SCIENCES SORBONNE UNIVERSITÉ

Examen Réparti 2 PSCR
Master 1 Informatique Jan 2021

UE 4I400

Année 2020-2021

2 heures – **Tout document papier autorisé**
Tout appareil de communication électronique interdit (téléphones...)

Introduction

- Le barème est sur 20 points et est donné à titre indicatif.
- Dans le code C++ demandé vous vous efforcerez d'écrire du code compilable.
- On ne demande dans le code ni les include, ni les qualifications de namespace `std::`
- On se permettra de fusionner déclarations et implantations, placez le code des opérations directement dans la classe.
- On considère que les appels système n'échouent pas.
- Si on vous demande des modifications sur un code, ne recopiez pas tout, référez aux numéros de lignes.
- En particulier sur les extraits de code, vous vous efforcerez d'écrire lisiblement et de limiter les ratures.

Le sujet est composé de trois exercices indépendants qu'on pourra traiter dans l'ordre qu'on souhaite.

1 Processus et Parallélisme (5 points)

On souhaite calculer le nombre de lignes dans un ensemble de fichiers en utilisant du parallélisme de processus. On propose pour cela d'invoquer la commande `wc -l` qui compte les lignes d'un fichier dans un processus séparé pour chaque fichier argument, puis de sommer les résultats. On va utiliser des pipe pour faire communiquer les processus, mais pour simplifier on ne demande pas de coder les invocations à `close` sur les filedescriptor pas ou plus utilisés.

Question 1. (2 points)Ecrire une fonction `void launchWC (int pipefdw, const char * path)` qui lance la commande `/usr/bin/wc -l` dans un nouveau processus sur le fichier "path", de façon à ce que sa sortie soit capturé par le file descriptor `pipefdw` représentant l'extrémité écriture d'un pipe. On n'attend pas la fin du processus fils dans cette fonction, le fils est lancé de façon asynchrone. On s'appuyera sur les primitives système `fork, exec, dup2`.

Question 2. (1 point) Ecrire une fonction `int parseInt(int pipefdr)` qui consomme dans le file descriptor `pipefdr` désignant l'extrémité en lecture d'un pipe une chaine de caractères (on supposera moins de 10 caractères) qui représente un entier et rend sa valeur. Le pipe ne contient que cette chaine de caractères. On s'appuiera sur la primitive système `read`, et on pourra utiliser si nécessaire `int atoi(char *)` de la lib standard du C.

Question 3. (2 points) Assembler ces éléments dans un main qui crée un pipe par nom de fichier passé en argument, et invoque `launchWC` sur ce fichier, puis collecte les résultats à l'aide de `parseInt`, affiche la somme sur sa sortie standard, et sort proprement (en faisant un wait pour chaque fils créé). On pourra par exemple utiliser un `vector<int> readfd` pour stocker les extrémités en lecture des pipes instanciés.

Scanné avec CamScanner

2 Questions de cours (3,5 points)

Question 1. (1,5 points) Parmi les appels système suivants, lesquels sont succeptibles d'être bloquants, i.e. le système ne rendra peut-être pas la main avant un délai conséquent ?
1.socket, 2.accept, 3.listen, 4.connect, 5.read, 6.write, 7.fork, 8.sleep, 9.wait, 10.kill, 11.sigaction, 12.sigprocmask, 13.sigsuspend

Question 2. (1 point) On considère l'utilisation de l'appel système `void alarm(int sec)` qui permet de déclencher un signal dans "sec" secondes, en poursuivant son exécution. Comment obtenir cet effet : recevoir un signal au bout de "sec" secondes tout en poursuivant son exécution sans cet appel système ?

Question 3. (1 point) En TCP sur une socket on peut-on utiliser à la fois "send/recv" et "read-/write". Pourquoi en UDP ne peut-on *pas* utiliser "read/write" ? Quelle API faut-il utiliser à la place ?

3 Arbre concurrents (11,5 points)

On considère un arbre binaire de recherche, chaque noeud de l'arbre (on utilisera une seule classe) porte deux pointeurs vers les sous-arbres gauche et droit (qui peuvent être `nullptr` désignant un sous-arbre vide), ainsi qu'une string. Le sous-arbre gauche (resp. droit) contient des string lexicographiquement inférieures strictement (respectivement supérieures strictement) à la string du noeud courant. On a un ensemble au sens mathématique, un seul noeud de l'arbre porte une string donnée.

