**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра ВТ**

отчёт

**по лабораторной работе №1**

**по дисциплине  
«Параллельные алгоритмы и системы»**

Тема: «**Алгоритмы блокировки потоков**»

|  |  |  |
| --- | --- | --- |
| Студентка гр. 7305 | \_\_\_\_\_\_\_\_\_\_\_\_\_ | Степанова А.А. |
|  |  |  |
| Преподаватель | \_\_\_\_\_\_\_\_\_\_\_\_\_ | Пазников А.А. |
|  |  |  |

Санкт-Петербург

2021

**Цель**

Ознакомиться с алгоритмами блокировки потоков, выполнить анализ эффективности программы, построить графики пропускной способности от числа потоков.

**Задание**

1. Разработать программу, реализующую параллельный алгоритм. Средства разработки: С/С++, Java.
2. Выполнить анализ эффективности параллельной программы: построить графики зависимости пропускной способности от числа потоков, оценить масштабирование.
3. Сформулировать выводы

**Выполнение работы**

При выполнении лабораторной работы были реализованы алгоритмы блокировки потоков, такие как MCS и CLH. Для реализации алгоритмов был выбран язык Java. Был осуществлён анализ эффективности параллельной программы по полученным данным, построены графики*.*

1. *MSC*

Экспериментальные результаты:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Кол-во потоков | 2 | 4 | 8 | 16 | 32 | 64 |
| Кол-во блокировок | 200150 | 110735 | 108321 | 94120 | 76872 | 61854 |

1. *CLH*

Экспериментальные результаты:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Кол-во потоков | 2 | 4 | 8 | 16 | 32 | 64 |
| Кол-во блокировок | 280630 | 145791 | 120945 | 98916 | 89631 | 74721 |

**Вывод**

В результате полученных данных, можно сделать вывод, о том, что наибольшей пропускной способностью обладает алгоритм CLH. Также можно заметить, что при увеличении числа потоков пропускная способность алгоритмов уменьшается.

**Листинг**

CLH

Main,java

import java.util.concurrent.ExecutorService;  
import java.util.concurrent.Executors;  
import java.util.concurrent.atomic.AtomicBoolean;  
import java.util.concurrent.atomic.AtomicInteger;  
  
public class Main {  
 private final static int *threads* = 64;  
 private final static AtomicInteger *count* = new AtomicInteger(0);  
 private final static AtomicBoolean *stop* = new AtomicBoolean(false);  
 private final static CLHLock *lock* = new CLHLock();  
  
 static class MyRunnable implements Runnable {  
 @Override  
 public void run() {  
 while (!*stop*.get()) {  
 *lock*.lock();  
 count.incrementAndGet();  
 lock.unlock();  
 }  
 }  
 }  
  
 public static void main(String[] args) {  
 ExecutorService exec = Executors.*newFixedThreadPool*(*threads*);  
  
  
 try {  
 for (int i = 0; i < *threads*; i++) {  
 exec.execute(new MyRunnable());  
 }  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
  
 try {  
 Thread.*sleep*(1000);  
 } catch (InterruptedException e) {  
 // *TODO Auto-generated catch block* e.printStackTrace();  
 }  
  
 *stop*.set(true);  
 exec.shutdown();  
  
 System.*out*.println("locks: " + *count*);  
 }  
  
}

CLHLock.java

import java.util.concurrent.atomic.AtomicReference;  
  
  
public class CLHLock{  
 AtomicReference<CLHNode> tail = new AtomicReference<CLHNode>(new CLHNode());  
 ThreadLocal<CLHNode> myPred;  
 ThreadLocal<CLHNode> myNode;  
  
 public static class CLHNode {  
 public volatile boolean locked = false;  
 }  
  
 public CLHLock() {  
 myNode = new ThreadLocal<CLHNode>() {  
 protected CLHNode initialValue() {  
 return new CLHNode();  
 }  
 };  
 myPred = new ThreadLocal<CLHNode>() {  
 protected CLHNode initialValue() {  
 return null;  
 }  
 };  
 }  
  
  
 public void lock() {  
 CLHNode qnode = myNode.get();  
 qnode.locked = true;  
 CLHNode pred = tail.getAndSet(qnode);  
 myPred.set(pred);  
 while (pred.locked) { }  
 }  
  
 public void unlock() {  
 CLHNode qnode = myNode.get();  
 qnode.locked = false;  
 myNode.set(myPred.get());  
 }  
  
}

MCS

Main.java

import java.util.concurrent.ExecutorService;  
import java.util.concurrent.Executors;  
import java.util.concurrent.atomic.AtomicBoolean;  
import java.util.concurrent.atomic.AtomicInteger;  
  
public class Main {  
 private final static int threads = 64;  
 private final static AtomicInteger count = new AtomicInteger(0);  
 private final static AtomicBoolean stop = new AtomicBoolean(false);  
 private final static MCSLock lock = new MCSLock();  
  
 static class MyRunnable implements Runnable {  
 @Override  
 public void run() {  
 while (!stop.get()) {  
 lock.lock();  
 count.incrementAndGet();  
 lock.unlock();  
 }  
 }  
 }  
  
 public static void main(String[] args) {  
 ExecutorService exec = Executors.*newFixedThreadPool*(*threads*);  
  
  
 try {  
 for (int i = 0; i < *threads*; i++) {  
 exec.execute(new MyRunnable());  
 }  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
  
 try {  
 Thread.*sleep*(1000);  
 } catch (InterruptedException e) {  
 // TODO Auto-generated catch block  
 e.printStackTrace();  
 }  
  
 stop.set(true);  
 exec.shutdown();  
  
 System.out.println("locks: " + count);  
 }  
  
}

MCSLock.java

import java.util.concurrent.atomic.AtomicReference;  
  
public class MCSLock{  
 private volatile AtomicReference<MCSNode> tail;  
 private volatile ThreadLocal<MCSNode> threadLocal;  
  
 public MCSLock() {  
 this.tail = new AtomicReference<>();  
 this.threadLocal = new ThreadLocal<>();  
 }  
  
  
 public void lock() {  
 MCSNode qNode = threadLocal.get();  
 if (qNode == null) {  
 qNode = new MCSNode();  
 threadLocal.set(qNode);  
 }  
  
 MCSNode pred = tail.getAndSet(qNode);  
 if (pred != null) {  
 qNode.setLocked(true);  
 pred.setNext(qNode);  
 while (qNode.getLocked()) { }  
 } else {  
 qNode.setLocked(false);  
 }  
 }  
  
  
 public void unlock() {  
 MCSNode qNode = threadLocal.get();  
 threadLocal.remove();  
  
 if (qNode == null || qNode.getLocked() == true) {  
 return;  
 }  
  
 if (qNode.getNext() == null) {  
 if(tail.compareAndSet(qNode, null)) {  
 return ;  
 }  
 while (qNode.getNext() == null) { }  
 }  
  
 if (qNode.getNext() != null) {  
 qNode.getNext().setLocked(false);  
 qNode.setNext(null);  
 }  
 }  
}

MCSNode.java

import java.util.concurrent.atomic.AtomicBoolean;  
  
public class MCSNode {  
 private MCSNode next;  
 private AtomicBoolean locked = new AtomicBoolean(true);  
  
 public boolean getLocked() {  
 return this.locked.get();  
 }  
  
 public void setLocked(boolean locked) {  
 this.locked.set(locked);  
 }  
  
 public MCSNode getNext() {  
 return next;  
 }  
  
 public void setNext(MCSNode next) {  
 this.next = next;  
 }  
}