# TD 9 : Future, Promise, Task

**Objectifs pédagogiques :**

- **future, promise, async en C++**
- **parallélisme de tâches**

### Introduction

Dans ce TD, nous allons étudier des briques participant à la mise en place d'un modèle de concurrence basé sur les tâches, partiellement décorrélé du modèle de parallélisme.

## 1 Future, Promise

L'objectif de cet exercice est de réaliser nos propres versions de `future` et `promise` simplifiées par rapport au standard (pas de gestion des exceptions) en appui sur les primitives de synchronisation classiques des thread (mutex, condition).

On propose d'implanter trois classes template : `shared_result`, `promise`, `future`.

### 1.1 Shared Result

La classe `shared_result` est le coeur de notre synchronisation ; c'est un conteneur qui *peut* héberger un objet de type $T$. Son API est la suivante :

- `T get ()` rend la valeur stockée, bloquant si elle n'est pas disponible.
- `void set(T &val)` positionne la valeur, débloque le client bloqué sur `get` le cas échéant.
- `bool is_set () const` (non bloquant) rend vrai si `set` a été invoqué sur cet objet.

**Question 1.** Proposez, en appui sur les primitives de synchronisation des threads C++11, une implantation pour cette classe.

Pool.h

```
#ifndef SRC_PROMISE_H_
#define SRC_PROMISE_H_

#include <mutex>
#include <condition_variable>
#include <memory>
#include <future>

namespace pr {

template<typename T>
class shared_result {
      // la donnée
      T res;
      // est-on valide ?
      bool isSet;
      // éléments de synchro
      std::mutex mut;
      std::condition_variable cv;
public:
      // init = not set
      shared_result(): res(),isSet(false) {}
```

```
      // bloquant si not set
      T get () {
            std::unique_lock<std::mutex> l (mut);
            // NB : cv.wait(l,pred) <=> while (!pred) cv.wait(l);
            // On pourrait utiliser direct l'attribut plutot que l'accesseur
            cv.wait(l,[this] () { return is_set();});

            // initialement (Q1)
            // en Q1 : version simple (paie la copie)
            // return res;

            // Q5 : on ajoute un move, qui invalide du coup res.
            // du coup mettre is_set à false parait raisonnable
            // mais en vrai on NE DOIT PAS refaire get sur ce future.
            isSet = false;
            return std::move(res);
      }
      void set (T & val) {
            {
                  std::unique_lock<std::mutex> l (mut);
                  // en Q1 : version simple (paie la copie)
                  // res = val;

                  // en Q5 : utilisation du move
                  // du coup on ne paie plus la copie, mais le résultat est invalidé par un get.
                  res = std::move(val);
                  isSet = true;
            }
            cv.notify_one();
      }
      bool is_set () const {
            return isSet;
      }
};

// Question 4 : gestion mémoire
// on utilise un shared_ptr, muni de son compteur de références
// Quand promise et future sont détruits, alors le shared_result peut être libéré
template<typename T>
using Resptr = std::shared_ptr<pr::shared_result<T>>;

// Au début (en question 2 et 3) on utilise juste un "pr::shared_result<T>*"
// et on évite de parler trop des destructeurs...

// Notre "future" ne fait quasiment que déléguer au shared_result.
// il manque un "make_shared" entre autres pour que ce soit plus intéressant.
template<typename T>
class future {

      // initialement juste un pointeur ici
      // en Q5 on utilise un shared_ptr plutot
      Resptr<T> result;
public :
      future (const Resptr<T> & res): result(res) {}
      // bloquant si non disponible
      T get () {
            return result->get();
```

```
      }
      bool isAvailable() const {
            return result->is_set();
      }
};

// La promise, s'occupe de construire le shared_state, le future
template<typename T>
class promise {
      // au début on utilise un simple pointeur
      Resptr<T> result;
public :
      // ctor : construire un shared result (avec new)
      // au début ce sera juste un pointeur ici.
      // Mais ne pas le delete dans le dtor, car future peut aussi le pointer
      // avec shared_ptr on s'affranchit de ces problèmes de delete
      promise(): result(new shared_result<T>()) {}

      // construction d'un future
      future<T> getFuture() {
            return future<T> (result);
      }
      // mise à jour de la valeur + débloquer
      void set_value(const T & r) {
            result->set(r);
      }
};

}

#endif /* SRC_PROMISE_H_ */
```

## 1.2 Future

La classe `future` représente le canal de lecture pour un client qui attend un résultat. On construit un objet `future` en lui passant l'adresse d'un `shared_result`. Son API est la suivante :

- `T get ()` : bloquant si pas encore disponible
- `bool isAvailable() const` : vrai si le résultat est prêt

**Question 2.** Proposez une implantation pour cette classe.

## 1.3 Promise

La classe `promise` représente le canal en écriture vers la donnée partagée, dans lequel le thread serviteur va poser le résultat de son calcul. Quand on construit un objet `promise`, il crée une instance de `shared_result`. Son API est la suivante :

- `future<T> getFuture()` construit et rend un objet future, attaché au résultat.
- `void set_value(const T & r)` affecte une valeur au résultat

**Question 3.** Proposez une implantation pour cette classe.

### 1.4 Gestion mémoire

**Question 4.** A quel moment faut-il désallouer le `shared_result` qui est pointé par le couple promise/future ? En utilisant les pointeurs intelligents de `<memory>` proposez une réalisation de ce comportement.

