**Лабораторная работа 5**

**11.04.2024**

**Уважаемые студенты!**

**Отчет по этому занятию следует представить в двух вариантах:**

**1) в конце занятия, чтобы подтвердить свое присутствие и активность на занятии;**

**2) к следующему занятию в завершенном виде.**

**Алгоритмы синхронизации потоков**

* Грубая (Coarse-grained) синхронизация;
* Тонкая (Fine-grained) синхронизация;
* Оптимистичная (Optimistic) синхронизация;
* Ленивая (Lazy) синхронизация;
* Неблокирующая (Nonblocking) синхронизация.

**Использованные источники**

1. Maurice Herlihy, Nir Shavit, Victor Luchangco, Michael Spear – The Art of Multiprocessor Programming-Elsevier Inc. (2021)

*Chapter 9 Linked lists: The role of locking*

**Задания**

***Задание 1.*** Определить проблему.

Проблема в том, что класс, который использует одну блокировку для передачи всех вызовов методов, не всегда масштабируем, даже если сама блокировка масштабируема. Грубая синхронизация хорошо работает, когда уровни параллелизма низки, но если слишком много потоков пытаются получить доступ к объекту одновременно, то объект становится последовательным препятствием, заставляя потоки ждать в очереди для доступа.

**Ответ:**

***Задание 2.*** Описать суть каждого алгоритма.

**Ответ:**

* Грубая (Coarse-grained) синхронизация

Грубая синхронизация относится к методу программирования, при котором доступ к общим ресурсам контролируется на высоком уровне, обычно путем блокировки всего ресурса. Этот подход может быть менее эффективным, чем детальная синхронизация, которая контролирует доступ к определенным частям общего ресурса, но его проще реализовать и обдумать.

* Тонкая (Fine-grained) синхронизация

Вместо использования одной блокировки для синхронизации каждого доступа к объекту мы разделяем объект на независимо синхронизируемые компоненты, позволяя выполнять вызовы методов, которые обращаются к непересекающимся компонентам одновременно.

* Оптимистичная (Optimistic) синхронизация

Многие объекты, такие как деревья или списки, состоят из нескольких компонентов, связанных между собой ссылками. Некоторые методы ищут конкретный компонент (например, список или узел дерева, содержащий определенный ключ). Один из способов снизить стоимость детальной блокировки — выполнять поиск вообще без получения каких-либо блокировок. Если метод находит искомый компонент, блокирует этот компонент, а затем проверяет, не изменился ли компонент за интервал между моментом его проверки и моментом его блокировки. Этот метод имеет смысл только в том случае, если он в большем количестве случаев успешен, чем не успешен, поэтому мы называем его оптимистичным.

* Ленивая (Lazy) синхронизация

Иногда имеет смысл отложить тяжелую работу. Например, задачу удаления компонента из структуры данных можно разделить на два этапа: компонент удаляется логически, а позже компонент можно удалить физически, отсоединив его от остальной структуры данных.

* Неблокирующая (Nonblocking) синхронизация

Иногда мы можем полностью устранить блокировки, полагаясь на встроенные атомарные операции, такие как CompareAndSet() для синхронизации.

***Задание 3.*** Описать эксперимент для демонстрации работы алгоритмов.

**Подсказка:**

* много элементов;
* много потоков;
* в типичных приложениях, использующих наборы, вызовов contains() значительно больше, чем вызовов add() или remove()

**Ответ (скелет программы):**

1. Coarse-grained:

* Создание экземпляра CoarseList и заполнение его большим количеством элементов.
* Создание нескольких потоков, которые будут выполнять операции чтения (contains()), добавления (add()) и удаления (remove()) элементов из списка.
* Запуск потоков и их параллельное взаимодействие с использованием списка.
* Измерение производительности и проверка корректности списка после выполнения операций.
* import lists.CoarseList;  
    
  import java.util.Random;  
    
  public class CoarseListExperiment {  
   private static final int *NUM\_THREADS* = 8;  
   private static final int *NUM\_ELEMENTS* = 10000;  
    
   public static void main(String[] args) {  
   CoarseList<Integer> list = new CoarseList<>();  
    
   for (int i = 0; i < *NUM\_ELEMENTS*; i++) {  
   list.add(i);  
   }  
    
   Thread[] threads = new Thread[*NUM\_THREADS*];  
   for (int i = 0; i < *NUM\_THREADS*; i++) {  
   threads[i] = new Thread(new ListOperationTask(list));  
   threads[i].start();  
   }  
    
   for (int i = 0; i < *NUM\_THREADS*; i++) {  
   try {  
   threads[i].join();  
   } catch (InterruptedException e) {  
   e.printStackTrace();  
   }  
   }  
    
   for (int i = 0; i < *NUM\_ELEMENTS*; i++) {  
   if (!list.contains(i)) {  
   System.*out*.println("List does not contain element: " + i);  
   }  
   }  
    
