# 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.

## 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.

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

## 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".

**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.

## 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.

**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.

**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).

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

# 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.

# 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.

# 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.