SCIENCES SORBONNE UNIVERSITÉ

Examen Réparti 1 PSCR
Master 1 Informatique Nov 2021

UE 4I400

Année 2021-2022

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 3 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.
- 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 `https://cppreference.com`

Le sujet est composé de 2 exercices indépendants qu'on pourra traiter dans l'ordre qu'on souhaite. Pour la majorité des questions il s'agit de fournir un code compilable correct.

1 Blocking Queue sur Linked List (15 points)

On souhaite construire une classe qui représente une queue FIFO (first-in first-out) basée sur une implantation à l'aide d'une liste chaînée. Au fil des questions on va en faire une classe de synchronisation avec un comportement bloquant.

On peut supposer une partie résolue pour coder la suivante.

1.1 Liste Chainée en mode FIFO (4 points)

La structure de données est FIFO, on va appeler la classe `LinkedQueue`.

Il offre deux principales opérations : `T * take ()` (extrait le prochain élément de la queue) et `bool put(T * t)` (insère un élément dans la queue).

Il dispose d'un pointeur sur le chainon de tete "head" et sur la fin la liste "last". Il stocke également une capacité (passée à la construction) et une taille courante.

On propose de partir de la réalisation suivante conçue pour stocker des pointeurs. Les pointeurs sont censés être valides ; c'est donc au client qui fait put (le *producteur*) de faire un `new` pour obtenir le pointeur à insérer. Le client qui fait take (le *consommateur*) doit donc symétriquement `delete` l'entrée quand il a terminé de s'en servir.

Notons qu'au lieu d'avoir $head == nullptr$ quand la liste est vide (comme on a fait en TD), le choix fait ici est de toujours avoir un chainon de tête dont l'item vaut $nullptr$ ($head- > item == nullptr$ est un invariant de la classe), et un chaînon de queue dont le next vaut nullptr ($last- > next == nullptr$ est un invariant de la classe). Quand la queue est vide $head == last$. Cette stratégie permet d'avoir moins de cas aux limites à traiter et de décorréler la notion de tête et de queue :les deux restent toujours des pointeurs valides.

LinkedQueue.hh

```
#pragma once
#include <string> // size_t

namespace pr {
/**
 * A bounded queue based on linked nodes.
 * This queue orders elements FIFO (first-in-first-out).
 * The head of the queue is that element that has been on the
 * queue the longest time.
 * The tail of the queue is that element that has been on the
 * queue the shortest time. New elements
 * are inserted at the tail of the queue, and the queue retrieval
 * operations obtain elements at the head of the queue.
 * Linked nodes are
 * dynamically created upon each insertion unless this would bring the
 * queue above capacity.
 *
 * @param <E> the type of elements held in this queue
 */
template<typename E>
class LinkedQueue {
   /**
    * Linked list node class/struct
    */
    struct Node {
      /** always nullptr for the head of the list */
      E * item;

      /**
       * One of:
       * - the real successor Node
       * - nullptr, meaning there is no successor (this is the last node)
       */
      Node * next;

      Node(E * x):item(x),next(nullptr) {}
   };

   /** The capacity bound, or Integer.MAX_VALUE if none */
   const size_t capacity;
   /** Current number of elements */
   size_t count;
   /**
    * Head of linked list.
    * Invariant: head->item == nullptr
    */
   Node *head;
   /**
```

```
49      * Tail of linked list.
50      * Invariant: last->next == nullptr
51      */
52     Node *last;
53
54     /**
55      * Links node at end of queue.
56      *
57      * @param node the node
58      */
59     void enqueue(Node * node) {
60        // assert last->next == nullptr;
61
62        // TODO Question 1
63
64     }
65
66     /**
67      * Removes a node from head of queue.
68      * @return the node
69      */
70      E * dequeue() {
71        // assert head->item == nullptr;
72
73        // TODO Question 1
74     }
75
76 public :
77
78     /**
79      * Creates a queue with the given (fixed) capacity.
80      * @param capacity the capacity of this queue
81      */
82      LinkedQueue(size_t capacity) {
83       // if (capacity <= 0) throw "IllegalArgumentException()";
84
85          // TODO Question 1
86     }
87
88     ~LinkedQueue() {
89
90        // TODO Question 1
91
92     }
93     /**
94      * Returns the number of elements in this queue.
95      */
96     size_t size() const {
97        return count;
98     }
99
100    /**
101     * Returns the number of additional elements that this queue can ideally
102     * (in the absence of memory or resource constraints) accept without
103     * blocking. This is always equal to the initial capacity of this queue
104     * less the current size of this queue.
105     */
106    size_t remainingCapacity() const {
107       return capacity - count;
108    }
109
```

```
110    /**
111     * Inserts the specified element at the tail of this queue
112     *
113     * // TODO Question 2
114     * or return false if full
115     *
116     */
117    bool put(E * e) {
118       // if (e == nullptr) throw "NullPointerInserted()";
119       Node * node = new Node(e);
120       enqueue(node);
121       count++;
122       return true;
123    }
124
125    /**
126     * Takes and returns the element at the head of the queue
127     *
128     * // TODO Question 2
129     * or return nullptr if empty.
130     *
131     */
132    E * take() {
133       E * x = dequeue();
134       count--;
135       return x;
136     }
137
138 } ; // end class
139
140 } // namespace
```

