**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра ВТ**

отчет

**по лабораторной работе №1**

**по дисциплине «GRID-технологии и облачные вычисления»**

Тема: Lock-free множество

|  |  |  |
| --- | --- | --- |
| Студент гр. 5307 |  | Пешков М.А. |
| Преподаватель |  | Пазников А.А. |

Санкт-Петербург

2020

**Цель работы.**

Изучить и реализовать Lock-free множество на Java.

**Аппаратное окружение.**

* CPU: i5-4210U
* RAM: 8GB DDR3

**Экспериментальные результаты.**

Были проведены тесты производительности с использованием jmh. На графиках ниже представлена разница в пропускной способности и времени выполнения между lock free множеством и synchronized множеством.

**Выводы.**

В результате выполнения данной лабораторной работы был освоен и реализован алгоритм построения lock-free множества на языке Java. Были проведены эксперименты с измерением пропускной способности и ускорения реализованного множества.

Приложение а

Код lock free set

package com.github.peshkovm.lab1;  
  
import java.util.concurrent.atomic.AtomicMarkableReference;  
  
public class LockFreeList implements Set<Integer> {  
 private final Node head;  
 private final Node tail;  
  
 public LockFreeList() {  
 this.tail = new Node(Integer.*MAX\_VALUE*, new AtomicMarkableReference<>(null, false));  
 this.head = new Node(Integer.*MIN\_VALUE*, new AtomicMarkableReference<>(tail, false));  
 }  
  
 @Override  
 public boolean add(Integer item) {  
 while (true) {  
 Window window = find(head, item);  
 Node pred = window.pred, curr = window.curr;  
 if (curr.item.equals(item)) {  
 return false;  
 } else {  
 Node node = new Node(item, new AtomicMarkableReference<>(curr, false));  
 if (pred.next.compareAndSet(curr, node, false, false)) {  
 return true;  
 }  
 }  
 }  
 }  
  
 @Override  
 public boolean remove(Integer item) {  
 boolean snip;  
 for (int i = 0; i < 100\_000; i++) {  
 Window window = find(head, item);  
 Node pred = window.pred, curr = window.curr;  
 if (!curr.item.equals(item)) {  
 return false;  
 } else {  
 Node succ = curr.next.getReference();  
 snip = curr.next.attemptMark(succ, true);  
 if (!snip) continue;  
 pred.next.compareAndSet(curr, succ, false, false);  
 return true;  
 }  
 }  
  
 throw new RuntimeException("Spinlock was spinning for too long");  
 }  
  
 @Override  
 public boolean contains(Integer item) {  
 boolean[] marked = {false};  
 Node curr = head;  
 while (curr.item < item) {  
 curr = curr.next.getReference();  
 Node succ = curr.next.get(marked);  
 }  
 return (curr.item.equals(item) && !marked[0]);  
 }  
  
 @Override  
 public synchronized void clear() {  
 Node curr = this.head.next.getReference();  
  
 while (curr != tail) {  
 remove(curr.item);  
 curr = curr.next.getReference();  
 }  
 }  
  
 @Override  
 public synchronized int size() {  
 Node curr = this.head.next.getReference();  
 int size = 0;  
  
 while (curr != tail) {  
 size++;  
 curr = curr.next.getReference();  
 }  
  
 return size;  
 }  
  
 private Window find(Node head, int item) {  
 Node pred = null, curr = null, succ = null;  
 boolean[] marked = {false};  
 boolean snip;  
 retry:  
 for (int i = 0; i < 100\_000; i++) {  
 pred = head;  
 curr = pred.next.getReference();  
 while (true) {  
 succ = curr.next.get(marked);  
 while (marked[0]) {  
 snip = pred.next.compareAndSet(curr, succ, false, false);  
 if (!snip) continue retry;  
 curr = succ;  
 succ = curr.next.get(marked);  
 }  
 if (curr.item >= item) return new Window(pred, curr);  
 pred = curr;  
 curr = succ;  
 }  
 }  
  
 throw new RuntimeException("Spinlock was spinning for too long");  
 }  
  
 private static class Window {  
 public Node pred, curr;  
  
