SCIENCES SORBONNE UNIVERSITÉ

Examen Réparti 1 PR
Master 1 Informatique Nov 2018

UE 4I400

Année 2018-2019

2 heures – **Tout document papier autorisé**
Tout appareil de communication électronique interdit (téléphones...)
Clé USB en lecture seule autorisée.

Introduction

- Le barème est sur 20 points et est donné à titre indicatif.
- Dans le code C++ demandé vous prendrez le temps d'assurer que la compilation fonctionne.
- On vous fournit une archive contenant un projet eclipse CDT par exercice, qu'il faudra modifier. Décompresser cette archive dans votre home, de façon à avoir un dossier `~/exam/` et les sous dossiers `~/exam/exo1/src` ...
- Pour importer ces projets dans Eclipse, le plus facile :"File->Import->General->Existing Projects into Workspace", Pointer le dossier `/exam`, Eclipse doit voir 4 projets, tous les importer (sans cocher "copy into workspace").
- Si vous préférez utiliser un autre IDE ou la ligne de commande, on vous fournit dans chaque répertoire source un Makefile trivial. Attention à bien positionner dans ce cas le PATH :
  `export PATH=/usr/local/eclipseCPP/bin:$PATH`
  pour trouver notre compilo g++ récent.
- A la fin de la séance, fermez simplement votre session. On ira par script récupérer les fichiers dans ce fameux dossier `~/exam/`. Assurez vous donc de bien suivre ces instructions.
- Vous n'avez pas un accès complet à internet, mais si vous configurez le proxy de votre navigateur (proxy, port 3128, tous les protocoles) vous aurez accès au site `http://cppreference.com`

Le sujet est composé de quatre exercices indépendants qu'on pourra traiter dans l'ordre qu'on souhaite. Pour certaines questions il faut compléter un fichier texte avec vos réponses, pour la majorité des questions il s'agit de fournir un code compilable correct.

Certaines questions (modification de code fourni) exigent un commentaire en plus de la modification. Lors de la correction il sera tenu compte des commentaires accompagnant vos réponses, pas seulement du code lui même.

1 Compilation, Link, Debug (5 points)

Notions testées : connaissance du langage, de la chaîne de compilation, des outils de debug (gdb, valgrind).

On vous fournit un programme `exo1` composé de trois fichiers, implantant une manipulation d'une liste chaînée de string. Malheureusement ce code contient un nombre important de bugs et d'erreurs.

Question 1. Identifiez et corrigez les erreurs dans ce programme.

On recherche au total

- Cinq fautes plutôt de nature syntaxiques empêchant la compilation ou le link
- Trois fautes graves à l'éxécution entraînant le plus souvent un crash du programme
- Deux fautes de gestion mémoire incorrecte, mais ne plantant pas nécessairement le programme

Pour chaque faute, ajoutez un commentaire débutant par `//FAUTE :` et sur une ligne décrivez la faute au dessus de votre modification. Par exemple,

```
// FAUTE : i n'est pas initialisé
i = 0;
```

main.cpp

```cpp
#include "List.h"
#include <string>
#include <iostream>
#include <cstring>


int main () {

      std::string abc = "abc";
      char * str = new char [4];
      str[0] = 'a';
      str[1] = 'b';
      str[2] = 'c';
      // FAUTE : manque un '\0'
      str[3] = '\0';
      // FAUTE : size_t ne passera jamais en négatif
      int i = 0;

      if (! strcmp (str, abc.c_str())) {
            std::cout << "Equal !";
      }

      pr::List list;
      list.push_front(abc);
      list.push_front(abc);

      std::cout << "Liste : " << list << std::endl;
      std::cout << "Taille : " << list.size() << std::endl;

      // Affiche à l'envers
      for (i= list.size() - 1 ; i >= 0 ; i--) {
            std::cout << "elt " << i << ": " << list[i] << std::endl;
      }

      // FAUTE : NE SURTOUT PAS FAIRE CA !
      // liberer les char de la chaine
// for (char *cp = str ; *cp ; cp++) {
//  delete cp;
// }

      // et la chaine elle meme
      delete [] str;

}
```

