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 arguement, 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`.

Node.h

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

#include <vector>
```

```cpp
#include <iostream>

void launchWC (int fd, const char * file) {
  if (fork()==0) {
    //std::cout << "running wc on :" << file << std::endl;
    dup2(fd, STDOUT_FILENO);
    execl("/usr/bin/wc","/usr/bin/wc","-l",file,NULL);
    perror("exec failed");
  }
}

int parseInt (int fd) {
  char buff [10];
  read(fd,buff,10);
  return atoi(buff);
}

int main (int argc, char ** argv) {

  std::vector<int> fd;
  for (int i=1; i < argc; i++) {
    int pfd[2];
    pipe(pfd);
    fd.push_back(pfd[0]);
    launchWC(pfd[1],argv[i]);
  }

  int sum = 0;
  for (int p : fd) {
    sum += parseInt(p);
  }
  std::cout << sum << std::endl;
  for (int i=1; i < argc; i++) {
    wait(NULL);
  }
  return 0;
}
```

Barème:

- 20 % fork
- 40 % dup2
- 40 % exec

On sera indulgent sur la syntaxe du exec tand qu'on y voit passer tous les arguments.

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.

Corrigé dans Q précédente.

Barème:

- 20 % un buffer
- 50 % read
- 30 % atoi

> Le code qui lirait sizeof(int) bytes directement à l'adresse d'un entier vaut zéro (copié collé mal choisi du support de TD).

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.

> Corrigé dans Q précédente.
> Barème:
> - 50 % boucle sur les arguments avec 15 % création d'un pipe, 15 % stockage, 20 % invocation à launchWC
> - 30 % boucle pour invoquer parseInt, APRES avoir lancé tous les fils
> - 20 % boucle sur wait

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

> Sont bloquants : `2. accept, 4. connect, 5. read, 6. write, 8. sleep, 9. wait, 13. sigsuspend`
> Barème sur 70%, à saisir tel quel dans la feuille. +10% pour chaque appel bloquant cité, -10% par appel non bloquant cité.

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 ?

> Il faut faire un deuxième flot de contrôle, thread ou processus, qui va sleep le temps indiqué puis m'envoyer le signal avec un bon `kill(pid,SIGALRM)`.
> 40 % on a clairement un deuxième flot de contrôle
> 30 % on cite sleep
> 30 % on cite kill

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 ?

> La socket TCP est mode PACKET, donc non connectée.
> on utilise "sendTo/recvFrom"
> 50 % on n'est pas connecté en UDP !
> 50 % sendTo/recvFrom

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

```
#pragma once

#include <string>

// C++ style interface
class INode {
public :
      virtual bool insert (const std::string & word) =0;
      virtual bool contains (const std::string & word) const =0;
};
```

- `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

```
#pragma once

#include <string>
#include "INode.h"

class Node : public INode {
      INode * left;
      INode * right;
      std::string s;
public :
      Node (const std::string & s):left(nullptr),right(nullptr),s(s){}
      bool insert (const std::string & word) {
            if (word == s) {
                  return false;
            } else if ( word < s ) {
                  if (left == nullptr) {
                        left = new Node(word);
                        return true;
                  } else {
                        return left->insert(word);
                  }
            } else {
                  if (right == nullptr) {
                        right = new Node(word);
                        return true;
                  } else {
                        return right->insert(word);
                  }
            }
      }
      bool contains (const std::string & word) const {
```

```
32            if (word == s) {
33                  return true;
34            } else if (word < s) {
35                  if (left == nullptr) {
36                        return false;
37                  } else {
38                        return left->contains(word);
39                  }
40            } else {
41                  if (right == nullptr) {
42                        return false;
43                  } else {
44                        return right->contains(word);
45                  }
46            }
47      }
48      Node (const Node & other) {
49            left = new Node(*other.left);
50            right = new Node(*other.right);
51            s = other.s;
52      }
53      ~Node() {
54            delete left;
55            delete right;
56      }
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) ?

