux sous commandes ne peuvent pas avoir plus de $argc$ éléments pour dimensionner, du coup on fait un memset à 0 pour avoir un nullptr à la fin de la boucle dans chacun des tableaux.

Bref, à l'issue de ça on a collecté les deux sous paquets d'arguments dans deux tableaux du C, qu'on peut passer correctement à l'API de $execv$. Le execv rappelons le ne va jamais revenir, soit on se prend un erreur sur execv, soit le contexte entier du processus est absorbé, pas de retour comme si c'était un call de fonction, on écrase le segment de code la pile tout ça, on place le curseur de programme en 0 et on tourne sur cette nouvelle image.

La suite c'est dans le cours, en gros entre le fork et le exec on substitue les flux sous les pieds du processus. On note que le père et le fils après un fork partagent les descripteurs de fichiers ouverts, c'est à dire que un "int fd" que le père a "open" est utilisable par le fils pour faire read/write/close. En particulier stdin/out/err sont partagés, on le constate bien dans nos petits programmes avec des fork.

Donc en remplaçant le sens de 0 (=STDIN) ou le sens de 1 (=STDOUT) entre le fork et l'exec, on contrôle où les flux d'entrées / sorties du fils vont aller.

pipe.cpp

```
#include <sys/types.h>                                                  1
#include <sys/wait.h>                                                   2
#include <unistd.h>                                                     3
#include <stdlib.h>                                                     4
```

```
#include <stdio.h>
#include <cstring>
#include <iostream>


void myexec (int argc, char ** args) {
      std::cerr << "args :" ;
      for (int i =0; i < argc ; i++) {
            if (args[i] != nullptr) {
                  std::cerr << args[i] << " ";
            } else {
                  break;
            }
      }
      std::cerr << std::endl;

      if (execv (args[0], args) == -1) {
            perror ("exec");
            exit (3);
      }
}

int main (int argc, char ** argv) {

      // On partitionne les args en deux jeux d'arguments
      char * args1[argc];
      char * args2[argc];

      memset(args1,0,argc*sizeof(char*));
      memset(args2,0,argc*sizeof(char*));

      int arg = 1;
      for ( ; arg < argc ; arg++) {
            if (! strcmp(argv[arg],"|")) {
                  arg++;
                  break;
            } else {
                  args1[arg-1] = argv[arg];
            }
      }
      for (int i=0; arg < argc ; i++,arg++) {
            args2[i] = argv[arg];
      }

      // OK, args1 = paramètre commande 1, args2 = deuxième commande.

      int tubeDesc[2];
      pid_t pid_fils;
      if (pipe (tubeDesc) == -1) {
            perror ("pipe");
            exit (1);
      }
      if ( (pid_fils = fork ( )) == -1 ){
            perror ("fork");
            exit (2);
      }
      if (pid_fils == 0) { /* fils 1 */

            dup2(tubeDesc[1],STDOUT_FILENO);
```

```
            close (tubeDesc[1]);
            close (tubeDesc[0]);

            myexec(argc,args1);
            // on ne revient pas du exec
      }

      // pere donc ici
      if ( (pid_fils = fork ( )) == -1 ){
            perror ("fork");
            exit (2);
      }
      if (pid_fils == 0) { /* fils 2 */
            dup2(tubeDesc[0],STDIN_FILENO);
            close (tubeDesc[0]);
            close (tubeDesc[1]);

            myexec(argc,args2);
      }

      // important
      close (tubeDesc[0]);
      close (tubeDesc[1]);

      wait(0);
      wait(0);
      return 0;
}
```

## 2 Sémaphore, Mémoire partagée

Reprenez l'exercice du TD 6 sur le stack partagé entre producteurs et consommateurs.

**Question 1.** Ecrivez progressivement une version avec $N$ producteurs et $M$ consommateurs, qui utilise un segment de mémoire partagée nommé, et se termine proprement sur Ctrl-C.

**Question 2.** Assurez vous de bien avoir libéré les ressources, on utilisera $O\_CREAT|O\_EXCL$ pour faire les *open* et on vérifiera que l'on peut exécuter deux fois le programme d'affilée.