List.h

```hpp
#ifndef SRC_LIST_H_
#define SRC_LIST_H_

#include <cstddef>
#include <string>
#include <ostream>

namespace pr {


class Chainon {
public :
      std::string data;
      Chainon * next;
      Chainon (const std::string & data, Chainon * next=nullptr);
      size_t length() ;
      void print (std::ostream & os) const;
};


class List {
public:

      Chainon * tete;

      List(): tete(nullptr) {}

      ~List() {
            for (Chainon * c = tete ; c ; ) {
                  Chainon * tmp = c->next;
                  delete c;
                  c = tmp;
            }
      }

      const std::string & operator[] (size_t index) const ;

      void push_back (const std::string& val) ;

      // FAUTE : corps dans le cpp
      void push_front (const std::string& val) ;

      bool empty() ;

      size_t size() const ;


      class ConstListIte {
            const Chainon * cur;
      public:
            // default ctor = end()
            ConstListIte (const Chainon * cur=nullptr) : cur(cur) {}

            const auto & operator* () const {
                  return cur->data;
            }
            const auto * operator-> () const {
                  return & cur->data;
            }
```

```
            ConstListIte & operator++ () {
                  cur = cur->next;
                  return *this;
            }
            bool operator!= (const ConstListIte &o) const {
                  return cur != o.cur;
            }
      };

      typedef ConstListIte const_iterator;

      const_iterator begin() const {
            return tete;
      }
      const_iterator end() const {
            return nullptr;
      }
};


std::ostream & operator<< (std::ostream & os, const List & vec) ;

} /* namespace pr */

#endif /* SRC_LIST_H_ */
```

List.cpp

```
// FAUTE : manque include
#include "List.h"

namespace pr {

// Chainon

Chainon::Chainon (const std::string & data, Chainon * next):data(data),next(next) {};

size_t Chainon::length() {
      size_t len = 1;
      if (next != nullptr) {
            len += next->length();
      }
      // FAUTE : stack overflow !
      return len;
}

// FAUTE : manque qualifier const
void Chainon::print (std::ostream & os) const {
      os << data ;
      if (next != nullptr) {
            os << ", ";
            // FAUTE MEMOIRE : acces nullptr
            next->print(os);
      }
}


// List
```

```
const std::string & List::operator[] (size_t index) const {
      auto it = begin();
      for (size_t i=0; i < index ; i++) {
            ++it;
      }
      return *it;
}

void List::push_back (const std::string& val) {
      if (tete == nullptr) {
            tete = new Chainon(val);
      } else {
            Chainon * fin = tete;
            while (fin->next) {
                  fin = fin->next;
            }
            fin->next = new Chainon(val);
      }
}

void List::push_front (const std::string& val) {
      tete = new Chainon(val,tete);
}

// FAUTE : manque List::
bool List::empty() {
      return tete == nullptr;
}

size_t List::size() const {
      if (tete == nullptr) {
            return 0;
      } else {
            return tete->length();
      }
}

// FAUTE : doit etre dans le namespace pr
std::ostream & operator<< (std::ostream & os, const pr::List & vec)
{
      os << "[";
      if (vec.tete != nullptr) {
            vec.tete->print (os) ;
      }
      os << "]";
      return os;
}

} // namespace pr
```

Les fautes 10 chaque :

- Manque include "List.h"
- print est const dans .cpp
- List::empty est qualifié
- namespace pr inclut operator«
- double implantation de push front
- length provoque un SO

- print provoque un NPE
- main contient une boucle infinie
- abc alloué sans '0' terminal
- delete correct du tableau abcManque include List.h

2 Le Rendez Vous (5 points)

Notions testées : Création et synchronisations de threads.

Un rendez-vous est un modèle de synchronisation dans lequel N thread doivent mutuellement s'attendre avant de poursuivre leur éxécution, ce modèle est confortable dans les algorithmes fonctionnant par vagues.

Question 1. Ecrire une classe `RendezVous` munie d'un constructeur prenant le nombre N de participants, et offrant une opération `void meet()` qui est bloquante pour les $N-1$ premiers threads, mais où le N-éme thread qui invoque $meet()$ débloque tous les N threads. Le $RendezVous$ retrouve alors son état initial, les $N-1$ prochaines invocations à $meet$ seront de nouveau bloquantes. On demande de séparer la déclaration de la classe (.h) de son implémentation (.cpp).