> Donc memory propose principalement
>
> - `unique_ptr<T>` qui détruit (delete) l'objet pointé quand le pointeur est détruit. La copie n'est pas supportée, on fait un move dans ce cas.
> - `shared_ptr<T>` qui ajoute un compteur de ref (atomic pour l'accès MT) à l'objet pointé, et détruit l'objet quand le compteur atteint zéro. La copie est supportée et incrémente le compteur.
>
> Ces classes redéfinissent les opérateurs * et -> pour simuler la syntaxe d'un pointeur normal.
>
> Ici on veut un "shared", pour plus de commodité on utilise une directive "using" pour définir ce type dans le corrigé. Avec ce typedef, la syntaxe d'utilisation dans le code ne bouge pas (i.e. on peut définir "T*" ou "shared_ptr<T>" interchangeablement).

**Question 5.** Si on suppose que le résultat est une classe qui alloue de la mémoire (e.g. une String), combien de copies sont payées dans l'implantation actuelle pour transmettre le résultat du thread serviteur au thread client ? Comment pourrait-on contourner ce problème ?

> Donc on paie une copie à chaque fois qu'on affecte à un objet de type T un autre objet de type T.
>
> On a une copie donc actuellement dans le `set_value`, et une copie dans le `get` de la classe shared result. Deux copies, c'est pas terrible, déjà qu'on paie des synchros...
>
> La solution, c'est d'utiliser la sémantique de std::move, std::move rend un objet initialisé à partir du premier objet, mais qui lui pique ses données. L'argument passé à move est donc invalidé par cet appel, mais on évite la copie. Par exemple le move d'une std::string copie le pointeur vers data, mais pas data, la string d'origine se retrouve à pointer dans le vide (ou plus précisément c'est une string vide après l'opération).
>
> On utilise donc un `std::move` au lieu `operator=` aux endroits où on fait une copie actuellement.
>
> L'effet, c'est que notre objet "shared_result" se retrouve vidé, dans un état invalide après l'appel à get. C'est la sémantique proposée par le standard.

**Question 6.** Si on souhaite supporter le scenario où plusieurs clients attendent le résultat calculé par le serviteur, le standard propose `shared_future`. Expliquez les différences avec le `future` simple.

> Donc effectivement, avec cette sémantique, plus possible d'avoir plusieurs future dans le système qui pointent le même shared_result.
>
> La classe `shared_future` offre un `const T & get()` au lieu de la signature `T get()` de la classe `future`. On construit un shared future, à partir d'un future avec opération membre "share", ou constructeur de `shared_future` prenant un `future` en argument.
>
> La version shared rend une const ref, qui reste valable tant que le shared result n'est pas détruit. Donc toujours pas de copie, sauf si un client affecte cet ref d'objet à un objet de son contexte (copie à la demande).

## 2 Pool de thread et Future

On revient sur la réalisation d'un pool de thread, comme au TD4.

Pool.h

```
#ifndef SRC_POOL_H_
```

```cpp
#define SRC_POOL_H_

#include "Queue.h"
#include <vector>
#include <thread>

namespace prTD4 {

class Job {
public:
	virtual void run () = 0;
	virtual ~Job() {};
};

// fonction passee a ctor de thread
inline void poolWorker(Queue<Job> * queue) {
	while (true) {
		Job * j = queue->pop();

		// pour la terminaison propre
		if (j == nullptr) {
			// on est non bloquant = il faut sortir
			return;
		}

		j->run();
		delete j;
	}
}

class Pool {
	Queue<Job> queue;
	std::vector<std::thread> threads;
public:
	Pool(int qsize) : queue(qsize) {}
	void start (int nbthread) {
		threads.reserve(nbthread);
		for (int i=0 ; i < nbthread ; i++) {
			threads.emplace_back(poolWorker, &queue);
		}
	}

	void stop() {
		// débloque les threads bloqués sur pop
		queue.setBlocking(false);
		// on attend qu'ils aient fini leur job actuel
		for (auto & t : threads) {
			t.join();
		}
		threads.clear();
	}
	~Pool() {
		stop();
	}
	void submit (Job * job) {
		queue.push(job);
	}
};

} /* namespace pr */
```

```
#endif /* SRC_POOL_H_ */
```

On veut réaliser une version `Pool<T>` auquel on puisse soumettre des `Job<T>` qui définit une seule opération `T run()` (donc au lieu de la signature `void run()` proposée). Pour éviter de tomber trop profondément dans les problèmes d'instantiation de template, tous les `Job<T>` soumis à un `Pool<T>` donné auront donc le même type de retour $T$.

La nouvelle signature de l'opération `void submitJob (Job *job)` est `future<T> submitJob(Job<T> *job)`.

**Question 1.** Modifier le Pool de thread et les définitions liées (Job, fonction exécutée par les thread...) pour obtenir ce nouveau comportement.