> Cela signifie que le sujet this est read-only, il n'est pas modifié quand on invoque contains.
>
> Si on ajoute const sur insert, on aura des fautes de compilation sur les instructions l17 et l24 où l'on affecte à left et right (donc this->left et this->right) ce qui est illégal dans un contexte où this est un pointeur const.
>
> BarèmeTODO :
>
> - 40 % const sur contains expliqué
> - 40 % const sur insert ne marche pas
> - 20 % la faute engendrée est clairement identifiée

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.

> on est sur le même noeud, left vaut nullptr.
>
> T1 : teste left == nullptr => true
>
> il commute/on entrelace T2
>
> T2 teste left == nullptr => true
>
> les deux insertions vont rendre true, mais on a une race condition, sans doute un seul des deux nouveaux fils sera inséré (un des deux va écrire dans left en dernier), mais de toutes façons du moment qu'on écrit à la même adresse en concurrence sans atomic on est dans du UB (undefined behavior).
>
> Barème :
>
> - 40 % on propose une séquence, même fausse qui pourrait être une faute
> - 40 % on est clairement dans le contexte **d'un noeud précis** dans les deux thread et on va accéder au même champ

- 20 % explication lisible, correcte et complète

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.

Non ça ne suffit pas, on voudrait atomiquement tester si le fils est null, et lui affecter une valeur si c'est le cas. Avec un compare and swap et en réecrivant un peu le code on pourrait y arriver mais ce n'est pas trivial, e.g. on va créer dans certains cas un Node pour finalement si le CAS échoue ne pas s'en servir.

De façon plus immédiate, on voit qu'on fait : `if (teste sur var) modifierVar;]` ce qui est quelque chose que atomic ne permet pas de traiter en général (cf TD sur la Banque).

Barème :

- 50 % non ça ne suffit pas
- 50 % on identifie la séquence test+affectation comme racine du problème

On propose de décorer lNode 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 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 note la similarité avec e.g. `Collections.synchronizedXXX` de l'API de Java. C'est basique comme approche mais effectivement ça garantit un accès mutli-thread correct sans se poser de questions.

Node.h

```
#pragma once

#include <string>
#include <mutex>
#include "INode.h"

class NodeBFL : public INode {
      INode * deco;
      mutable std::mutex m;
public :
      NodeBFL (INode * deco):deco(deco){}
      bool insert (const std::string & word) {
            std::unique_lock<std::mutex> l(m);
            return deco->insert(word);
      }
      bool contains (const std::string & word) const {
            std::unique_lock<std::mutex> l(m);
            return deco->contains(word);
      }
      ~NodeBFL() {
            delete deco;
      }
```

```
};
```

Barème :

- 10 % attribut INode * deco (mais on acceptera aussi Node *),
- 10 % mutex mutable (5 % si non mutable)
- 10 % ctor
- 60 = 30 % *2 : insert et contains, on doit voir la délégation (15) et la protection du mutex (15)
- 10 % dtor

Fautes fréquentes :
-10 % si on ne déclare pas ": public INode" et/ou on n'a pas correctement @Override les méthodes de INode, e.g. contains n'est pas const.

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

```
#pragma once

class RWLock {

public :
      void startRead () {
      }
      void endRead () {
      }
      void startWrite () {
      }
      void endWrite () {
      }
}
```

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.

