**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 L.А. |
| Преподаватель |  | Пазников А.А. |

Санкт-Петербург

2020

**Цель работы.**

Изучить и реализовать Lock-free очередь 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 очередью и synchronized очередью.

**Выводы.**

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

Приложение а

Код lock free queue

package com.github.denpeshkov;  
  
import java.util.concurrent.atomic.AtomicReference;  
  
public class LockFreeQueue<T> implements Pool<T> {  
 private final AtomicReference<Node<T>> head;  
 private final AtomicReference<Node<T>> tail;  
  
 public LockFreeQueue() {  
 Node<T> node = new Node<>();  
 this.head = new AtomicReference<>(node);  
 this.tail = new AtomicReference<>(node);  
 }  
  
 public void enq(T item) {  
 Node<T> node = new Node<>();  
 node.item = item;  
  
 while (true) {  
 Node<T> last = tail.get();  
 Node<T> next = last.next.get();  
  
 if (last == tail.get()) {  
 if (next == null) {  
 if (last.next.compareAndSet(null, node)) {  
 tail.compareAndSet(last, node);  
 return;  
 }  
 } else {  
 tail.compareAndSet(last, next);  
 }  
 }  
 }  
 }  
  
 public T deq() {  
 while (true) {  
 Node<T> first = head.get(), last = tail.get();  
 Node<T> next = first.next.get();  
  
 if (first == head.get()) {  
 if (first == last) {  
 if (next == null) {  
 throw new IllegalArgumentException("Queue is empty");  
 }  
 tail.compareAndSet(last, next);  
 } else if (head.compareAndSet(first, next)) {  
 return next.item;  
 }  
 }  
 }  
 }  
  
 @Override  
 public synchronized void clear() {  
 final int size = size();  
 for (int i = 0; i < size; i++) {  
 deq();  
 }  
 }  
  
 @Override  
 public synchronized int size() {  
 Node<T> curr = head.get();  
 int size = 0;  
  
 while (curr != null) {  
 size++;  
 curr = curr.next.get();  
 }  
  
 return size - 1;  
 }  
  
 private static class Node<V> {  
 V item;  
 AtomicReference<Node<V>> next = new AtomicReference<>(null);  
 }  
}

Код performance test

package com.github.denpeshkov;  
  
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* = "den/src/test/resources/res.csv";  
  
 @State(Scope.*Benchmark*)  
 public abstract static class SetState {  
 @Contended protected Pool<Integer> pool;  
 @Contended protected AtomicInteger i;  
  
 @Setup(Level.*Trial*)  
 public void init0() {  
 init();  
 }  
  
 @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() {  
 pool.clear();  
 i.set(0);  
 }  
 }  
  
 public static class LockFreeQueueAddState extends SetAddState {  
 @Override  
 public void init() {  
 pool = new LockFreeQueue<>();  
 i = new AtomicInteger(0);  
 }  
  
 @TearDown(Level.Iteration)  
 public void check(BenchmarkParams params) {  
 if (pool.size() != i.get()) {  
 throw new RuntimeException("lockFreeQueue.size()!=listSize");  
 }  
 }  
 }  
  
 public static class SynchronizedQueueAddState extends SetAddState {  
 @Override  
 public void init() {  
 pool = new SynchronizedQueue<>();  
 i = new AtomicInteger(0);  
 }  
  