   System.*out*.println("Experiment completed.");  
   }  
    
   private static class ListOperationTask implements Runnable {  
   private CoarseList<Integer> list;  
   private Random random;  
    
   public ListOperationTask(CoarseList<Integer> list) {  
   this.list = list;  
   this.random = new Random();  
   }  
    
   @Override  
   public void run() {  
   for (int i = 0; i < *NUM\_ELEMENTS*; i++) {  
   int element = random.nextInt(*NUM\_ELEMENTS*);  
   if (i % 3 == 0) {  
   list.contains(element);  
   } else if (i % 3 == 1) {  
   list.add(element);  
   } else {  
   list.remove(element);  
   }  
   }  
   }  
   }  
  }

1. **Fine-grained**
2. import java.util.Random;  
     
   public class FineListExperiment {  
    private static final int *NUM\_THREADS* = 10;  
    private static final int *NUM\_OPERATIONS* = 10000;  
     
    public static void main(String[] args) {  
    FineList<Integer> fineList = new FineList<>();  
     
    for (int i = 0; i < *NUM\_OPERATIONS*; i++) {  
    fineList.add(i);  
    }  
     
    Thread[] threads = new Thread[*NUM\_THREADS*];  
    for (int i = 0; i < *NUM\_THREADS*; i++) {  
    threads[i] = new Thread(new Worker(fineList, *NUM\_OPERATIONS*));  
    threads[i].start();  
    }  
     
    for (Thread thread : threads) {  
    try {  
    thread.join();  
    } catch (InterruptedException e) {  
    e.printStackTrace();  
    }  
    }  
     
    int elementToCheck = new Random().nextInt(*NUM\_OPERATIONS*);  
    boolean containsElement = fineList.contains(elementToCheck);  
    System.*out*.println("Element " + elementToCheck + " is present in the list: " + containsElement);  
    }  
     
    static class Worker implements Runnable {  
    private final FineList<Integer> fineList;  
    private final int numOperations;  
     
    public Worker(FineList<Integer> fineList, int numOperations) {  
    this.fineList = fineList;  
    this.numOperations = numOperations;  
    }  
     
    @Override  
    public void run() {  
    Random random = new Random();  
     
    long startTime = System.*nanoTime*();  
     
    for (int i = 0; i < numOperations; i++) {  
    int elementToCheck = random.nextInt(numOperations);  
    boolean containsElement = fineList.contains(elementToCheck);  
     
    int elementToAdd = random.nextInt(numOperations + *NUM\_THREADS*);  
    boolean added = fineList.add(elementToAdd);  
     
    int elementToRemove = random.nextInt(numOperations);  
    boolean removed = fineList.remove(elementToRemove);  
    }  
     
    long endTime = System.*nanoTime*();  
    long elapsedTime = endTime - startTime;  
    double elapsedTimeInSeconds = (double) elapsedTime / 1\_000\_000\_000;  
     
    System.*out*.println("Thread " + Thread.*currentThread*().getId() + " finished. Elapsed time: " + elapsedTimeInSeconds + " seconds");  
    }  
    }  
   }

3.Optimistic:

import java.util.Random;  
  
public class OptimisticListExperiment {  
 private static final int *NUM\_THREADS* = 8;  
 private static final int *NUM\_ELEMENTS* = 10000;  
  
 public static void main(String[] args) {  
 OptimisticList<Integer> list = new OptimisticList<>();  
  
 for (int i = 0; i < *NUM\_ELEMENTS*; i++) {  
 list.add(i);  
 }  
  
 Thread[] threads = new Thread[*NUM\_THREADS*];  
 for (int i = 0; i < *NUM\_THREADS*; i++) {  
 threads[i] = new Thread(new ListOperationTask(list));  
 threads[i].start();  
 }  
  
 for (int i = 0; i < *NUM\_THREADS*; i++) {  
 try {  
 threads[i].join();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
  