Voir corrigé du TD.

## 3 Messagerie par mémoire partagée

Résoudre l'exercice "Une messagerie instantanée en mémoire partagée" https://www-master.ufr-info-p6.jussieu.fr/2017/Une-messagerie-instantanee-en.

Le fichier "chat_common.h" est dans le dépot git.

Solution en C brute de l'an dernier.

common.h

```
#ifndef H_CHAT_COMMON
```

```c
#define H_CHAT_COMMON

#define _XOPEN_SOURCE 700
#define _REENTRANT
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <ctype.h>

#include <sys/ipc.h>
#include <sys/mman.h>
#include <sys/stat.h> /* Pour les constantes des modes */
#include <fcntl.h> /* Pour les constantes O_* */
#include <semaphore.h>
#include <signal.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <time.h>
#include <errno.h>

#include <string.h>

#define MAX_MESS 50
#define MAX_USERS 10
#define TAILLE_MESS 10

struct message {
  long type;
  char content[TAILLE_MESS];
};

struct myshm {
  int read; /* nombre de messages retransmis par le serveur */
  int write; /* nombre de messages non encore retransmis par le serveur */
  int nb; /* nombre total de messages emis */
  sem_t sem;
  struct message messages[MAX_MESS];
};

char *getName(char *name);

#endif
```