RDV.h

```
#ifndef EXO2COR_RDV_H_
#define EXO2COR_RDV_H_

#include <string>
#include <mutex>
#include <condition_variable>

namespace pr {

class RendezVous {
      const size_t N;
      size_t cur;
      std::mutex m;
      std::condition_variable cv;
public :
      RendezVous(size_t N);
      void meet();
};

}

#endif
```

RDV.cpp

```
#include "RDV.h"

namespace pr {

RendezVous::RendezVous(size_t N) : N(N),cur(0) {}

void RendezVous::meet() {
      std::unique_lock<std::mutex> l(m);
      ++cur;
```

```
      if (cur < N) {
            cv.wait(l);
      } else {
            cv.notify_all();
            cur = 0;
      }
}

}
```

Question 2. A l'aide d'une instance de cette classe, écrivez un *main* qui crée $N-1$ threads, chacun devant itérer trois fois le comportement :

- Afficher "Début" et son numero d'ordre de création
- dormir pour une durée alétoire comprise entre 0.1 et 1 seconde,
- puis se synchroniser avec les autres threads à l'aide d'un rendez-vous
- Afficher "Fin" et son numero d'ordre de création

avant de se terminer.

Le thread principal doit également participer au trois rendez vous, puis se terminer *proprement*.

On ne cherchera pas à protéger la sortie standard, i.e. on accepte que les sorties des différents threads soient entrelacées.

main.cpp

```
#include "RDV.h"
#include <thread>
#include <iostream>
#include <vector>

void worker (pr::RendezVous * prdv, int ind) {
      for (int i=0; i < 3 ; i++) {
            std::cout << "Debut " << ind << std::endl;
            std::this_thread::sleep_for( std::chrono::milliseconds((rand() % 1000)) );
            prdv->meet();
            std::cout << "Fin " << ind << std::endl;
      }

}

int main () {
      const int N = 4;
      // construire un RendezVous pour N participants
      pr::RendezVous rdv (N);

      // instancier des threads
      std::vector<std::thread> threads;
      threads.reserve(N-1);
      for (int i = 0 ; i < N -1; i++) {
            threads.emplace_back(std::thread(worker,&rdv,i));
      }

      // participer au rendez vous trois fois
      for (int i=0; i < 3 ; i++) {
            rdv.meet();
```

```
      }

      // sortie propre
      for (auto & t : threads) {
            t.join();
      }
      std::cout << "fini" << std::endl;
      return 0;
}
```

3 Fork et Wait (5 points)

Notions testées : fork, wait lecture de code.

On considère le programme suivant :

fork.cpp

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

#define N 3

int main () {
      pid_t pids [N];
      memset(pids,0,N*sizeof(pid_t));
      pid_t p;

      for (int i = 0 ; i < N ; i++) {
            p = fork();
            pids [i] = p;
            if (p && i == N-1) {
                  for (int j = 0 ; j < N ; j++) {
                        if (pids[j] != 0)
                              waitpid(pids[j],0,0);
                  }
            } else if (!p) {
                  pids [i] = fork();
            }
      }
      for (int j = 0 ; j < N ; j++) {
            if (pids[j] != 0)
                  waitpid(pids[j],0,0);
      }

      return 0;
}
```

Question 1. Combien de processus sont créés par ce programme (en comptant le processus initial) pour la valeur donnée N=3. Combien de processus sont crées en fonction de N ? Avec N = 3, à un instant donné, combien de processus au maximum peuvent être en cours d'exécution ?

Vous placerez vos réponses dans le fichier "exo3/reponse.txt".

```
[ythierry@localhost exo3cor]$ ./a.out
fin de  pid=19041 ppid=19037
fin de  pid=19048 ppid=19043
fin de  pid=19050 ppid=19044

fin de  pid=19052 ppid=19046
fin de  pid=19037 ppid=19034
fin de  pid=19054 ppid=19047

fin de  pid=19053 ppid=19051
fin de  pid=19043 ppid=19035
fin de  pid=19046 ppid=19040

fin de  pid=19044 ppid=19036
fin de  pid=19055 ppid=19049
fin de  pid=19047 ppid=19038

fin de  pid=19051 ppid=19045
fin de  pid=19060 ppid=19058
fin de  pid=19059 ppid=19057

fin de  pid=19040 ppid=19036
fin de  pid=19049 ppid=19039
fin de  pid=19045 ppid=19039

fin de  pid=19058 ppid=19056
fin de  pid=19057 ppid=19042
fin de  pid=19036 ppid=19034

fin de  pid=19039 ppid=19035
fin de  pid=19056 ppid=19042
fin de  pid=19042 ppid=19038

