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;
```

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).

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.

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".

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).

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.