serveur.c

```c
#include <chat_common.h>

int shouldexit = 0;

void exithandler(int sig){
  shouldexit = sig; /* i.e !=0 */
}

struct user {
  char *name;
  struct myshm *shm;
} *users[MAX_USERS];
```

```c

int main(int argc, char *argv[]){
  char *shmname;
  int shm_id, i;
  struct myshm *shm_pere;
  struct sigaction action;

  if (argc <= 1) {
    fprintf(stderr, "Usage: %s id_server\n", argv[0]);
    exit(EXIT_FAILURE);
  }

  /* Met un handler pour arrêter le programme de façon clean avec Ctrl-C */
  action.sa_handler = exithandler;
  action.sa_flags = 0;
  sigemptyset(&action.sa_mask);
  sigaction(SIGINT, &action, 0);

  /* On crée l'identifiant "/<server_id>_shm:0" */
  shmname = (argv[1]);

  /* Crée le segment en lecture écriture */
  if ((shm_id = shm_open(shmname, O_RDWR | O_CREAT, 0666)) == -1) {
    perror("shm_open shm_pere");
    exit(EXIT_FAILURE);
  }

  /* Allouer au segment une taille de NB_MESS messages */
  if(ftruncate(shm_id, sizeof(struct myshm)) == -1) {
    perror("ftruncate shm_pere");
    exit(EXIT_FAILURE);
  }

  /* Mappe le segment en read-write partagé */
  if((shm_pere = mmap(NULL, sizeof(struct myshm), PROT_READ | PROT_WRITE, MAP_SHARED,
      shm_id, 0)) == MAP_FAILED){
    perror("mmap shm_pere");
    exit(EXIT_FAILURE);
  }
  shm_pere->read = 0;
  shm_pere->write = 0;
  shm_pere->nb = 0;

  /* Initialisation du sémaphore (mutex) */
  if(sem_init(&(shm_pere->sem), 1, 1) == -1){
    perror("sem_init shm_pere");
    exit(EXIT_FAILURE);
  }

  /* Initialisation du tableau de structure users à NULL */
  for(i=0; i<MAX_USERS; i++){
    users[i] = NULL;
  }

  while(!shouldexit){
    sem_wait(&(shm_pere->sem));
    /* Si il y a eu au moins une écriture */
    if(shm_pere->read != shm_pere->write || shm_pere->nb == MAX_MESS){
      /* On récupère le premier message et on incrémente le nombre lu */
```

```
    struct message message = shm_pere->messages[shm_pere->read];

    switch(message.type){
    case 0:{
      /* Connexion */
      char *username;
      int shm_id_user;
      i=0;

      /* Récupère la première case user vide */
      while(i<MAX_USERS && users[i] != NULL) i++;
      if(i == MAX_USERS){perror("impossible d'ajouter l'user"); exit(EXIT_FAILURE);}

      users[i] = malloc(sizeof(struct user));
      if(users[i] == NULL){perror("impossible d'ajouter l'user"); exit(EXIT_FAILURE);}

      /* Copie le nom du fils */
      users[i]->name = malloc((strlen(message.content) + 1) * sizeof(char));
      strcpy(users[i]->name, message.content);
      users[i]->name[strlen(message.content)] = '\0';

      printf("Connexion de %s\n", users[i]->name);

      /* Récupère le segment partagé de l'user et le mappe */
      username = message.content;
      if((shm_id_user = shm_open(username, O_RDWR | O_CREAT, 0666)) == -1) {
        perror("shm_open shm_fils");
        exit(EXIT_FAILURE);
      }
      if((users[i]->shm = mmap(NULL, sizeof(struct myshm), PROT_READ | PROT_WRITE,
          MAP_SHARED, shm_id_user, 0)) == MAP_FAILED){
        perror("mmap shm_fils");
        exit(EXIT_FAILURE);
      }

      break;
    }
    case 1:{
      /* Envoi de message */
      int temp = 0;
      printf("Réception message serveur : %s\n", message.content);
      for(i=0; i<MAX_USERS; i++){
        if(users[i] != NULL){
          struct message msg;
          struct myshm *shm = users[i]->shm;
          temp++;

          /* On copie le message dans la file du fils */
          msg.type = 1;
          strcpy(msg.content, message.content);

          sem_wait(&(shm->sem));

          while(shm->nb == MAX_MESS){
            sem_post(&(shm->sem));
            sleep(1);
            sem_wait(&(shm->sem));
          }
```

```
        /* TODO : vérifier que c'est pas plein */

        printf("\t\tEnvoi à %s\n", users[i]->name);


        shm->messages[shm->write] = msg;
        shm->write = (shm->write + 1) % MAX_MESS;
        shm->nb++;

        sem_post(&(shm->sem));
       }
      }

      if(temp == 0){
       printf("Nobody found!\n");
      }
      break;
     }
     case 2:{
      printf("Déconnexion de %s\n", message.content);
      /* Déconnexion */
      i=0;

      /* On récupère le user correspondant à l'user qui se déconnecte */
      while(i<MAX_USERS && (users[i] == NULL || strcmp(users[i]->name, message.content)
          != 0)) i++;
      if(i == MAX_USERS){
       perror("Something went wrong! L'user n'existe pas!");
       exit(EXIT_FAILURE);
      }

      /* On libère l'espace mémoire */
      free(users[i]->name);
      munmap(users[i]->shm, sizeof(struct myshm));
      free(users[i]);
      users[i] = NULL;

      break;
     }
     }
     shm_pere->read = (shm_pere->read + 1) % MAX_MESS;
     shm_pere->nb--;
   }
   sem_post(&(shm_pere->sem));
  }

  sem_close(&(shm_pere->sem));

  munmap(shm_pere, sizeof(struct myshm));

  shm_unlink(shmname);

  return EXIT_SUCCESS;
}
```