On donne le code de la classe `Node` qui réalise linterface `INode`:

INode.h

```
1  #pragma once
2
3  #include <string>
4
5  // C++ style interface
6  class INode {
7  public :
8        virtual bool insert (const std::string & word) =0;
9        virtual bool contains (const std::string & word) const =0;
10 };
```

- `bool insert(const string & s)` : essaie d'insérer la string *s* et rend vrai si une insertion a lieu, et faux sinon.
- `bool contains(const string & s) const` : teste si la string *s* est contenue dans l'arbre.

Node.h

```
1  #pragma once
2
3  #include <string>
4  #include "INode.h"
5
6  class Node : public INode {
7        INode * left;
8        INode * right;
9        std::string s;
10 public :
11       Node (const std::string & s):left(nullptr),right(nullptr),s(s){}
12       bool insert (const std::string & word) {
13             if (word == s) {
```

Scanné avec CamScanner

train d'éxécuter une méthode de INode.)

Question 4. (2 points) Donnez le code de cette classe `NodeBFL`, son constructeur `NodeBFL(INode * deco)`, son destructeur (qui doit libérer l'arbre détenu), et ses opérations qui sont réalisées par délégation (i.e. contains invoque contains sur l'arbre décoré). On utilisera des `mutex` et/ou des `condition_variable` pour assurer l'exclusion mutuelle.

On souhaite être plus flexible, et supporter des accès concurrents en lecture (méthode contains) tout en garantissant l'exclusion mutuelle entre deux écritures (insert) ou entre une lecture et une écriture. Pour cela on propose de réaliser un reader/writer lock dont le squelette est donné ici. Le contrat consiste à invoquer `startRead` (potentiellement bloquant) avant de démarrer une lecture et `endRead` quand elle est terminée, ou similairement à utiliser la paire `startWrite/endWrite` pour réaliser une écriture. Les opérations `start` sont parfois bloquantes, les opérations `end` débloquent les threads en attente.

Node.h

```
1  #pragma once
2
3  class RWLock {
4
5  public :
6        void startRead () {
7        }
8        void endRead () {
9        }
10       void startWrite () {
11       }
12       void endWrite () {
13       }
14 }
```

Question 5. (2 points) Compléter cette classe, on recommande de comptabiliser le nombre de lecteurs et d'écrivains et d'utiliser une seule condition pour gérer les notifications utiles.

Question 6. (1,5 points) On souhaite créer un nouveau décorateur `NodeRW` (sur le même modèle que la question 4) qui utilise ce nouveau mécanisme `RWLock` pour protéger un `INode` donné des accès concurrents. Définir les attributs et le code des méthodes `insert` et `contains` de cette classe.

Question 7. (2,5 points) On souhaite explorer une autre piste, l'ajout d'un mutex dans chaque noeud de l'abre, i.e. déclarer un attribut `std::mutex m;` dans la classe `Node` fournie par l'énoncé directement. On veut avec ce grain plus fin augmenter les possibilités d'accès concurrents, par exemple deux threads qui insèrent en prarallèle sur le sous-arbre gauche et droit d'un noeud est un scenario parfaitement acceptable, mais que les stratégies actuelles par décoration ne supportent pas. Expliquez où placer les instructions `m.lock()` et `m.unlock()` dans les opérations `insert` et `contains`de la classe Node fournie pour maximiser les accès concurrents tout en assurant l'absence de data-race. On recommande dans cette question de ne pas utiliser le mécanisme `unique_lock` mais plutôt de préciser manuellement les `lock/unlock` pertinents.

On fera référence aux numéros de ligne plutot que de trop recopier l'énoncé (e.g. "avant la ligne 10 ajouter m.lock()"). Vous ne traiterez que le cas du "sous-arbre gauche" entièrement. On supposera donc que l'autre cas est symétrique.