Pool.h

```
#ifndef SRC_POOL_H_
#define SRC_POOL_H_

#include "promise.h"
#include "Queue.h"
#include <vector>
#include <thread>

namespace pr {

template<typename T>
class Job {
public:
      virtual T run () = 0;
      virtual ~Job() {};
};

// AJOUT : un mécanisme qui stocke la paire Job/promise associée
// dès le submit on doit créer une promise, pour rendre toute suite le future associé
// quand un thread qui pop ce PackagedTask a fini de traiter le job, on set value dans cette promise.
template<typename T>
class PackagedTask {
      Job<T> * job;
      promise<T> result;
public :
      // implicitement : promise<T> est construit, ce qui crée un shared_result<T> (avec un new)
      // et y réfère via un shared_ptr
      PackagedTask (Job<T> * job) : job(job){}
      // API d'une paire
      promise<T> & getPromise() { return result; }
      Job<T> * getJob() { return job; }
      // NB : avec le shared_ptr utilisé pour référer au shared_result, pas de souci ici
      // la promise va mourir aussi, mais pas le résultat shared_result
      ~PackagedTask() { delete job; }
};


// fonction passee a ctor de thread
template<typename T>
inline void poolWorker(Queue<PackagedTask<T>> * queue) {
      while (true) {
            // le pop nous rend maintenant un PackagedTask, une paire Job/Promise
```

```
            PackagedTask<T> * task = queue->pop();
            // pour la terminaison propre
            if (task == nullptr) {
                  // on est non bloquant = il faut sortir
                  return;
            }
            Job<T> * j = task->getJob();
            // On invoque run = traitement
            // dans la foulée, on met à jour la promise avec ce résultat
            // ca debloque potentiellement le client en attente sur le future
            task->getPromise().set_value(j->run());
            // destruction des données du Job, de la promise qui a été satisfaite
            delete task;
      }
}

template<typename T>
class Pool {
      // on stocke des paires Job/Promise
      Queue<PackagedTask<T>> queue;
      std::vector<std::thread> threads;
public:
      Pool(int qsize) : queue(qsize) {}
      void start (int nbthread) {
            threads.reserve(nbthread);
            for (int i=0 ; i < nbthread ; i++) {
                  threads.emplace_back(poolWorker<int>, &queue);
            }
      }

      void stop() {
            // débloque les threads bloqués sur pop
            queue.setBlocking(false);
            // on attend qu'ils aient fini leur job actuel
            for (auto & t : threads) {
                  t.join();
            }
            threads.clear();
      }
      ~Pool() {
            stop();
      }
      future<T> submit (Job<T> * job) {
            // on ecapsule dans un PackagedTask
            // ce qui engendre la création de la promise, donc du shared_result
            PackagedTask<T> * t = new PackagedTask<T>(job);
            queue.push(t);
            // on rend le future
            return t->getPromise().getFuture();
      }
};

} /* namespace pr */

#endif /* SRC_POOL_H_ */
```

**Question 2.** Expliquer comment utiliser ce nouveau mécanisme dans un main.

main.cpp

```cpp
#include "Pool.h"
#include <iostream>
#include <thread>
#include <vector>

using namespace std;
using namespace pr;


class SleepJob : public Job<int> {
	int foo (int v) {
		// traiter un gros calcul
		this_thread::sleep_for(1s);
		return v % 255;
	}
	int arg;
public :
	SleepJob(int arg) : arg(arg) {}
	int run () {
		std::cout << "Computing for arg =" << arg << std::endl;
		int ret = foo(arg);
		std::cout << "Obtained for arg =" << arg << " result " << ret << std::endl;
		return ret;
	}
	~SleepJob(){}
};

int main () {
	const int NBTHREAD = 5;
	const int NBJOB = 30;

	Pool<int> pool(NBTHREAD);
	pool.start(NBTHREAD);

	vector<pr::future<int> > results;

	for (int i = 0; i < NBJOB ; i++) {
		results.emplace_back(pool.submit(new SleepJob(i)));
	}

	for (auto & i : results) {
		std::cout << i.get() << std::endl;
	}
	// garantie : tout est fini ici
	pool.stop();

	return 0;
}
```

## 3 Taches et Threads, Concurrence et Parallélisme

**Question 1.** Discutez des différences au niveau syntaxe (invocation, utilisation) et sémantique (créations de thread, parallélisme effectif) entre le système d'exécution asynchrone que l'on vient de construire et les comportements (en fonction de la policy `launch_deferred|launch_async`) de la

fonction `async` proposée dans le standard.

Donc async, au niveau syntaxe est plutôt plus confortable que nous.