Question 1. (3 points) Complétez les parties manquantes de la classe, indiquées par "//TODO". On demande donc l'implantation :

- du constructeur,
- du destructeur,
- des fonctions "enqueue" et "dequeue".

Enqueue doit correctement mettre à jour la liste pour insérer en fin de liste le noeud argument. Dequeue doit lire et décaler la tête de la liste, et ne pas oublier de libérer la mémoire du chaînon lu. On demande de ne pas modifier le reste de la classe à ce stade, uniquement ces quatre fonctions. On fournit un petit main basique qui utilise la queue.

LinkedQueue.hh

```
1     /**
2      * Links node at end of queue.
3      *
4      * @param node the node
5      */
6     void enqueue(Node * node) {
7        // assert last.next == nullptr;
8        last = last->next = node;
9     }
10
11    /**
12     * Removes a node from head of queue.
13     *
```

```
    * @return the node
    */
    E * dequeue() {
      // assert head.item == nullptr;
      Node * h = head;
      Node * first = h->next;
      delete h; // help GC, C++ style
      head = first;
      E * x = first->item;
      first->item = nullptr;
      return x;
   }


public :
   /**
    * Creates a {@code LinkedBlockingQueue} with the given (fixed) capacity.
    *
    * @param capacity the capacity of this queue
    * @throws IllegalArgumentException if {@code capacity} is not greater
    * than zero
    */
    LinkedQueue(size_t capacity):capacity(capacity),count(0) {
     // if (capacity <= 0) throw "IllegalArgumentException()";
      last = head = new Node(nullptr);
   }

   ~LinkedQueue() {
      while (head != nullptr) {
            Node * h = head;
            head = head->next;
            delete h->item; // ok if nullptr
            delete h;
      }
   }
```

Barème :

- 20 % le constructeur : 10 initialise capa et count, 10 initialise head et tail sur le même chainon vide.
- 20 % le destructeur nettoie tout. 15 pour le nettoyage, on donne les points même si on a fait la version recursive, ie. le destructeur de Node est implanté et recursif. 5 % pour nettoyer aussi les item.
- 20 % enqueue : 10% on chaine le nouvel élément, 10 % on décale last
- 40 % dequeue : 10% décale head, 10% delete l'ancien, 10 % récupère effectivement l'item, 10% restaure l'invariant head->item = nullptr

Question 2. (1 point)Actuellement, la queue n'est pas protégée contre les débordements de sous ou sur capacité. Modifiez le comportement pour que `put` rende un booléen : *false* et pas d'effet si la queue est pleine et *true* si l'insertion est réussie. Modifiez aussi `take` pour qu'il rende un *nullptr* si la queue est vide (au lieu de la corrompre). On ajoute donc au contrat de la classe qu'il est interdit d'insérer des *nullptr*.