fin de  pid=19038 ppid=19035
fin de  pid=19035 ppid=19034
fin de  pid=19034 ppid=4227
```

Donc père initial, obtient N (ici 3) fils, puis les attend.

Chaque fils engendre immédiatement un petit fils identique.

On a donc 3 processus après la boucle i=0 (on stoppe là si N=1).

Pour i=1, si on rentre dans la boucle on a donc 3 processus qui vont fork, soit 6 total. Les trois fils se dupliquent instantanément soit 9 processus. (on stoppe là si N=2)

Pour i=2, chacun des 9 processus engendre un fils et un petit fils, soit 18 processus de plus. Total 27 processus. (on stoppe là si N=3)

On généralise à 3^N.

Les wait sont placés de façon a maximiser le parallélisme, on ne wait jamais avant d'avoir fini de faire des fork. Donc le nombre maximum de processus en parallèle est aussi 3^N.

Question 2. Les *waitpid* ne sont pas actuellement utilisés à bon escient, le programme exploite le fait que *waitpid* rend -1 et ne bloque pas l'appelant si l'on wait un *pid* qui ne corresponde pas à un de ses fils. Modifiez le programme pour ne faire que les wait utiles et nécessaires (*waitpid* ne doit jamais rendre -1).

Donc deux changements principaux :
Dans la boucle, après le fork, nettoyer le début du tableau de pids (= pid de mes frères/oncles).
Après la boucle, n'invoquer la boucle de wait que si on est sorti du for avec le statut "fils", i.e. p vaut 0.

fork.cpp

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

#define N 3

int main () {
      pid_t pids [N];
      memset(pids,0,N*sizeof(pid_t));
      pid_t p;

      for (int i = 0 ; i < N ; i++) {
            p = fork();
            pids [i] = p;
            if (p && i == N-1) {
                  for (int j = 0 ; j < N ; j++) {
                        if (pids[j] != 0)
                              if (waitpid(pids[j],0,0) == -1) {
                                    std::cout << "oops pid=" << getpid() << " childpid="
                                          << pids[j] << std::endl;
                              }
                  }
            } else if (!p) {
                  pids [i] = fork();
                  // Attention ce sont mes grand freres, ou mes oncles pour le nouveau
                      fils
                  for (int j = 0 ; j < i ; j++) {
                        pids[j] =0;
                  }
            }
      }
      // Seuls les fils de la dernière étape sortent "mal" de la boucle
      if (!p) {
            for (int j = 0 ; j < N ; j++) {
                  if (pids[j] != 0)
                        if (waitpid(pids[j],0,0) == -1) {
                              std::cout << "oops2 pid=" << getpid() << " childpid=" <<
                                  pids[j] << std::endl;
                        }
            }
      }
      std::cout << "fin de pid=" << getpid() << " ppid=" << getppid() << std::endl;

      // JUSTE POUR CONTROLE DE CORRECTION
      int pid;
      if ( (pid = wait(0)) != -1) {
            std::cout << "oops3 pid=" << getpid() << " childpid=" << pid << std::endl;
      }

      return 0;
}
```

> Il y a d'autres stratégies possibles.

4 Synchronisations et Pool de Thread (5 points)

Notions testées : synchronisations, thread, pool de thread.

On considère un pool de thread comme au TD 4, utilisant une `Queue<Job>` pour représenter le travail à exécuter. Le Pool et la classe Queue sont fournis, ainsi qu'un main.

On considère une application où chaque tâche (Job) peut potentiellement engendrer des nouveaux jobs, sans que le client puisse prévoir combien de tâches au total vont être nécessaires.

Par exemple, dans une application de ray tracing, chaque forme rencontrée par le rayon pourrait engendrer de nouvelles tâches pour calculer l'éclairage, les ombres, les reflets...

Pour l'exercice on ne se soucie pas de savoir précisément ce que font les tâches, il suffit de savoir qu'elles peuvent en engendrer de nouvelles.

Question 1. Implantez la méthode "void waitUntilDone()" de la classe Pool. Cette méthode doit être bloquante jusqu'à ce que

- a) la queue soit vide
- b) tous les threads du pool soient bloqués en attente de nouveaux jobs.

On doit donc garantir qu'aucun des threads esclaves n'est en train de travailler, sinon il pourrait engendrer des nouvelles tâches.

On suggère d'écrire les synchronisations au sein de la classe Queue ; la version de "waitUntilDone()" de Pool ne faisant que déléguer à celle que vous ajouterez dans Queue.

On pourra utiliser un compteur comptabilisant le nombre de threads actuellement en train de traiter une tâche (donc tous ceux qui ne sont pas bloqués sur *pop*).