Scanné avec CamScanner

```
14
15                return false;
16          } else if ( word < s ) {
17                if (left == nullptr) {
18                      left = new Node(word);
19                      return true;
20                } else {
21                      return left->insert(word);
22                }
23          } else {
24                if (right == nullptr) {
25                      right = new Node(word);
26                      return true;
27                } else {
28                      return right->insert(word);
29                }
30          }
31    }
32    bool contains (const std::string & word) const {
33          if (word == s) {
34                return true;
35          } else if (word < s) {
36                if (left == nullptr) {
37                      return false;
38                } else {
39                      return left->contains(word);
40                }
41          } else {
42                if (right == nullptr) {
43                      return false;
44                } else {
45                      return right->contains(word);
46                }
47          }
48    }
49    Node (const Node & other) {
50          left = new Node(*other.left);
51          right = new Node(*other.right);
52          s = other.s;
53    }
54    ~Node() {
55          delete left;
56          delete right;
57    }
};
```

Question 1. (1 point) Que signifie le `const` placé après la déclaration de l'opération `contains` ? Peut-on ajouter `const` aussi sur `insert` ou cela cause-t-il des difficultés de compilation (lesquelles) ?

Question 2. (1,5 point) Si l'on est dans un contexte multi-thread, expliquez une faute qui pourrait se produire si l'on essaie de faire deux insertions en concurrence. On expliquera un entrelacement possible de deux thread qui mène à un résultat incorrect.

Question 3. (1 point) L'utilisation d'attributs `atomic` pour les pointeurs `left` et `right` pourrait-elle suffire pour résoudre le problème d'accès concurrents identifiés à la question précédente ? Justifiez votre réponse.

On propose de décorer INode pour permettre de construire des abres qui sont thread-safe. On s'appuie sur le *design pattern* Decorator, vous allez définir une classe `NodeBFL` qui réalise l'interface `INode` et qui contient un pointeur vers un `INode` (l'objet décoré dans le pattern), qu'on lui passe à la construction. Ce node "Big Fat Lock" doit protéger des accès concurrents l'arbre qu'il détient ; dans cette question on souhaite obtenir une exclusion mutuelle complète : un seul thread à la fois en

Scanné avec CamScanner

SCIENCES SORBONNE UNIVERSITÉ

Examen Réparti 1 PSCR
Master 1 Informatique Déc 2020

UE 4I400

Année 2020-2021

2 heures – **Tout document papier autorisé**
Tout appareil de communication électronique interdit (téléphones...)

Introduction

- Le barème est sur 20 points et est donné à titre indicatif.
- Dans le code C++ demandé vous vous efforcerez d'écrire du code compilable.
- On ne demande dans le code ni les include, ni les qualifications de namespace `std::`
- On se permettra de fusionner déclarations et implantations, placez le code des opérations directement dans la classe.
- On considère que les appels système n'échouent pas.
- Si on vous demande des modifications sur un code, ne recopiez pas tout, référez aux numéros de lignes.

Le sujet est composé de trois exercices indépendants qu'on pourra traiter dans l'ordre qu'on souhaite.

1 ResultatCalcul (9 points)

1.1 Classe ResultatCalcul

La classe `ResultatCalcul` héberge un résultat (un `std::string` pour s'épargner une version générique) qui doit être calculé par un thread esclave et accédé par un thread maitre. Il permet au maître d'attendre la complétion du travail du ou des esclaves.

On propose l'API suivante :

- `const string & getResult() const` : Rend la valeur stockée dès qu'elle est disponible. L'invocation bloque l'appelant si nécessaire jusqu'à ce que la valeur devienne disponible.
- `bool isSet() const` : est vrai si et seulement si la valeur a déjà été affectée, i.e. `getResult` n'est plus bloquant.
- `void setResult(const string & res)` : permet de positionner la valeur stockée, débloque les threads éventuellement bloqués sur `getResult`.. On ne doit invoquer `setResult` qu'une seule fois sur une instance donnée de `ResultatCalcul`, si on l'invoque de nouveau cela n'a aucun effet.

Question 1. (3 points) Implanter cette classe avec son constructeur par défaut et si nécessaire un destructeur.

1.2 Utilisation de la classe

On souhaite ecrire un programme qui :

Scanné avec CamScanner

- crée un vecteur de `ResultatCalcul` de taille N
- construit N threads esclave, chaque esclave va calculer un des résultats : la valeur calculée sera simplement "slaveXX" où XX est son *numéro d'ordre de création*.
- affiche le contenu du vecteur, une valeur par ligne, quand il est calculé
- se termine proprement.