LinkedQueue.hh

```
   bool put(E * e) {
```

```
        // if (e == nullptr) throw "NullPointerInserted()";
        if (count == capacity) {
              return false;
        }
        Node * node = new Node(e);
        enqueue(node);
        count++;
        return true;
    }


    E * take() {
        if (count ==0) {
              return nullptr;
        }
        E * x = dequeue();
        count--;
        return x;
     }
```

Barème :

- 50 % put protégée
- 50 % take protégée

1.2 Multi-thread safe, Synchronisation (6 points)

Copiez votre classe actuelle, dans un nouveau fichier "NaiveBlockingQueue.hh" et renommer la classe en NaiveBlockingQueue dans cette version.

Question 3. (1,5 points) A l'aide d'un seul mutex, modifier cette classe afin de la rendre utilisable dans un contexte multi-threadé concurrent (rendre la classe *MultiThread-safe*).

Ajouts attributs (condition pour la question suivante) :

NaiveBlockingQueue.hh

```
    /** Lock held by take, poll, etc */
    mutable std::mutex mutex;

    /** Wait queue for waiting takes */
    std::condition_variable cond;
```

Comportement thread safe :

NaiveBlockingQueue.hh

```
    size_t size() const {
        std::unique_lock<std::mutex> l(mutex);
        return count;
```

+ la même chose dans remainingCapacity, put et take.

Barème :

- 20 % la classe stocke un mutex mutable
- 30 % put protégée
- 30 % take protégée

- 20 % size,capacity protégée

Il faut que le lock englobe le corps et garantisse l'exclusion mutuelle. Réponses correctes avec lock/unlock à la main ok.

Attention à ne pas blinder enqueue/dequeue ! -40% si on a réussi à créer un deadlock. -10% de pénalité pour le recursive_mutex qui évite l'interblocagemais est mal adapté.

Question 4. (2,5 points) A l'aide d'une seule variable de condition, modifier cette classe afin d'en faire une classe de synchronisation au comportement bloquant : *take* est bloquant si la queue est vide, et *put* est bloquant si la capacité est atteinte.

Ajout de la condition en attribut

Comportement bloquant :

NaiveBlockingQueue.hh

```
1   bool put(E * e) {
2      if (e == nullptr) throw "NullPointerException()";
3      std::unique_lock<std::mutex> l(mutex);
4      Node * node = new Node(e);
5      {
6            // scope for lock
7            std::unique_lock<std::mutex> l(mutex);
8
9         while (count == capacity) {
10           cond.wait(l);
11        }
12        enqueue(node);
13        count++;
14
15     } // unlock before notifications
16
17     // just ping the condition it will sort itself out
18     cond.notify_all();
19     return true;
20  }
21
22
23  E * take() /* throws InterruptedException*/ {
24     E * x = nullptr;
25     {
26           // scope for lock
27        std::unique_lock<std::mutex> l(mutex);
28        while (count == 0) {
29           cond.wait(l);
30        }
31        x = dequeue();
32        count--;
33     }// unlock before notifications
34
35     // just ping the condition it will sort itself out
36     cond.notify_all();
37
38     return x;
39  }
```

Barème :

- 20 % la classe stocke une condition var

- 40 % put protégée. 30 % on wait correctement sur la bonne condition(20%), sous lock(10%). 10 % on notifie
- 40 % take protégé. 30 % on wait correctement sur la bonne condition(20%), sous lock(10%). 10 % on notifie

Pas de points en plus ou en moins pour notifications hors ou dans la section critique.