- On passe à async une belle fonction (ou lambda) et ses arguments correctement typés, juste comme quand on invoque le ctor de thread. Chez nous, il faut définir un Job concret pour héberger la fonction. La réalisation du standard s'appuie sur des instantiations de template assez costaud, qui ne nous intéressent pas particulièrement dans l'UE (les solutions proposées dans l'UE du coup sont portables vers d'autres langages facilement).
- On n'instancie rien pour invoquer async, au compilo/runtime de gérer. Chez nous il faut explicitement dimensionner le Pool (taille Queue, nombre de threads), en revanche on a plus de contrôle.
- async deferred n'a pas d'équivalent chez nous, on pourrait cependant assez facilement intégrer cette idée en passant un paramètre supplémentaire à "submit" job qui définisse une policy deferred. On rendrait un future qui triche, qui maintient une ref à la packagedTask, et fait le job de promise (executer la fonction) au moment du get. Ca c'est le principe, en pratique on va se débrouiller pour jeter toute l'infra structure (packagedTask), pour éliminer toutes les traces de synchros (mutex, conditions etc...) quand on passe par ce scenario.
- async avec la politique "async" force en fait la création d'un nouveau thread selon le standard. C'est juste un sucre syntaxique sur le ctor de thread, qui encapsule tout ça dans des future/promise pour nous. Ce n'est pas le cas chez nous, on ne fait pas de nouveau thread sur submit.
- async avec la politique par défaut (OR booléen entre les deux flags) ou la version qui ne précise pas de policy pourrait faire un peu ce qu'elle veut, en pratique seul le compilo VC++ (microsoft) semble utiliser effectivement un pool de thread en sous jacent (selon SO). Ca évolue cela dit selon les compilos et les versions.

**Question 2.** Comment dimensionner un pool de thread utilisé dans le contexte d'un serveur TCP ? Dans le contexte d'un ray tracer concurrent ? Que peut-on en déduire vis à vis de la seule fonction `async` proposée par le standard ?

Le problème fondamental qui se pose : combien de threads (support du parallélisme) faut-il créer par rapport à la concurrence exprimée ?

On voudrait un système (plutot proche du notre, i.e. (Job = unité de concurrence) != thread) qui supporte allégrement la création de dizaines de milliers de taches, sans écrouler le système en créant des milliers de thread.

La grosse difficulté : prenons le Pool de thread utilisé dans le contexte du TCPServer. Quel est le bon dimensionnement (nombre de threads) quand on le crée ?

Ce n'est pas lié au nombre de CPU sur la machine, mais au nombre de clients en parallèle que je veux bien accepter. Si je crée un thread par CPU, je serai vite à court de thread, vu qu'ils vont tous s'endormir régulièrement sur des I/O hyper lentes (réseau).

Au contraire pour le raytracer, qui bosse en mémoire pure sans IO ou blocages, il ne faut pas plus que disons 1 à deux thread par CPU physique (2 à 3 si on a de l'hyperthreading). Si on fait trop de thread (proportionnel au nombre de pixels) on va écrouler les performances. On devrait aussi augmenter le grain du parallélisme (par exemple un Job par ligne au lieu d'un par pixel), ou se trouver une implém de Queue qui soit lock-free (sans doute un Stack du coup) sinon on paie trop de synchros dans notre implém pour passer par des taches (i.e. être très content de spécifier une tâche par pixel) sans un surcout important.

**Question 3.** Essayez de proposer un mécanisme qui permettrait avec un nombre fixe de threads de traiter un nombre plus grand de connexions simultanées dans un serveur TCP.

Un vrai système à base tâches doit réussir dans ce genre de cas à NE PAS PIEGER UN THREAD ! Par exemple on pourrait :

- enregistrer dans un fd_set la socket sur laquelle on va se bloquer, le passer à un thread qui fait un select en boucle. Ensuite au lieu de se bloquer, on trouve un moyen de stocker le contexte e.g. stack frame, variables locales..., on l'enregistre associé au fd sur lequel on va se bloquer. Le thread part ensuite pour aller consommer une autre tâche.
- Si on se fait réveiller sur select, on retrouve la tâche associée (AVEC UN STACKFRAME) et on rempile ça dans les tâches à faire (souvent plutôt en tête si c'est une FIFO queue).
- Les threads qui ramassent des taches doivent donc réussir à se replacer dans le contexte du thread qui a exécuté (partiellement) la fonction. On parle de "continuation passing style", en gros on resoumet un Job qui contient la suite (continuation) du traitement.
- Attention aux reprises de contexte par un autre thread cela dit, en particulier les fonctions thread sensitive. Il n'y en a pas tant que ça, mais mutex/lock/unlock qui nécessite d'être fait par le même thread, ça pique un peu si on a quitté le contexte avec un mutex locké à la main.
- En C++ pas très facile non plus d'automatiquement stocker un contexte de stack/état et de le recharger. Pour notre serveur TCP, on saurait le faire, vu que l'objet ConnectionHandler est déjà porteur de tout l'état intéressant de la session et qu'il ne fait rien de compliqué niveau structure du stack, comme des récursions ou autres. Il y a du code cela dit à produire pour atteindre ça. Si on doit être général, ce n'est pas un problème simple ce qui explique sans doute l'absence de réponse dans le standard actuel.

Morale : ces problèmes n'étant pas traité par le standard, async reste plutôt gadget à ce stade. On attend avec impatience ses évolutions futures, tel quel, ce n'est pas quelque chose qu'on utilise vraiment en production. Le contrôle est trop rudimentaire sur le comportement final ; autant gérer explicitement ses propres threads (en créer pour chaque appel coûte vraiment cher), la promesse de task-based concurrency (dont on pourrait croire que c'est l'intention) n'est pas atteinte par "async" à ce stade.