 System.*out*.println("Experiment completed.");  
 }  
  
 private static class ListOperationTask implements Runnable {  
 private OptimisticList<Integer> list;  
 private Random random;  
  
 public ListOperationTask(OptimisticList<Integer> list) {  
 this.list = list;  
 this.random = new Random();  
 }  
  
 @Override  
 public void run() {  
 long startTime = System.*nanoTime*();  
  
 for (int i = 0; i < *NUM\_ELEMENTS*; i++) {  
 int element = random.nextInt(*NUM\_ELEMENTS*);  
 list.contains(element);  
 }  
  
 long endTime = System.*nanoTime*();  
 long elapsedTime = endTime - startTime;  
 double elapsedTimeInSeconds = (double) elapsedTime / 1\_000\_000\_000;  
  
 System.*out*.println("Thread " + Thread.*currentThread*().getId() + " finished. Elapsed time: " + elapsedTimeInSeconds + " seconds");  
 }  
 }  
}

1. Lazy:
2. import java.util.Random;  
     
   public class LazyListExperiment {  
    private static final int *NUM\_THREADS* = 8;  
    private static final int *NUM\_ELEMENTS* = 10000;  
     
    public static void main(String[] args) {  
    LazyList<Integer> list = new LazyList<>();  
     
    for (int i = 0; i < *NUM\_ELEMENTS*; i++) {  
    list.add(i);  
    }  
     
    Thread[] threads = new Thread[*NUM\_THREADS*];  
    for (int i = 0; i < *NUM\_THREADS*; i++) {  
    threads[i] = new Thread(new ListOperationTask(list));  
    threads[i].start();  
    }  
     
    for (int i = 0; i < *NUM\_THREADS*; i++) {  
    try {  
    threads[i].join();  
    } catch (InterruptedException e) {  
    e.printStackTrace();  
    }  
    }  
     
    System.*out*.println("Experiment completed.");  
    }  
     
    private static class ListOperationTask implements Runnable {  
    private LazyList<Integer> list;  
    private Random random;  
     
    public ListOperationTask(LazyList<Integer> list) {  
    this.list = list;  
    this.random = new Random();  
    }  
     
    @Override  
    public void run() {  
    long startTime = System.*nanoTime*();  
     
    for (int i = 0; i < *NUM\_ELEMENTS*; i++) {  
    int element = random.nextInt(*NUM\_ELEMENTS*);  
    list.contains(element);  
    }  
     
    long endTime = System.*nanoTime*();  
    long elapsedTime = endTime - startTime;  
    double elapsedTimeInSeconds = (double) elapsedTime / 1\_000\_000\_000;  
     
    System.*out*.println("Thread " + Thread.*currentThread*().getId() + " finished. Elapsed time: " + elapsedTimeInSeconds + " seconds");  
    }  
    }  
   }
3. Nonblocking:
4. import java.util.Random;  
     
   public class LockFreeListExperiment {  
    private static final int *NUM\_THREADS* = 8;  
    private static final int *NUM\_ELEMENTS* = 10000;  
     
    public static void main(String[] args) {  
    LockFreeList<Integer> list = new LockFreeList<>();  
     
    for (int i = 0; i < *NUM\_ELEMENTS*; i++) {  
    list.add(i);  
    }  
     
    Thread[] threads = new Thread[*NUM\_THREADS*];  
    for (int i = 0; i < *NUM\_THREADS*; i++) {  
    threads[i] = new Thread(new ListOperationTask(list));  
    threads[i].start();  
    }  
     
    for (int i = 0; i < *NUM\_THREADS*; i++) {  
    try {  
    threads[i].join();  
    } catch (InterruptedException e) {  
    e.printStackTrace();  
    }  
    }  
     
    System.*out*.println("Experiment completed.");  
    }  
     
    private static class ListOperationTask implements Runnable {  
    private LockFreeList<Integer> list;  
    private Random random;  
     
    public ListOperationTask(LockFreeList<Integer> list) {  
    this.list = list;  
    this.random = new Random();  
    }  
     