Question 5. (2 points) Ecrivez un programme principal qui crée 3 thread producteur et 3 thread consommateur, chacun devant respectivement ajouter 2000 ou retirer 2000 éléments de la queue, initialisée avec une capacité de 100. Le programme doit ensuite se terminer proprement et sans fautes ou fuites mémoire. On utilisera une queue de *std* :: *string*; le contenu des string est sans importance (on peut simplement mettre des $''toto''$ dans la queue).

main.cpp

```cpp
#include "LinkedBlockingQueue.hh"
#include "LinkedQueue.hh"
#include <thread>
#include <vector>
#include <iostream>

// Fichier à completer
void producer (int n, pr::LinkedBlockingQueue<std::string> & queue) {
      for (int i=0; i < n ; i++) {
            queue.put(new std::string("P"));
      }
      std::cout << "Producer "<< n << " finished" << std::endl;
}

void consumer(int n, pr::LinkedBlockingQueue<std::string> & queue) {
      for (int i=0; i < n ; i++) {
            std::string * s = queue.take();
            // std::cout << *s;
            // avoid I/O it will force commutation
            delete s;
      }
      std::cout << "Consumer "<< n << " finished" << std::endl;
}

int main () {
      // blocking queue
      {
            // a faire varier
            const int NBOPERATION = 2000;
            const int QCAPACITY = 100;
            const int NBCONSPROD = 3;

            pr::LinkedBlockingQueue<std::string> queue (QCAPACITY);

            std::vector<std::thread> threads;
            threads.reserve(NBCONSPROD*2);

            // instancier NBREAD threads lecteurs
            for (int i=0; i < NBCONSPROD ; i++) {
                  threads.emplace_back(consumer,NBOPERATION, std::ref(queue));
                  threads.emplace_back(producer,NBOPERATION, std::ref(queue));
            }

```

```
            // sortie propre
            for (auto & t: threads)
                t.join();
        }

        return 0;
}
```

Barème :

- 40 % les fonctions producteur/consommateur font la boucle demandée
- 40 % création des threads, dont 20% sur le fait de passer correctement une instance commune par pointeur ou std::ref au thread.
- 20 % join

1.3 Verrouillage fin (5 points)

Copiez votre classe actuelle, dans un nouveau fichier "LinkedBlockingQueue.hh" et renommer la classe en LinkedBlockingQueue dans cette version.

Il est actuellement impossible qu'un producteur et un consommateur accèdent simultanément à la queue. Pourtant, quand la file n'est ni pleine ni vide, chacun devrait pouvoir manipuler son extrémité de la liste chaînée sans causer d'interférences.

On propose d'implanter la stratégie suivante à l'aide d'un atomic et de deux mutex et deux conditions distinctes: *notEmpty* et *notFull*.

1. On souhaite que les producteurs soient en exclusion mutuelle ; un seul thread actif à la fois dans *put*. On souhaite aussi que les consommateurs soient en exclusion mutuelle ; un seul thread actif à la fois dans *take*. Cependant on veut autoriser un thread actif dans put et dans take simultanément.
2. Les threads qui tentent de *put* sur queue pleine doivent attendre sur la condition *notFull* que la file ne soit plus pleine
3. Les threads qui tentent de *take* sur queue vide doivent attendre sur la condition *notEmpty* que file ne soit plus vide
4. Un thread consommateur qui vient de réussir à *take* alors que la file était pleine avant notifiera un producteur (s'il y en a en attente).
5. Un thread consommateur qui vient de réussir à *take* notifiera le consommateur suivant (s'il y en a en attente) à condition qu'il ne vienne pas de vider la queue en consommant le dernier élément.
6. Un thread producteur qui vient de réussir à *put* alors que la file était vide avant notifiera un consommateur (s'il y en a en attente).
7. Un thread producteur qui vient de réussir à *put* notifiera le producteur suivant (s'il y en a en attente) à condition qu'il ne vienne pas de remplir la queue.

De plus, comme le remplissage *count* de la la file est manipulé conjointement par producteur et consommateur, on propose d'utiliser un `atomic`.

Toutes les notifications seront des notifications individuelles ($notify\_one()$). La partie "s'il y en a en attente" ne se teste pas, on notifie et s'il n'y a personne en wait ça ne fait rien simplement.