Même sans avoir des connexions TCP ou autre, si on a le modèle de Pool avec un future, et qu'on considère une application comme au partiel, i.e. où les Job peuvent soumettre de nouveaux Job, mais cette fois attendent sur le future la réponse, on est mal en termes de décorrélation du parallélisme et de la concurrence. Le risque de tomber à court de thread et donc de se deadlock est élevé.

Un modèle de taches sérieux doit supporter ces scenarios (tâche avec un comportement bloquant) et les traiter (reprise du contexte).

# 4 TME 9

Dans cette séance :

- Finaliser le TME8, on souhaite atteindre la partie où l'on utilise Protobuf.
- On peut introduire les mécanismes vu dans le TD9 autour de notre pool de thread. Dans ce cas, utilisez les version `std::promise, std:::future` au lieu de celles implantées "à la main" dans le TD.
- Expérimentez avec async, par exemple, modifier le TME4 pour comparer l'utilisation du Pool de thread à celle de "async". Faites varier le grain du parallélisme, essayez d'améliorer les performances en déployant vos connaissances en parallélisme.

# TD 10 : Exercices de type Examen

**Objectifs pédagogiques :**

- **révisions**

**Introduction :** Dans ce TD, nous allons réviser les briques essentielles de la concurrence et du parallélisme vus dans l'UE. Les exercices[1] sont censés être accessibles sans assistant de TD.

## 1 Simulation des Pipes en Thread

Nous voulons offrir un ensemble de fonctions qui permet à des threads de communiquer par un tube. Autrement dit, un mécanisme de communication unidirectionnel où chacune des extrémités permet à une thread soit de lire dans un tube soit d'y écrire des caractères. A cette fin, nous avons défini la structure suivante qui regroupe des informations d'un tube donné :

```
class tube {
  char vect [TAILLE_PIPE]; /* vecteur pour sauvegarder le contenu du tube */
  int lect_off, ecr_off; /* indice de la prochaine case de vect à lire ou libre
      respectivement */
  int nb_char; /* nombre de caractères dans le tube */
public:
  tube():lect_off(0),ecr_off(0),nb_char(0){}
};
```

où :

- `vect [TAILLE_PIPE]` : vecteur de taille `TAILLE_PIPE` qui sauvegarde les caractères qui n'ont pas été encore lus dans le tube. Ce vecteur est géré de façon circulaire ;
- `lect_off et ecr_off` : indice de la prochaine case de vect à lire ou à écrire respectivement ;
- `nb_char` : nombre de caractères dans `vect` ;

On propose de réaliser les méthodes suivantes du tube, conçues pour un contexte multi-thread.

- Lecture de $n$ caractères dans le tube : `int read (char *buf, int n);`.
  - Fonction bloquante qui lit au plus $n$ caractères du tube.
  - Si le tube contient $x$ caractères, la fonction extrait du tube $nb_lu = min(x, n)$ caractères qui sont alors copiés dans buf;
  - Si le tube est vide, la thread est mise en sommeil jusqu'à ce que le tube ne soit plus vide;
  - La fonction renvoie le nombre de caractères lus dans le tube ($nb_lu$ caractères).
- Ecriture de n caractère dans le tube : `int write (char *buf, int n);`.
  - Fonction qui écrit de façon atomique $n$ caractères dans le tube, s'il y a de la place.
  - S'il y a au moins n emplacements libres dans le tube, une écriture atomique est réalisée et le nombre de caractères écrits est renvoyé. Dans ce cas tous les threads lecteur en attente sur le tube seront réveillées.
  - Sinon la fonction renvoie -1.

Nous considérons que les programmes utilisent correctement les fonctions, c'est-à-dire qu'une thread ne peut que lire ou écrire dans un tube.

**Question 1.** Ajouter les attributs utiles à la classe `tube` et codez ces deux méthodes. On suggère de procéder caractère par caractère pour plus de facilité, ou de faire deux cas selon qu'on déborde ou non (stockage circulaire) et une à deux copies (avec `memcpy`) selon le cas.

[1] Les deux premiers exercices sont adaptés d'une annale de 2006, rédigée par L. Arantes, O. Marin et P.Sens.