client.c

```
#include <chat_common.h>
```

```c

int shouldexit = 0;

struct myshm *shm_client, *shm_pere;

void *reader(void* arg){
  while(!shouldexit){
    sem_wait(&(shm_client->sem));

    /* Si il y a eu au moins une écriture */
    if(shm_client->read != shm_client->write || shm_client->nb == MAX_MESS){
      /* On récupère le premier message et on incrémente le nombre lu */
      struct message message = shm_client->messages[shm_client->read];
      shm_client->read = (shm_client->read + 1) % MAX_MESS;
      printf("%s", message.content);
      shm_client->nb--;
    }
    sem_post(&(shm_client->sem));
  }
  pthread_exit(NULL);
  return NULL;
}

void *writer(void* arg){
  while(!shouldexit){
    struct message msg;

    msg.type = 1;
    fgets(msg.content, TAILLE_MESS, stdin);

    /* Evite d'envoyer un message avec Ctrl-C dedans */
    if(shouldexit) break;

    sem_wait(&(shm_pere->sem));

    while(shm_pere->nb == MAX_MESS){
      sem_post(&(shm_pere->sem));
      sleep(1);
      sem_wait(&(shm_pere->sem));
    }

    shm_pere->messages[shm_pere->write] = msg;
    shm_pere->write = (shm_pere->write + 1) % MAX_MESS;
    shm_pere->nb++;

    sem_post(&(shm_pere->sem));
  }
  pthread_exit(NULL);
  return NULL;
}

void exithandler(int sig){
  shouldexit = sig; /* i.e. !=0 */
}

int main(int argc, char *argv[]){
  char *shm_client_name, *shm_pere_name;
  int shm_client_id, shm_pere_id;
  pthread_t tids[2];
```

```c
struct message msg;
struct sigaction action;

if (argc <= 2) {
  fprintf(stderr, "Usage: %s id_client id_server\n", argv[0]);
  exit(EXIT_FAILURE);
}

printf("Connexion à %s sous l'identité %s\n", argv[2], argv[1]);

/* Met un handler pour arrêter le programme de façon clean avec Ctrl-C */
action.sa_handler = exithandler;
action.sa_flags = 0;
sigemptyset(&action.sa_mask);
sigaction(SIGINT, &action, 0);

shm_client_name = (argv[1]);
shm_pere_name = (argv[2]);

/* Crée le segment en lecture écriture */
if ((shm_client_id = shm_open(shm_client_name, O_RDWR | O_CREAT, 0666)) == -1) {
  perror("shm_open shm_client");
  exit(errno);
}
if ((shm_pere_id = shm_open(shm_pere_name, O_RDWR | O_CREAT, 0666)) == -1) {
  perror("shm_open shm_pere");
  exit(errno);
}

/* Allouer au segment une taille de NB_MESS messages */
if(ftruncate(shm_client_id, sizeof(struct myshm)) == -1) {
  perror("ftruncate shm_client");
  exit(errno);
}

/* Mappe le segment en read-write partagé */
if((shm_pere = mmap(NULL, sizeof(struct myshm), PROT_READ | PROT_WRITE, MAP_SHARED,
    shm_pere_id, 0)) == MAP_FAILED){
  perror("mmap shm_pere");
  exit(errno);
}
if((shm_client = mmap(NULL, sizeof(struct myshm), PROT_READ | PROT_WRITE, MAP_SHARED,
    shm_client_id, 0)) == MAP_FAILED){
  perror("mmap shm_pere");
  exit(errno);
}
shm_client->read = 0;
shm_client->write = 0;
shm_client->nb = 0;

/* Initialisation du sémaphore (mutex) */
if(sem_init(&(shm_client->sem), 1, 1) == -1){
  perror("sem_init shm_client");
  exit(errno);
}

msg.type = 0;
strcpy(msg.content, argv[1]);
sem_wait(&(shm_pere->sem));
```

```c
 shm_pere->messages[shm_pere->write] = msg;
 shm_pere->write = (shm_pere->write + 1) % MAX_MESS;
 shm_pere->nb++;

 sem_post(&(shm_pere->sem));

 /* On crée les threads et on attend qu'ils se terminent */
 pthread_create(&(tids[0]), NULL, reader, NULL);
 pthread_create(&(tids[1]), NULL, writer, NULL);
 pthread_join(tids[0], NULL);
 pthread_join(tids[1], NULL);

 printf("Déconnexion...\n");

 msg.type = 2;
 sem_wait(&(shm_pere->sem));

 shm_pere->messages[shm_pere->write] = msg;
 shm_pere->write = (shm_pere->write + 1) % MAX_MESS;
 shm_pere->nb++;

 sem_post(&(shm_pere->sem));

 sem_close(&(shm_client->sem));

 munmap(shm_client, sizeof(struct myshm));

 shm_unlink(shm_client_name);

 return EXIT_SUCCESS;
}
```