 Window(Node myPred, Node myCurr) {  
 pred = myPred;  
 curr = myCurr;  
 }  
 }  
  
 private static class Node {  
 final Integer item;  
 AtomicMarkableReference<Node> next;  
  
 Node(Integer item, AtomicMarkableReference<Node> next) {  
 this.item = item;  
 this.next = next;  
 }  
 }  
}

Код performance test

package com.github.peshkovm.lab1;  
  
import java.io.IOException;  
import java.nio.file.Files;  
import java.nio.file.Paths;  
import java.nio.file.StandardOpenOption;  
import java.util.Collection;  
import java.util.concurrent.TimeUnit;  
import java.util.concurrent.atomic.AtomicInteger;  
import jdk.internal.vm.annotation.Contended;  
import org.openjdk.jmh.annotations.Benchmark;  
import org.openjdk.jmh.annotations.BenchmarkMode;  
import org.openjdk.jmh.annotations.Fork;  
import org.openjdk.jmh.annotations.Level;  
import org.openjdk.jmh.annotations.Measurement;  
import org.openjdk.jmh.annotations.Mode;  
import org.openjdk.jmh.annotations.OutputTimeUnit;  
import org.openjdk.jmh.annotations.Scope;  
import org.openjdk.jmh.annotations.Setup;  
import org.openjdk.jmh.annotations.State;  
import org.openjdk.jmh.annotations.TearDown;  
import org.openjdk.jmh.annotations.Threads;  
import org.openjdk.jmh.annotations.Warmup;  
import org.openjdk.jmh.infra.BenchmarkParams;  
import org.openjdk.jmh.infra.Blackhole;  
import org.openjdk.jmh.results.RunResult;  
import org.openjdk.jmh.runner.Runner;  
import org.openjdk.jmh.runner.options.Options;  
import org.openjdk.jmh.runner.options.OptionsBuilder;  
  
public class PerformanceTest {  
 private static final int *SET\_SIZE* = 50\_000;  
 private static final int *NUM\_ITERATIONS* = 10;  
 private static final int *ITERATION\_TIME* = 1000;  
 private static final int *NUM\_OF\_FORKS* = 2;  
 private static final String *RES\_FILE\_PATH* = "max/src/test/resources/res.csv";  
  
 @State(Scope.*Benchmark*)  
 public abstract static class SetState {  
 @Contended protected Set<Integer> set;  
 @Contended protected AtomicInteger i;  
  
 @Setup(Level.*Iteration*)  
 public void prepareSet0() {  
 prepareSet();  
 }  
  
 public abstract void init();  
  
 public abstract void prepareSet();  
 }  
  
 public abstract static class SetAddState extends SetState {  
  
 @Override  
 public void prepareSet() {  
 set.clear();  
 i.set(0);  
 }  
  
 @Setup(Level.*Trial*)  
 public void init0() {  
 init();  
 }  
 }  
  
 public static class LockFreeListAddState extends SetAddState {  
 @Override  
 public void init() {  
 set = new LockFreeList();  
 i = new AtomicInteger(0);  
 }  
  
 @TearDown(Level.*Iteration*)  
 public void check(BenchmarkParams params) {  
 if (set.size() != i.get()) {  
 throw new RuntimeException("lockFreeList.size()!=listSize");  
 }  
 }  
 }  
  
 public static class SynchronizedListAddState extends SetAddState {  
 @Override  
 public void init() {  
 set = new SynchronizedList();  
 i = new AtomicInteger(0);  
 }  
  
 @TearDown(Level.*Iteration*)  
 public void check(BenchmarkParams params) {  
 if (set.size() != i.get()) {  
 throw new RuntimeException("synchronizedList.size()!=listSize");  
 }  
 }  
 }  
  