Question 2. (2 points) Proposez un programme C++ (un `main`) qui a ce comportement et qui maximise le parallélisme. On interdit l'utilisation de variables globales.

Quand N est grand, on constate que créer trop de threads n'est pas rentable. On souhaite au contraire ne construire que K threads esclave (au lieu de N), en supposant K petit devant N. Chaque esclave dans cette version doit traiter N/K tâches (on ignore l'arrondi, N est multiple de K).

On fait donc un découpage par lots de taille égale des résultats à calculer, chaque lot ayant une taille N/K.

Question 3. (2 points) Modifiez votre programme pour avoir ce comportement.

1.3 Comparaison à un Pool

On voudrait comparer le comportement de votre programme à une version qui utiliserait un pool de threads, comme élaboré en TD/TME, et construirait un `Job` pour chaque résultat à calculer. (NB : on ne demande pas le code de cette version !)

Question 4. (1,5 points) Expliquez en quoi la solution avec un Pool diffère de la solution proposée en Q3 si on suppose que le calcul de chaque résultat a une durée fixe (homogène) ou au contraire que le calcul d'un résultat prend une durée aléatoires difficile à prédire. On s'intéresse à l'utilisation des ressources de la machine en temps total de calcul sur une machine multi-core.

2 Arbre, pointeurs, gestion mémoire en C++(5 points)

On considère un arbre binaire de recherche, chaque noeud de l'arbre (on utilisera une seule classe) porte deux pointeurs vers les sous-arbres gauche et droit (qui peuvent être `nullptr` désignant un sous-arbre vide), ainsi qu'une string. Le sous-arbre gauche (resp. droit) contient des string lexicographiquement inférieures strictement (respectivement supérieures strictement) à la string du noeud courant. On a un ensemble au sens mathématique, un seul noeud de l'arbre porte une string donnée.

Question 1. Donnez le code de la classe `Node` :

- (0,5 pt) ses attributs left, right et value.
- (0,5 pt) un constructeur à un seul argument (la string) qui initialise les sous arbres gauche et droit à vide.
- (2 pt) `bool insert(const string & s)` : essaie d'insérer la string s et rend vrai si une insertion a lieu, et faux sinon. Soit le noeud courant porte déjà la string à insérer, soit il faut réaliser une récursion sur le sous arbre adapté (en comparant s à la valeur stockée). Si la récursion atteint un sous arbre vide on crée un nouveau noeud qu'on accroche dans l'arbre.
- (1 pt) un constructeur par copie, qui suit la signature standard `Node(const Node &)` et copie tout l'arbre. On ne demande pas l'opérateur d'affectation qui serait similaire.
- (1 pt) un destructeur qui doit désallouer tout l'arbre dont le noeud courant est la racine.

3 Arbres de processus (6 points)

On considère le code suivant.

Scanné avec CamScanner

fork.cpp

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

// a faire varier
#define N 4

int main () {
      int i = -1;
      int j = -1;
      for (i=0; i < N; ++i) {
            if (fork() == 0) {
                  for (j=i+1; j < N; ++j) {
                        if (fork() != 0) {
                              break;
                        }
                  }
                  break;
            }
      }
      std::cout << "Processus " << getpid() << std::endl;
      return 0;
}
```

Question 1. (2 points) Donnez le nombre total (sans compter le main) de processus engendrés pour N=1, N=2, N=3, N=4.

Question 2. (1 point) Exprimez le nombre processus engendrés comme une fonction de N.

Question 3. (1,5 points) Tout en maximisant le parallélisme, ajouter les instructions `wait` utiles juste avant le `return 0;` pour garantir que le main ne se termine qu'après que l'ensemble de ses descendants soient terminés. NB: On évitera d'invoquer `wait` plus de fois que l'on a crée de processus fils. On recommande de s'appuyer sur les valeurs de i et j pour décider du comportement à adopter après les boucles, ou d'introduire des nouvelles variables au choix.

Question 4. (1,5 points) Les instructions de synchronisation `wait` sont placées actuellement juste après l'affichage; on décide plutôt de placer ces instruction **avant** l'affichage sur `std::cout`. Quelles sont les garanties sur l'ordre dans lequel les affichages seront produits dans ces deux solutions ?

Scanné avec CamScanner