 @TearDown(Level.Iteration)  
 public void check(BenchmarkParams params) {  
 if (pool.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.MICROSECONDS)  
 public static class LockFreeQueueAccelerationTest {  
 @Threads(1)  
 @Warmup(batchSize = SET\_SIZE)  
 @Measurement(batchSize = SET\_SIZE)  
 @Benchmark  
 public Pool<Integer> enq\_1\_thread(final LockFreeQueueAddState state, final Blackhole bh) {  
 final Pool<Integer> lockFreeQueue = state.pool;  
 final AtomicInteger i = state.i;  
  
 lockFreeQueue.enq(i.getAndIncrement());  
 return lockFreeQueue;  
 }  
  
 @Threads(2)  
 @Warmup(batchSize = SET\_SIZE / 2)  
 @Measurement(batchSize = SET\_SIZE / 2)  
 @Benchmark  
 public Pool<Integer> enq\_2\_thread(final LockFreeQueueAddState state, final Blackhole bh) {  
 final Pool<Integer> lockFreeQueue = state.pool;  
 final AtomicInteger i = state.i;  
  
 lockFreeQueue.enq(i.getAndIncrement());  
 return lockFreeQueue;  
 }  
  
 @Threads(3)  
 @Warmup(batchSize = SET\_SIZE / 3)  
 @Measurement(batchSize = SET\_SIZE / 3)  
 @Benchmark  
 public Pool<Integer> enq\_3\_thread(final LockFreeQueueAddState state, final Blackhole bh) {  
 final Pool<Integer> lockFreeQueue = state.pool;  
 final AtomicInteger i = state.i;  
  
 lockFreeQueue.enq(i.getAndIncrement());  
 return lockFreeQueue;  
 }  
  
 @Threads(4)  
 @Warmup(batchSize = SET\_SIZE / 4)  
 @Measurement(batchSize = SET\_SIZE / 4)  
 @Benchmark  
 public Pool<Integer> enq\_4\_thread(final LockFreeQueueAddState state, final Blackhole bh) {  
 final Pool<Integer> lockFreeQueue = state.pool;  
 final AtomicInteger i = state.i;  
  
 lockFreeQueue.enq(i.getAndIncrement());  
 return lockFreeQueue;  
 }  
  
 @Threads(10)  
 @Warmup(batchSize = SET\_SIZE / 10)  
 @Measurement(batchSize = SET\_SIZE / 10)  
 @Benchmark  
 public Pool<Integer> enq\_10\_thread(final LockFreeQueueAddState state, final Blackhole bh) {  
 final Pool<Integer> lockFreeQueue = state.pool;  
 final AtomicInteger i = state.i;  
  
 lockFreeQueue.enq(i.getAndIncrement());  
 return lockFreeQueue;  
 }  
 }  
  
 @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 LockFreeQueueThroughputTest {  
 @Threads(1)  
 @Benchmark  
 public Pool<Integer> enq\_1\_thread(final LockFreeQueueAddState state, final Blackhole bh) {  
 final Pool<Integer> lockFreeQueue = state.pool;  
 final AtomicInteger i = state.i;  
  
 lockFreeQueue.enq(i.getAndIncrement());  
 return lockFreeQueue;  
 }  
  
 @Threads(2)  
 @Benchmark  
 public Pool<Integer> enq\_2\_thread(final LockFreeQueueAddState state, final Blackhole bh) {  
 final Pool<Integer> lockFreeQueue = state.pool;  
 final AtomicInteger i = state.i;  
  
 lockFreeQueue.enq(i.getAndIncrement());  
 return lockFreeQueue;  
 }  
  
 @Threads(3)  
 @Benchmark  
 public Pool<Integer> enq\_3\_thread(final LockFreeQueueAddState state, final Blackhole bh) {  
 final Pool<Integer> lockFreeQueue = state.pool;  
 final AtomicInteger i = state.i;  
  
 lockFreeQueue.enq(i.getAndIncrement());  
 return lockFreeQueue;  
 }  
  
 @Threads(4)  
 @Benchmark  
 public Pool<Integer> enq\_4\_thread(final LockFreeQueueAddState state, final Blackhole bh) {  
 final Pool<Integer> lockFreeQueue = state.pool;  
 final AtomicInteger i = state.i;  
  