Node.h

```
#pragma once

class RWLock {
      int nbR;
      int nbW; // ou juste un bool
      std::mutex m;
      std::condition_variable cv;
public :
      void startRead () {
            unique_lock<std::mutex> l(m);
            while (nbW > 0)
                  cv.wait(l);
```

```
            nbR++;
      }
      void endRead () {
            unique_lock<std::mutex> l(m);
            nbR--;
            cv.notify_all();
      }
      void startWrite () {
            unique_lock<std::mutex> l(m);
            while (nbR > 0 || nbW > 0)
                  cv.wait(l);
            nbW++;
      }
      void endWrite () {
            unique_lock<std::mutex> l(m);
            nbW--;
            cv.notify_all();
      }
}
```

Barème :

- 20 % attributs, cv+mutex, deux compteurs
- 80 = 4* 20 % par méthode. On enlève 10 % par élément manquant ou mal fait dans la méthode.
- start => protection mutex, condition du wait dans un while, wait lui-même, incrément
- end => protection mutex, décrément, notify

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.

Node.h

```
#pragma once

#include <string>
#include <mutex>
#include "INode.h"

class NodeRW : public INode {
      INode * deco;
      mutable RWLock l;
public :
      NodeBFL (INode * deco):deco(deco){}
      bool insert (const std::string & word) {
            l.startWrite();
            bool b = deco->insert(word);
            l.endWrite();
            return b;
      }
      bool contains (const std::string & word) const {
            l.startRead();
            bool b = deco->contains(word);
            l.endRead();
            return b;
      }
```

```
24        ~NodeRW() {
25              delete deco;
26        }
27  };
```

Barème : On essaie de ne pas repénaliser des décorateurs faux à la question 4. Les points sont donc centrés sur la partie spécifique à la question, et pas sur la structure d'un décorateur.

- 10 % attributs, INode * deco (mais on acceptera aussi Node *),
- 10 % attribut RWlock mutable (-5 % si non mutable)
- 80 = 40 % *2 : insert et contains, on doit voir la délégation (10) et la protection du RWlock (30)

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 `containsde` 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.

Node.h

```
1   #pragma once
2
3   #include <string>
4   #include <mutex>
5   #include "INode.h"
6
7   class NodeMutex : public INode {
8         INode * left;
9         INode * right;
10        std::string s;
11        mutable std::mutex m;
12  public :
13        NodeMutex (const std::string & s):left(nullptr),right(nullptr),s(s){}
14        bool insert (const std::string & word) {
15              if (word == s) {
16                    return false;
17              } else if ( word < s ) {
18                    m.lock(); // avant 16
19                    if (left == nullptr) {
20                          left = new Node(word);
21                          m.unlock(); // avant 18
22                          return true;
23                    } else {
24                          m.unlock(); // avant 20
25                          return left->insert(word);
26                    }
27              } else {
28                    m.lock(); // avant 23
```

```
                if (right == nullptr) {
                        right = new Node(word);
                        m.unlock(); // avant 25
                        return true;
                } else {
                        m.unlock(); // avant 27
                        return right->insert(word);
                }
        }
    }
    bool contains (const std::string & word) const {
        if (word == s) {
                return true;
        } else if (word < s) {
                m.lock(); // avant 35
                if (left == nullptr) {
                        m.unlock(); // avant 36
                        return false;
                } else {
                        m.unlock(); // avant 38
                        return left->contains(word);
                }
        } else {
                m.lock(); // avant 41
                if (right == nullptr) {
                        m.unlock(); // avant 42
                        return false;
                } else {
                        m.unlock(); // avant 44
                        return right->contains(word);
                }
        }
    }
    NodeMutex (const Node & other) :left(nullptr),right(nullptr) {
        if (other.left) left = new Node(*other.left);
        if (other.right) right = new Node(*other.right);
        s = other.s;
    }
    ~NodeMutex() {
        delete left;
        delete right;
    }
};
```

Barème:

- 20 % il n'y a pas de data race dans la solution
- 20 % le test de la string se fait hors contrôle du mutex
- 20 % les tests + affectation sont bien protégés par une section critique.
- 40 % les récursions se font SANS lock à la main

On peut faire des solutions incorrectes (e.g. un uniquelock à l'entrée de chaque méthode) qui auront donc un peu de points (pas de data race + section critique = 40 %).

Faute fréquente : pas de mutable => -5 %.