Rien de bien mystérieux, juste attention à bien itérer. Une version avec memcpy ici.

tube.h

```
#ifndef SRC_TUBE_H_
#define SRC_TUBE_H_

#include <mutex>
#include <cstring>
#include <condition_variable>
#include <csignal>

namespace pr {

#define TAILLE_PIPE 4096

class tube {
       char vect [TAILLE_PIPE]; /* vecteur pour sauvegarder le contenu du tube */
       int lect_off, ecr_off; /* indice de la prochaine case de vect à lire ou libre
           respectivement */
       int nb_char; /* nombre de caractères dans le tube */
       std::mutex m;
       std::condition_variable cv;

       int nbthread ; // AJOUT
     public:
       tube():lect_off(0),ecr_off(0),nb_char(0){}
       int read (char *buf, int n) {captures all by value
             std::unique_lock<std::mutex> l(m);
             cv.wait(l,[&](){ return nb_char > 0 ;});
             n = n<nb_char?n:nb_char;
             int tot = n;
             if (lect_off + n > TAILLE_PIPE) {
                 auto alire = TAILLE_PIPE - lect_off;
                 ::memcpy(buf,vect + lect_off, alire);
              lect_off = 0;
                 buf += alire;
                 nb_char -=alire;
                 n -= alire;
             }
             ::memcpy(buf,vect + lect_off, n);
             lect_off += n;
             nb_char -=n;
             return tot;
       }
       int write(char *buf, int n) {
             std::unique_lock<std::mutex> l(m);

             // AJOUT en question 3 : signaux
             if (nbthread == 1) {
                   raise(SIGPIPE);
             }
             // FIN AJOUT

             if (n > TAILLE_PIPE - nb_char) {
                   return -1;
             }
             int tot = n;
             if (ecr_off + n > TAILLE_PIPE) {
                   auto aecr = TAILLE_PIPE - ecr_off;
```

```
                    ::memcpy(vect + ecr_off, buf, aecr);
                    ecr_off = 0;
                    buf += aecr;
                    nb_char +=aecr;
                    n -= aecr;
                }
                ::memcpy(vect + ecr_off, buf,n);
                ecr_off += n;
                nb_char +=n;
                cv.notify_all();
                return tot;
        }
};

}


#endif /* SRC_TUBE_H_ */
```

**Question 2.** On suppose à présent que l'on ajoute un attribut `nbThread` comptabilisant le nombre de threads au total qui ont une référence vers une le tube. On suppose que cet attribut est correctement mis à jour au fil de la manipulation du pipe ( par exemple en appui sur `shared_ptr`). Modifier la fonction `write` pour qu'elle délivre un signal `SIGPIPE` au thread qui invoque write s'il est le seul à pouvoir réaliser une lecture.

cf corrigé précédent.

**Question 3.** Quel sera l'effet du signal sur la thread dans write ? Sur les autres threads du programmes ?

Pas terrible, on cible le processus pas une thread, on ne sait pas quelle thread va se le prendre, mais de toutes façons le handler est PROCESSUS spécifique, dont un des threads va traiter SIGPIPE (comportement par défaut = mourir).

## 2 IPC

Un programmeur très (mais alors vraiment très) inexpérimenté a voulu rédiger une petite application dans laquelle un processus émetteur envoie un message mot par mot à deux processus récepteurs de sa famille. Le résultat recherché de l'application est de faire afficher (NB : une seule fois) le message original dans son ordre initial. Par exemple, si l'émetteur envoie successivement les mots "je", suis", "en", "examen", l'affichage final de l'application sera : "je suis en examen". Le code produit par notre programmeur égaré est le suivant :

lost.cpp

```
#include <unistd.h>
#include <cstdlib>
#include <cstdio>
#include <cstring>
#include <sys/types.h>
#include <sys/wait.h>

```

```
int i, n = 6;
int pid;

void send(char *buf) {
      const char *msg[] = {"Je", "suis", "un", "vilain", "programme", "\n"};
      for(i = 0; i < n; i++) {
            strcpy(buf, msg[i]);
      }
      exit(0);
}

void receive(char *buf) {
      if ((pid = fork()) == -1) {perror("fork"); exit(1);}
      for(i = 0; i < (n/2); i++) {
            printf("%s ", buf);
      }
}

int main()
{
      char *shared = new char[50];

      if ((pid = fork()) == -1) {perror("fork"); exit(1);}

      if (pid == 0)
            send(shared);
      else
            receive(shared);

      if (pid == 0) exit(0);
      for (i = 0; i < 2; i++)
            wait(0);
      printf("fin du programme\n");
      delete [] shared;
      return 0;
}
```

**Question 1.** Expliquez la mise en place du parallélisme, en particulier identifier l'arborescence des processus engendrés et le code exécuté par chacun d'eux.

> bon la structure de controle avec les fork est encore étrange.
> Donc premier fils fait "send" ce qui termine sur exit, fin.
> Ensuite père fait receive, ce qui engendre un second fils qui fait aussi le receive.
> En sortant de receive le deuxième fils se heurte au test pid==0 et meurt.
> Le père attend les fils.
> Fin.

**Question 2.** Expliquez en deux points pourquoi ce programme ne peut pas fonctionner.

> 1. Sans mémoire partagée, les modifications de variables effectuées par un processus ne sont pas visibles par d'autres processus.
> 2. Même si (1) ne posait pas problème, sans synchronisation on ne peut pas guarantir que le contenu de la variable sera affiché après chaque modification.

**Question 3.** Sans altérer les créations de processus, corrigez le code à l'aide d'IPC afin que l'application fonctionne.

