

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 Chaînée en mode FIFO (4 points)

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

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

Il dispose d'un pointeur sur le chaînon 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 chaînon 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 décorrélérer la notion de tête et de queue : les deux restent toujours des pointeurs valides.

LinkedQueue.hh

```

1  #pragma once
2  #include <string> // size_t
3
4  namespace pr {
5  /**
6   * A bounded queue based on linked nodes.
7   * This queue orders elements FIFO (first-in-first-out).
8   * The head of the queue is that element that has been on the
9   * queue the longest time.
10  * The tail of the queue is that element that has been on the
11  * queue the shortest time. New elements
12  * are inserted at the tail of the queue, and the queue retrieval
13  * operations obtain elements at the head of the queue.
14  * Linked nodes are
15  * dynamically created upon each insertion unless this would bring the
16  * queue above capacity.
17  *
18  * @param <E> the type of elements held in this queue
19  */
20  template<typename E>
21  class LinkedQueue {
22  /**
23   * Linked list node class/struct
24   */
25   struct Node {
26       /** always nullptr for the head of the list */
27       E * item;
28
29       /**
30        * One of:
31        * - the real successor Node
32        * - nullptr, meaning there is no successor (this is the last node)
33        */
34       Node * next;
35
36       Node(E * x):item(x),next(nullptr) {}
37   };
38
39   /** The capacity bound, or Integer.MAX_VALUE if none */
40   const size_t capacity;
41   /** Current number of elements */
42   size_t count;
43   /**
44    * Head of linked list.
45    * Invariant: head->item == nullptr
46    */
47   Node *head;
48   /**

```

```

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.

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*.

1.2 Multi-thread safe, Synchronisation (6 points)

Copiez votre classe actuelle, dans un nouveau fichier “NaiveBlockingQueue.hh” et renommez 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*).

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.

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).

1.3 Verrouillage fin (5 points)

Copiez votre classe actuelle, dans un nouveau fichier "LinkedBlockingQueue.hh" et renommez 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 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")

Question 7. (1 point) Expliquez ce qui va se produire dans le scénario 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. Expliquez comment vont se passer les notifications dans ce scénario sachant que l'on utilise *notify_one()* pour toutes les notifications. (On répondra dans le fichier "questions.txt")

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

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 créé. On souhaite imposer que les affichages soient fait dans l'ordre inverse de parenté; l'affichage des fils doit précéder celui du père. Le main ne doit se terminer que quand tout les processus sont terminés. On souhaite maximiser le parallélisme potentiel.

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*).

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.