 @Fork(*NUM\_OF\_FORKS*)  
 @Warmup(iterations = *NUM\_ITERATIONS*, time = *ITERATION\_TIME*, timeUnit = TimeUnit.*MILLISECONDS*)  
 @Measurement(iterations = *NUM\_ITERATIONS*, time = *ITERATION\_TIME*, timeUnit = TimeUnit.*MILLISECONDS*)  
 @BenchmarkMode(Mode.*SingleShotTime*)  
 @OutputTimeUnit(TimeUnit.*MILLISECONDS*)  
 public static class LockFreeListAccelerationTest {  
 @Threads(1)  
 @Warmup(batchSize = *SET\_SIZE*)  
 @Measurement(batchSize = *SET\_SIZE*)  
 @Benchmark  
 public void add\_1\_thread(final LockFreeListAddState state, final Blackhole bh) {  
 final Set<Integer> lockFreeList = state.set;  
 final AtomicInteger i = state.i;  
  
 bh.consume(lockFreeList.add(i.getAndIncrement()));  
 }  
  
 @Threads(2)  
 @Warmup(batchSize = *SET\_SIZE* / 2)  
 @Measurement(batchSize = *SET\_SIZE* / 2)  
 @Benchmark  
 public void add\_2\_thread(final LockFreeListAddState state, final Blackhole bh) {  
 final Set<Integer> lockFreeList = state.set;  
 final AtomicInteger i = state.i;  
  
 bh.consume(lockFreeList.add(i.getAndIncrement()));  
 }  
  
 @Threads(3)  
 @Warmup(batchSize = *SET\_SIZE* / 3)  
 @Measurement(batchSize = *SET\_SIZE* / 3)  
 @Benchmark  
 public void add\_3\_thread(final LockFreeListAddState state, final Blackhole bh) {  
 final Set<Integer> lockFreeList = state.set;  
 final AtomicInteger i = state.i;  
  
 bh.consume(lockFreeList.add(i.getAndIncrement()));  
 }  
  
 @Threads(4)  
 @Warmup(batchSize = *SET\_SIZE* / 4)  
 @Measurement(batchSize = *SET\_SIZE* / 4)  
 @Benchmark  
 public void add\_4\_thread(final LockFreeListAddState state, final Blackhole bh) {  
 final Set<Integer> lockFreeList = state.set;  
 final AtomicInteger i = state.i;  
  
 bh.consume(lockFreeList.add(i.getAndIncrement()));  
 }  
  
 @Threads(10)  
 @Warmup(batchSize = *SET\_SIZE* / 10)  
 @Measurement(batchSize = *SET\_SIZE* / 10)  
 @Benchmark  
 public void add\_10\_thread(final LockFreeListAddState state, final Blackhole bh) {  
 final Set<Integer> lockFreeList = state.set;  
 final AtomicInteger i = state.i;  
  
 bh.consume(lockFreeList.add(i.getAndIncrement()));  
 }  
 }  
  
 @Fork(*NUM\_OF\_FORKS*)  
 @Warmup(iterations = *NUM\_ITERATIONS*, time = *ITERATION\_TIME*, timeUnit = TimeUnit.*MILLISECONDS*)  
 @Measurement(iterations = *NUM\_ITERATIONS*, time = *ITERATION\_TIME*, timeUnit = TimeUnit.*MILLISECONDS*)  
 @BenchmarkMode(Mode.*Throughput*)  
 public static class LockFreeListThroughputTest {  
 @Threads(1)  
 @Benchmark  
 public void add\_1\_thread(final LockFreeListAddState state, final Blackhole bh) {  
 final Set<Integer> lockFreeList = state.set;  
 final AtomicInteger i = state.i;  
  
 bh.consume(lockFreeList.add(i.getAndIncrement()));  
 }  
  
 @Threads(2)  
 @Benchmark  
 public void add\_2\_thread(final LockFreeListAddState state, final Blackhole bh) {  
 final Set<Integer> lockFreeList = state.set;  
 final AtomicInteger i = state.i;  
  