Vous ajouterez un commentaire `// AJOUT` ou `// MODIF` avant chaque modification apportée aux fichiers.

> Comme suggéré, on utilise un compteur dans Queue + une nouvelle condition.
>
> Le thread principal, au fil de la création des threads esclaves incrémente le compteur. C'est important que ce soit le "père" et non les threads eux-même qui s'inscrivent, sinon on peut se terminer tout de suite au début du programme (i.e. avant de laisser les threads s'inscrire, on voit 0 thread actifs).
>
> Chaque thread, juste avant de faire un wait dans pop (il sait qu'il va wait), décrémente le compteur. Si jamais le compteur atteint 0 à cet instant (où la queue est vide sinon on ne serait pas en train de se préparer à wait), on notifie sur la nouvelle condition.
>
> Chaque thread, juste en ressortant du wait dans pop incrémente le compteur (s'il ne gagne pas la course du while englobant le wait, il le redécrémente avant de redormir.)
>
> Le client insère un unique Job (la graine) et fait un waitUntilDone.

Queue.h

```
#ifndef SRC_QUEUE_H_
#define SRC_QUEUE_H_

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

template <typename T>
class Queue {
      T ** tab;
      const size_t allocsize;
```

```
	size_t begin;
	size_t sz;
	bool isBlocking;
	mutable std::mutex m;
	std::condition_variable cv;

	size_t active;
	std::condition_variable cv_inactive;


	// fonctions private, sans protection mutex
	bool empty() const {
		return sz == 0;
	}
	bool full() const {
		return sz == allocsize;
	}
public:
	// AJOUT INIT 0
	Queue(size_t size) :allocsize(size), begin(0), sz(0),isBlocking(true),active(0) {
		tab = new T*[size];
		::memset(tab, 0, size * sizeof(T*));
	}
	size_t size() const {
		std::unique_lock<std::mutex> lg(m);
		return sz;
	}
	void setBlocking(bool isBlocking) {
		std::unique_lock<std::mutex> lck(m);
		this->isBlocking = isBlocking;
		cv.notify_all();
	}
	T* pop() {
		std::unique_lock<std::mutex> lck(m);
		while (empty() && isBlocking) {
			// on dort en attendant des jobs
			// AJOUT
			stopActive();
			if (active == 0) {
				cv_inactive.notify_all();
			}

			cv.wait(lck);

			// AJOUT
			startActive();
		}
		if (empty()) {
			return nullptr;
		}
		if (full()) {
			cv.notify_all();
		}
		auto ret = tab[begin];
		tab[begin] = nullptr;
		sz--;
		begin = (begin + 1) % allocsize;
		return ret;
	}
```

```
      bool push(T* elt) {
            std::unique_lock<std::mutex> lg(m);
            while (full() && isBlocking) {
                  cv.wait(lg);
            }
            if (full()) {
                  return false;
            }
            if (empty()) {
                  cv.notify_all();
            }
            tab[(begin + sz) % allocsize] = elt;
            sz++;
            return true;
      }
      ~Queue() {
            // ?? lock a priori inutile, ne pas detruire si on travaille encore avec
            for (size_t i = 0; i < sz; i++) {
                  auto ind = (begin + i) % allocsize;
                  delete tab[ind];
            }
            delete[] tab;
      }

      // AJOUTS
      void startActive () {
            active++;
      }
      void stopActive () {
            active--;
      }

      // AJOUT
      void waitUntilFinished() {
            std::unique_lock<std::mutex> lg(m);
            while (active > 0 && ! empty()) {
                  cv_inactive.wait(lg);
            }
      }
};



#endif /* SRC_QUEUE_H_ */
```

Pool.cpp

```
#include "Pool.h"

namespace pr {

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

            if (j == nullptr) {
                  // on est non bloquant = il faut sortir
                  return;
```

```
            }

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

void Pool::start (int nbthread) {
      threads.reserve(nbthread);
      for (int i=0 ; i < nbthread ; i++) {
            threads.emplace_back(poolWorker, &queue, &stopping);
            // AJOUT
            queue.startActive();
      }
}
// Ajout a la fin du TD, pour la terminaison
void Pool::stop() {
      stopping = true;
      queue.setBlocking(false);
      for (auto & t : threads) {
            t.join();
      }
      threads.clear();
}
Pool::~Pool() {
      stop();
}
void Pool::submit (Job * job) {
      queue.push(job);
}


void Pool::waitUntilDone() {
      // A COMPLETER
      queue.waitUntilFinished();
}

}
```