Ajout d'un semaphore/mutex, ajout d'une allocation de segment partagé (a priori anonyme).
Au niveau synchros, on est dans du ping pong entre l'ecrivain et les lecteurs.
Donc deux sémaphores, on débloque celui de l'autre quand c'est son tour. Le sémaphore "write" démarre à 1, celui du read démarre à 0.
L'ecrivain fait un $P(write)ecrire V(read)$ et le lecteur fait $P(read)lire V(write)$.

found.cpp

```cpp
#include <unistd.h>
#include <cstdlib>
#include <cstdio>
#include <cstring>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/mman.h>
#include <semaphore.h>
#include <fcntl.h>

int i, n = 6;
int pid;

sem_t * semread;
sem_t * semwrite;

void send(char *buf) {
      const char *msg[] = {"Je", "suis", "un", "vilain", "programme", "\n"};
      for(i = 0; i < n; i++) {
            sem_wait(semwrite);
            strcpy(buf, msg[i]);
            sem_post(semread);
      }
      exit(0);
}

void receive(char *buf) {
      if ((pid = fork()) == -1) {perror("fork"); exit(1);}
      for(i = 0; i < (n/2); i++) {
            sem_wait(semread);
            printf("%s\n", buf);
            sem_post(semwrite);
      }
}

int main()
{
      // mmap : address, size, protection, flags, filedescriptor, offset
      char *shared = (char *)
                  mmap(0,50,PROT_READ | PROT_WRITE,MAP_SHARED | MAP_ANONYMOUS,-1,0);
      semread = sem_open("/readsem",O_CREAT,0666,0); // init à 0
      semwrite = sem_open("/writesem",O_CREAT, 0666,1); // init à 1
      if ((pid = fork()) == -1) {perror("fork"); exit(1);}

      if (pid == 0)
            send(shared);
      else
            receive(shared);

      if (pid == 0) exit(0);
      for (i = 0; i < 2; i++)
            wait(0);
```

```
      printf("fin du programme\n");

      sem_close(semread);
      sem_close(semwrite);
      sem_unlink("/readsem");
      sem_unlink("/writesem");
      munmap (shared,50);
      return 0;
}
```

**Question 4.** Quelles modifications faut-il apporter pour que chacun des processus récepteurs affiche le message dans son intégralité ? Voici un exemple de résultat attendu :

```
Récepteur1> Je
Récepteur2> Je
Récepteur2> suis
Récepteur1> suis
..
```

Une solution passe par un deuxiéme sémaphore pour le deuxième lecteur. Avec un seul pour les deux lecteurs on ne s'en sort pas vraiment.

Le sem de l'écrivain démarre à 2, il débloque les deux lecteurs.

found2.cpp

```
#include <unistd.h>
#include <cstdlib>
#include <cstdio>
#include <cstring>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/mman.h>
#include <semaphore.h>
#include <fcntl.h>

int i, n = 6;
int pid;

sem_t * semread1;
sem_t * semread2;
sem_t * semwrite;

void send(char *buf) {
      const char *msg[] = {"Je", "suis", "un", "vilain", "programme", "\n"};
      for(i = 0; i < n; i++) {
            sem_wait(semwrite);
            sem_wait(semwrite);
            strcpy(buf, msg[i]);
            sem_post(semread1);
            sem_post(semread2);
      }
      exit(0);
}

void receive(char *buf) {
```

```
      if ((pid = fork()) == -1) {perror("fork"); exit(1);}
      for(i = 0; i < (n/2); i++) {
            if (pid != 0)
                  sem_wait(semread1);
            else
                  sem_wait(semread2);
            printf("%s\n", buf);
            sem_post(semwrite);
      }
}

int main()
{
      // mmap : address, size, protection, flags, filedescriptor, offset
      char *shared = (char *)
                  mmap(0,50,PROT_READ | PROT_WRITE,MAP_SHARED | MAP_ANONYMOUS,-1,0);
      semread1 = sem_open("/readsem1",O_CREAT,0666,0); // init à 0
      semread2 = sem_open("/readsem2",O_CREAT,0666,0); // init à 0
      semwrite = sem_open("/writesem",O_CREAT, 0666,2); // init à 2
      if ((pid = fork()) == -1) {perror("fork"); exit(1);}

      if (pid == 0)
            send(shared);
      else
            receive(shared);

      if (pid == 0) exit(0);
      for (i = 0; i < 2; i++)
            wait(0);
      printf("fin du programme\n");

      sem_close(semread1);
      sem_close(semread2);
      sem_close(semwrite);
      sem_unlink("/readsem1");
      sem_unlink("/readsem2");
      sem_unlink("/writesem");
      munmap (shared,50);
      return 0;
}
```

## 3 Primitive select

On fournit une classe `ServerSocket`basée sur le TD 7.