Question 6. (1 point) Expliquez pourquoi si on vient de tester que la file n'est pas pleine dans un producteur qui fait put et qu'on se fait "doubler" par un consommateur qui vient modifier *count*, le comportement du producteur va rester correct. (On répondra dans le fichier "questions.txt")

C'est parce que on wait que $count < capacity$, si c'est vrai et qu'on se fait doubler par un consommateur qui fait $count - -$, ça va rester vrai. On ne peut pas se faire doubler par un autre thread qui fait put, on a un mutex pour ça.

Barème :

- 40 % la réponse indique que l'on a vaguement compris que l'autre ne peut qu'améliorer
- 40 % réponse claire et audible, justifiée, avec un exemple...
- 20% réponse indique qu'on a peur normalement de faire des protections avec deux mutex, qui montre que l'étudiant a compris pourquoi on pose cette question à cet instant. On précise le fait que notre mutex nous protège des autres put potentiels...

Question 7. (1 point) Expliquez ce qui va se produire dans le scenario suivant, deux consommateurs C1 et C2 tentent de chacun de consommer un message sur une file vide, ensuite on passe la main à un producteur P qui parvient à insérer trois éléments dans la queue d'affilée. Expliquer comment vont se passer les notifications dans ce scenario sachant que l'on utilise $notify\_one()$ pour toutes les notifications. (On répondra dans le fichier "questions.txt")

Donc C1 essaie de take et s'endort sur notEmpty.

C2 idem s'endort sur notEmpty.

P passe et pose trois item; le premier déclenche le test "on était vide avant", on notifie "notEmpty", un de C1 ou C23" est reveillé. Les deux items suivants n'entrainent pas de notification utile, on notifie notFull deux fois mais personne n'attend.

C1 (ou C2) se réveille et consomme un item, vu que après consommation on est encore notEmpty, on notifie donc la condition ce qui réveille l'autre consommateur.

On a donc une chaine de notifications, avec un consommateur qui notifie le suivant.

Barème :

- 30 % on exhibe un scenario de façon audible, même si ce n'est pas le bon vraiment. l'exercice est compris.
- 30 % clarté et pertinence de la réponse
- 40% on a bien eu une chaine de notifications dans le scenario, où C1 réveille C2.

Question 8. (3 points) Suivant ces instructions, programmer cette synchronisation fine dans la classe *LinkedBlockingQueue*.

LinkedBlockingQueue.hh

```
1   /** Current number of elements */
2   std::atomic<size_t> count;
3
4   /** Lock held by take, poll, etc */
5   std::mutex takeLock;
6
7   /** Wait queue for waiting takes */
8   std::condition_variable notEmpty;
9
10  /** Lock held by put, offer, etc */
11  std::mutex putLock;
12
13  /** Wait queue for waiting puts */
14  std::condition_variable notFull;
```

LinkedBlockingQueue.hh

```
1   void put(E * e) {
2      if (e == nullptr) throw "NullPointerException()";
3      Node * node = new Node(e);
4      size_t c;
5      {
6            // scope for lock
7            std::unique_lock<std::mutex> l(putLock);
8         /*
9          * Note that count is used in wait guard even though it is
10         * not protected by lock. This works because count can
11         * only decrease at this point (all other puts are shut
12         * out by lock), and we (or some other waiting put) are
13         * signalled if it ever changes from capacity. Similarly
14         * for all other uses of count in other wait guards.
15         */
16        while (count == capacity) {
17           notFull.wait(l);
18        }
19        enqueue(node);
20        c = count++; // atomic getAndIncrement
21
22     } // unlock before notifications
23
24     // notify the next producer if any, there is room !
25     if (c+1 < capacity)
26       notFull.notify_one();
27
28     // we were empty before this put, there is now work for consumers.
29     if (c==0)
30        notEmpty.notify_one();
31  }
32
33
34  E * take() /* throws InterruptedException*/ {
35      E * x = nullptr;
36      size_t c;
37      {
38           // scope for lock
39         std::unique_lock<std::mutex> l(takeLock);
40         while (count == 0) {
41            notEmpty.wait(l);
42         }
43         x = dequeue();
44         c = count--; // atomic getAndDecrement
45      }// unlock before notifications
46
47      // notify the next consumer if any, there is still work !
48      if (c > 1)
49         notEmpty.notify_one();
50
51      // we were full before this take, there is now space for producers.
52      if (c==capacity)
53         notFull.notify_one();
54      return x;
55   }
```

Barème :

- 20 % la classe stocke deux mutex et un atomic
- 40 % put : 10% mutex, 10% wait, 10% notif 1, 10% notif 2
- 40 % put : 10% mutex, 10% wait, 10% notif 1, 10% notif 2

pas de points sur le fait de sortir le lock de remainingCapacity (on a atomic).