    @Override  
    public void run() {  
    long startTime = System.*nanoTime*();  
     
    for (int i = 0; i < *NUM\_ELEMENTS*; i++) {  
    int element = random.nextInt(*NUM\_ELEMENTS*);  
    list.contains(element);  
    }  
     
    long endTime = System.*nanoTime*();  
    long elapsedTime = endTime - startTime;  
    double elapsedTimeInSeconds = (double) elapsedTime / 1\_000\_000\_000;  
     
    System.*out*.println("Thread " + Thread.*currentThread*().getId() + " finished. Elapsed time: " + elapsedTimeInSeconds + " seconds");  
    }  
    }  
   }

***Задание 4.*** Выполнить эксперименты для сравнения двух (любых) алгоритмов.

***Отчет:***

* Текст программы

public class SynchronizationTest {  
 private final int numThreads = 10000;  
 private final int numOperations = 100000;  
  
 public static void main(String[] args) {  
 try {  
 new SynchronizationTest().test();  
 } catch (InterruptedException e) {  
 System.*err*.println(e.getMessage());  
 }  
 }  
  
 private void test() throws InterruptedException {  
 System.*out*.println(  
 "Грубая синхронизация заняла " + testAlgorithm(new CoarseGrainedSet(), numThreads, numOperations) + " наносекунд"  
 );  
  
 System.*out*.println(  
 "Тонкая синхронизация заняла " + testAlgorithm(new FineGrainedSet(), numThreads, numOperations) + " наносекунд"  
 );  
 }  
  
 private long testAlgorithm(  
 SynchronizedSet set,  
 int numThreads,  
 int numOperations  
 ) throws InterruptedException {  
 Thread[] threads = new Thread[numThreads];  
 long startTime = System.*nanoTime*();  
  
 final Runnable test = () -> {  
 for (int j = 0; j < numOperations; j++) {  
 set.add(j);  
 set.contains(j);  
 set.remove(j);  
 }  
 };  
  
 for (int i = 0; i < numThreads; i++) {  
 threads[i] = new Thread(test);  
 threads[i].start();  
 }  
 for (Thread thread : threads) {  
 thread.join();  
 }  
 long endTime = System.*nanoTime*();  
  
 return endTime - startTime;  
 }  
}

public class CoarseGrainedSet implements SynchronizedSet {  
 private final Set<Integer> set = new HashSet<>();  
 private final Object lock = new Object();  
  
 public void add(int value) {  
 synchronized(lock) {  
 set.add(value);  
 }  
 }  
  
 public boolean contains(int value) {  
 synchronized(lock) {  
 return set.contains(value);  
 }  
 }  
  
 public void remove(int value) {  
 synchronized(lock) {  
 set.remove(value);  
 }  
 }  
}

public class FineGrainedSet implements SynchronizedSet {  
 private final ConcurrentHashMap<Integer, ReentrantLock> map =  
 new ConcurrentHashMap<>();  
  
 @Override  
 public void add(int value) {  
 ReentrantLock lock = new ReentrantLock();  
 lock.lock();  
 try {  
 map.put(value, lock);  
 } finally {  
 lock.unlock();  
 }  
 }  
  
 @Override  
 public boolean contains(int value) {  
 ReentrantLock lock = map.get(value);  
 if (lock != null) {  
 lock.lock();  
 try {  
 return map.containsKey(value);  
 } finally {  
 lock.unlock();  
 }  
 }  
 return false;  
 }  
  
 @Override  
 public void remove(int value) {  
 ReentrantLock lock = map.get(value);  
 if (lock != null) {  
 lock.lock();  
 try {  
 map.remove(value);  
 } finally {  
 lock.unlock();  
 }  
 }  
 }  
}

* Результаты экспериментов

Грубая синхронизация:

|  |  |  |
| --- | --- | --- |
| Кол-во потоков | Кол-во операций | Время, nanos |
| 100 | 10000 | 263569500 |
| 10 | 10000 | 67286300 |
| 1000 | 1000 | 581063100 |

Тонкая синхронизация

|  |  |  |
| --- | --- | --- |
| Кол-во потоков | Кол-во операций | Время, nanos |
| 100 | 10000 | 161151700 |
| 10 | 10000 | 45562900 |
| 1000 | 1000 | 229049700 |

* Выводы

Тонкая синхронизация во всех случаях работает быстрее, чем грубая синхронизация. Это ожидаемо, поскольку тонкая синхронизация обычно позволяет большему количеству потоков работать параллельно.