 bh.consume(lockFreeList.add(i.getAndIncrement()));  
 }  
  
 @Threads(3)  
 @Benchmark  
 public void add\_3\_thread(final LockFreeListAddState state, final Blackhole bh) {  
 final Set<Integer> lockFreeList = state.set;  
 final AtomicInteger i = state.i;  
  
 bh.consume(lockFreeList.add(i.getAndIncrement()));  
 }  
  
 @Threads(4)  
 @Benchmark  
 public void add\_4\_thread(final LockFreeListAddState state, final Blackhole bh) {  
 final Set<Integer> lockFreeList = state.set;  
 final AtomicInteger i = state.i;  
  
 bh.consume(lockFreeList.add(i.getAndIncrement()));  
 }  
  
 @Threads(10)  
 @Benchmark  
 public void add\_10\_thread(final LockFreeListAddState state, final Blackhole bh) {  
 final Set<Integer> lockFreeList = state.set;  
 final AtomicInteger i = state.i;  
  
 bh.consume(lockFreeList.add(i.getAndIncrement()));  
 }  
 }  
  
 @Fork(*NUM\_OF\_FORKS*)  
 @Warmup(iterations = *NUM\_ITERATIONS*, time = *ITERATION\_TIME*, timeUnit = TimeUnit.*MILLISECONDS*)  
 @Measurement(iterations = *NUM\_ITERATIONS*, time = *ITERATION\_TIME*, timeUnit = TimeUnit.*MILLISECONDS*)  
 @BenchmarkMode(Mode.*SingleShotTime*)  
 @OutputTimeUnit(TimeUnit.*MILLISECONDS*)  
 public static class SynchronizedListAccelerationTest {  
 @Threads(1)  
 @Warmup(batchSize = *SET\_SIZE*)  
 @Measurement(batchSize = *SET\_SIZE*)  
 @Benchmark  
 public void add\_1\_thread(final SynchronizedListAddState state, final Blackhole bh) {  
 final Set<Integer> synchronizedList = state.set;  
 final AtomicInteger i = state.i;  
  
 bh.consume(synchronizedList.add(i.getAndIncrement()));  
 }  
  
 @Threads(2)  
 @Warmup(batchSize = *SET\_SIZE* / 2)  
 @Measurement(batchSize = *SET\_SIZE* / 2)  
 @Benchmark  
 public void add\_2\_thread(final SynchronizedListAddState state, final Blackhole bh) {  
 final Set<Integer> synchronizedList = state.set;  
 final AtomicInteger i = state.i;  
  
 bh.consume(synchronizedList.add(i.getAndIncrement()));  
 }  
  
 @Threads(3)  
 @Warmup(batchSize = *SET\_SIZE* / 3)  
 @Measurement(batchSize = *SET\_SIZE* / 3)  
 @Benchmark  
 public void add\_3\_thread(final SynchronizedListAddState state, final Blackhole bh) {  
 final Set<Integer> synchronizedList = state.set;  
 final AtomicInteger i = state.i;  
  
 bh.consume(synchronizedList.add(i.getAndIncrement()));  
 }  
  
 @Threads(4)  
 @Warmup(batchSize = *SET\_SIZE* / 4)  
 @Measurement(batchSize = *SET\_SIZE* / 4)  
 @Benchmark  
 public void add\_4\_thread(final SynchronizedListAddState state, final Blackhole bh) {  
 final Set<Integer> synchronizedList = state.set;  
 final AtomicInteger i = state.i;  
  
 bh.consume(synchronizedList.add(i.getAndIncrement()));  
 }  
  
 @Threads(10)  
 @Warmup(batchSize = *SET\_SIZE* / 10)  
 @Measurement(batchSize = *SET\_SIZE* / 10)  
 @Benchmark  
 public void add\_10\_thread(final SynchronizedListAddState state, final Blackhole bh) {  
 final Set<Integer> synchronizedList = state.set;  
 final AtomicInteger i = state.i;  
  