On ne notes pas vraiment comment sont testés les conditions pour notifier, donc on n'exige pas la forme (correcte) du corrigé "c= count++", à partir du moment où l'on teste à peu près la bonne chose on accepte.

pas de points en plus ou en moins pour les section critiques serrées n'incluant pas les notifications.

2 Fork (5 points)

Question 9. (2,5 points) Dans le fichier main.cpp, écrire un programme qui crée N processus (N lu dans son premier argument ligne de commande), chaque fils doit créer $N/2$ processus (division entière), chacun de ces fils doit lui même créer N/4 processus...etc.

Chaque processus doit afficher son *pid*, celui de son père, ainsi que le nombre de fils qu'il a crée. On souhaite imposer que les affichages soient fait dans l'ordre inverse de parenté; l'affichage des fils doit préceder celui du père. Le main ne doit se terminer que quand tout les processus sont terminés. On souhaite maximiser le parallélisme potentiel.

main.cpp

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

using namespace std;



int main(int argc, const char ** argv) {
      if (argc < 2) {
            return 1;
      }
      int N = atoi(argv[1]);

      for (int i=0; i < N ; ) {
            if (fork() == 0) {
                  // fils
                  N/=2;
                  i=0;
            } else {
                  i++;
            }
      }


      for (int i=0; i < N ; i++) {
            wait(nullptr);
      }

      std::cout << "Process " << getpid() << " parent="<< getppid() << " with nbsons="
          << N << std::endl;
```

```
33
34        return 0;
35  }
```

Barème :

- 20 % le main crée N fils
- 20 % les fils eux même continuent dans la boucle
- 20 % les indices sont justes, ça répond à la question.
- 20 % wait
- 20 % l'affichage dont 10 % sur le fait de le placer après le wait.

Question 10. (1 points) Donner le nombre de processus engendrés au total pour N=1, N=2, N=3, N=4, N=8. (On répondra dans le fichier *questions.txt*).

```
$ ./exo2 0 | wc -l
1
$ ./exo2 1 | wc -l
2
$ ./exo2 2 | wc -l
5
$ ./exo2 3 | wc -l
7
$ ./exo2 4 | wc -l
21
$ ./exo2 8 | wc -l
169
```

Barème :

- 20 % par réponse * 5

On accepte le compte décalé de 1 (manque le main).

Question 11. (1,5 points) Dans le fichier *main2.cpp*, implanter une version récursive ; chaque fils doit lui-même invoquer le programme courant (i.e. argv[0]) à l'aide d'une des variantes de "exec" en passant un argument N diminué de moitié. On pourra consulter le *man* pour se remémorer l'API, on recommande la variante *execl* a priori.

main2.cpp

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

int main(int argc, const char ** argv) {
      if (argc < 2) {
            return 1;
      }
      int N = atoi(argv[1]);

      for (int i=0; i < N ; ) {
```

```
14              if (fork() == 0) {
15                    // fils
16                    N/=2;
17                    execl(argv[0],argv[0],std::to_string(N).c_str(),NULL);
18              } else {
19                    i++;
20              }
21        }
22
23
24        for (int i=0; i < N ; i++) {
25              wait(nullptr);
26        }
27
28        std::cout << "Process " << getpid() << " parent="<< getppid() << " with nbsons="
              << N << std::endl;
29
30        return 0;
31  }
```

Barème :

- 30 % on essaie d'exec
- 30 % on s'en sort sur les arguments int à convertir en string
- 40 % l'invocation est correcte, l'execution du code marches et donne la même chose que v1.