ServerSocket.h

```
#ifndef SRC_SERVERSOCKET_H_
#define SRC_SERVERSOCKET_H_
#include "Socket.h"
namespace pr {

class ServerSocket {
      int socketfd;
public :
      // Demarre l'ecoute sur le port donne
      ServerSocket(int port);
```

```
      int getFD() { return socketfd;}
      bool isOpen() const {return socketfd != -1;}

      Socket accept();
      void close();
};
} // ns pr
#endif /* SRC_SERVERSOCKET_H_ */
```

ServerSocket.cpp

```
#include "ServerSocket.h"
#include <cstring>
#include <unistd.h>
#include <iostream>
namespace pr {
ServerSocket::ServerSocket(int port) :socketfd(-1) {
      // create socket
      int fd = socket(AF_INET,SOCK_STREAM,0);
      if (fd == -1) {
            perror("create socket");
            return;
      }

      // bind
      struct sockaddr_in sin; /* Nom de la socket de connexion */
      memset(&sin, 0, sizeof(sin)); // utile ?
      sin.sin_family = AF_INET;
      sin.sin_addr.s_addr = htonl(INADDR_ANY); // on attend sur n'importe quelle interface
          de la machine
      sin.sin_port = htons(port); // host to network

      /* nommage, meme probleme de typage sockaddr que dans la Socket */
      if (bind(fd, (struct sockaddr *) &sin, sizeof(sin)) < 0) {
            perror("bind");
            // on veut le close de unistd, pas le close membre de cette classe
            ::close(fd);
            return;
      }

      // listen
      if (listen(fd, 50) < 0) {
            perror("listen");
            ::close(fd);
            return;
      }

      // success !
      socketfd = fd;
}

Socket ServerSocket::accept() {
      struct sockaddr_in exp;
      socklen_t len = sizeof(exp);
      // on qualifie sinon le compilo rale
      int scom = ::accept(socketfd, (struct sockaddr *) &exp, &len);
      if (scom < 0) {
            perror("accept");
      } else {
            // en appui sur operator << pour (sin_addr *) développé plus haut
```

```
            std::cout << "Accepted connection from " << &exp << std::endl;
      }
      return scom;
}

void ServerSocket::close() {
      if (socketfd != -1) {
            ::close(socketfd);
            socketfd = -1;
      }
}
} // ns pr
```

**Question 1.** Coder dans ServerSocket (fourni) un comportement tel que si on est bloqué dans `accept()` et qu'un autre thread appelle `close`, on sort de l'accept (en rendant une Socket non connectée `sockFD` vaut -1).

On référera aux numéros de lignes pour indiquer où faire les modifications.

Donc solution basée sur select et un pipe, cf aussi la discussion en fin de corrigé sur "stopServer" du TD7, dont cette question est une variante.

ServerSocket.cpp

```
#include "ServerSocket.h"
#include <cstring>
#include <unistd.h>
#include <iostream>

namespace pr {

// Evidemment ce sont des attributs de la classe,
// je les mets ici pour compiler ok avec le même header que la classe de l'énoncé
int pipefd[2];

ServerSocket::ServerSocket(int port) :socketfd(-1) {

      // create socket
      int fd = socket(AF_INET,SOCK_STREAM,0);
      if (fd == -1) {
            perror("create socket");
            return;
      }

      // bind
      struct sockaddr_in sin; /* Nom de la socket de connexion */

      memset(&sin, 0, sizeof(sin)); // utile ?
      sin.sin_family = AF_INET;
      sin.sin_addr.s_addr = htonl(INADDR_ANY); // on attend sur n'importe quelle
          interface de la machine
      sin.sin_port = htons(port); // host to network

      /* nommage, meme probleme de typage sockaddr que dans la Socket */
      if (bind(fd, (struct sockaddr *) &sin, sizeof(sin)) < 0) {
            perror("bind");
            // on veut le close de unistd, pas le close membre de cette classe
            ::close(fd);
            return;
      }
```

```
	// listen
	if (listen(fd, 50) < 0) {
		perror("listen");
		::close(fd);
		return;
	}

	// success !
	socketfd = fd;
}

Socket ServerSocket::accept() {
	struct sockaddr_in exp;
	socklen_t len = sizeof(exp);

	int readpipe = pipefd[0];

	// Construire un set pour le select (atente simultanee)
	fd_set read;
	FD_ZERO(&read);
	FD_SET(readpipe, &read);
	FD_SET(getFD(), &read);
	// premier argument = plus grand fildescriptor dans un des set + 1
	// on peut comprendre que fd_set est un genre de bitset, on donne sa taille ici.
	select(std::max(readpipe, getFD()) +1, &read, /*write*/nullptr, /*except*/nullptr,
	     /*timeout*/nullptr);

	// au retour du select, read est modifié pour contenir le ou les fd qui nous ont
	    reveillé
	if (FD_ISSET(readpipe,&read)) {
		std::cout << "asked to quit" << std::endl;
		return Socket();
	}
	// sinon ben c'est la socket qui nous a réveillé : un client s'est pointé


	// on qualifie sinon le compilo rale
	int scom = ::accept(socketfd, (struct sockaddr *) &exp, &len);
	if (scom < 0) {
		perror("accept");
	} else {
		// en appui sur operator << pour (sin_addr *) développé plus haut
		std::cout << "Accepted connection from " << &exp << std::endl;
	}
	return scom;
}

void ServerSocket::close() {
	if (socketfd != -1) {
		int n = 42; // arbitraire
		write(pipefd[1],&n,sizeof(int));
		::close(socketfd);
		::close(pipefd[0]);
		::close(pipefd[1]);
		socketfd = -1;
	}
}
```

```
}
```