

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'appuiera 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 chaîne de caractères (on supposera moins de 10 caractères) qui représente un entier et rend sa valeur. Le pipe ne contient que cette chaîne de caractères. On s'appuiera sur la primitive système `read`, et on pourra utiliser si nécessaire `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.

2 Questions de cours (3,5 points)

Question 1. (1,5 points) Parmi les appels système suivants, lesquels sont susceptibles 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 l'interface `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  public :
8      INode * left;
9      INode * right;
10     std::string s;
11     Node (const std::string & s):left(nullptr),right(nullptr),s(s){}
12     bool insert (const std::string & word) {
13         if (word == s) {

```

```

14         return false;
15     } else if ( word < s ) {
16         if (left == nullptr) {
17             left = new Node(word);
18             return true;
19         } else {
20             return left->insert(word);
21         }
22     } else {
23         if (right == nullptr) {
24             right = new Node(word);
25             return true;
26         } else {
27             return right->insert(word);
28         }
29     }
30 }
31 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) ?

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 `lNode` pour permettre de construire des arbres 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'exécuter 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'arbre, 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 parallèle sur le sous-arbre gauche et droit d'un noeud est un scénario 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 plutôt 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.