 bh.consume(synchronizedList.add(i.getAndIncrement()));  
 }  
 }  
  
 @Fork(*NUM\_OF\_FORKS*)  
 @Warmup(iterations = *NUM\_ITERATIONS*, time = *ITERATION\_TIME*, timeUnit = TimeUnit.*MILLISECONDS*)  
 @Measurement(iterations = *NUM\_ITERATIONS*, time = *ITERATION\_TIME*, timeUnit = TimeUnit.*MILLISECONDS*)  
 @BenchmarkMode(Mode.*Throughput*)  
 public static class SynchronizedListThroughputTest {  
 @Threads(1)  
 @Benchmark  
 public void add\_1\_thread(final SynchronizedListAddState state, final Blackhole bh) {  
 final Set<Integer> synchronizedList = state.set;  
 final AtomicInteger i = state.i;  
  
 bh.consume(synchronizedList.add(i.getAndIncrement()));  
 }  
  
 @Threads(2)  
 @Benchmark  
 public void add\_2\_thread(final SynchronizedListAddState state, final Blackhole bh) {  
 final Set<Integer> synchronizedList = state.set;  
 final AtomicInteger i = state.i;  
  
 bh.consume(synchronizedList.add(i.getAndIncrement()));  
 }  
  
 @Threads(3)  
 @Benchmark  
 public void add\_3\_thread(final SynchronizedListAddState state, final Blackhole bh) {  
 final Set<Integer> synchronizedList = state.set;  
 final AtomicInteger i = state.i;  
  
 bh.consume(synchronizedList.add(i.getAndIncrement()));  
 }  
  
 @Threads(4)  
 @Benchmark  
 public void add\_4\_thread(final SynchronizedListAddState state, final Blackhole bh) {  
 final Set<Integer> synchronizedList = state.set;  
 final AtomicInteger i = state.i;  
  
 bh.consume(synchronizedList.add(i.getAndIncrement()));  
 }  
  
 @Threads(10)  
 @Benchmark  
 public void add\_10\_thread(final SynchronizedListAddState state, final Blackhole bh) {  
 final Set<Integer> synchronizedList = state.set;  
 final AtomicInteger i = state.i;  
  
 bh.consume(synchronizedList.add(i.getAndIncrement()));  
 }  
 }  
  
 public static void main(String[] args) throws Exception {  
 Options opt =  
 new OptionsBuilder()  
 .include(PerformanceTest.class.getName())  
 .jvmArgsAppend("-XX:-RestrictContended")  
 .syncIterations(true)  
 .build();  
  
 final Collection<RunResult> runResults = new Runner(opt).run();  
  
 Files.*deleteIfExists*(Paths.*get*(*RES\_FILE\_PATH*));  
 Files.*createFile*(Paths.*get*(*RES\_FILE\_PATH*));  
 Files.*write*(  
 Paths.*get*(*RES\_FILE\_PATH*),  
 ("Id,"  
 + "Mode,"  
 + "Cnt,"  
 + "Threads,"  
 + "Score,"  
 + "Error,"  
 + "Units"  
 + System.*lineSeparator*())  
 .getBytes(),  
 StandardOpenOption.*APPEND*);  
  
 runResults.forEach(  
 runResult -> {  
 final String id = runResult.getParams().id();  
 final Mode mode = runResult.getParams().getMode();  
 final long sampleCount = runResult.getPrimaryResult().getSampleCount();  
 final int threads = runResult.getParams().getThreads();  
 final double score = runResult.getPrimaryResult().getScore();  
 final double scoreError = runResult.getPrimaryResult().getScoreError();  
 final String scoreUnit = runResult.getPrimaryResult().getScoreUnit();  
  
 try {  
 Files.*write*(  
 Paths.*get*(*RES\_FILE\_PATH*),  
 (id  
 + ","  
 + mode  
 + ","  
 + sampleCount  
 + ","  
 + threads  
 + ","  
 + score  
 + ","  
 + scoreError  
 + ","  
 + scoreUnit  
 + System.*lineSeparator*())  
 .getBytes(),  
 StandardOpenOption.*APPEND*);  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 });  
 }  
}