 lockFreeQueue.enq(i.getAndIncrement());  
 return lockFreeQueue;  
 }  
  
 @Threads(10)  
 @Benchmark  
 public Pool<Integer> enq\_10\_thread(final LockFreeQueueAddState state, final Blackhole bh) {  
 final Pool<Integer> lockFreeQueue = state.pool;  
 final AtomicInteger i = state.i;  
  
 lockFreeQueue.enq(i.getAndIncrement());  
 return lockFreeQueue;  
 }  
 }  
  
 @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.MICROSECONDS)  
 public static class SynchronizedQueueAccelerationTest {  
 @Threads(1)  
 @Warmup(batchSize = SET\_SIZE)  
 @Measurement(batchSize = SET\_SIZE)  
 @Benchmark  
 public Pool<Integer> enq\_1\_thread(final SynchronizedQueueAddState state, final Blackhole bh) {  
 final Pool<Integer> synchronizedQueue = state.pool;  
 final AtomicInteger i = state.i;  
  
 synchronizedQueue.enq(i.getAndIncrement());  
 return synchronizedQueue;  
 }  
  
 @Threads(2)  
 @Warmup(batchSize = SET\_SIZE / 2)  
 @Measurement(batchSize = SET\_SIZE / 2)  
 @Benchmark  
 public Pool<Integer> enq\_2\_thread(final SynchronizedQueueAddState state, final Blackhole bh) {  
 final Pool<Integer> synchronizedQueue = state.pool;  
 final AtomicInteger i = state.i;  
  
 synchronizedQueue.enq(i.getAndIncrement());  
 return synchronizedQueue;  
 }  
  
 @Threads(3)  
 @Warmup(batchSize = SET\_SIZE / 3)  
 @Measurement(batchSize = SET\_SIZE / 3)  
 @Benchmark  
 public Pool<Integer> enq\_3\_thread(final SynchronizedQueueAddState state, final Blackhole bh) {  
 final Pool<Integer> synchronizedQueue = state.pool;  
 final AtomicInteger i = state.i;  
  
 synchronizedQueue.enq(i.getAndIncrement());  
 return synchronizedQueue;  
 }  
  
 @Threads(4)  
 @Warmup(batchSize = SET\_SIZE / 4)  
 @Measurement(batchSize = SET\_SIZE / 4)  
 @Benchmark  
 public Pool<Integer> enq\_4\_thread(final SynchronizedQueueAddState state, final Blackhole bh) {  
 final Pool<Integer> synchronizedQueue = state.pool;  
 final AtomicInteger i = state.i;  
  
 synchronizedQueue.enq(i.getAndIncrement());  
 return synchronizedQueue;  
 }  
  
 @Threads(10)  
 @Warmup(batchSize = SET\_SIZE / 10)  
 @Measurement(batchSize = SET\_SIZE / 10)  
 @Benchmark  
 public Pool<Integer> enq\_10\_thread(final SynchronizedQueueAddState state, final Blackhole bh) {  
 final Pool<Integer> synchronizedQueue = state.pool;  
 final AtomicInteger i = state.i;  
  
 synchronizedQueue.enq(i.getAndIncrement());  
 return synchronizedQueue;  
 }  
 }  
  
 @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 SynchronizedQueueThroughputTest {  
 @Threads(1)  
 @Benchmark  
 public Pool<Integer> enq\_1\_thread(final SynchronizedQueueAddState state, final Blackhole bh) {  
 final Pool<Integer> synchronizedQueue = state.pool;  
 final AtomicInteger i = state.i;  
  
 synchronizedQueue.enq(i.getAndIncrement());  
 return synchronizedQueue;  
 }  
  
 @Threads(2)  
 @Benchmark  
 public Pool<Integer> enq\_2\_thread(final SynchronizedQueueAddState state, final Blackhole bh) {  
 final Pool<Integer> synchronizedQueue = state.pool;  
 final AtomicInteger i = state.i;  
  
 synchronizedQueue.enq(i.getAndIncrement());  
 return synchronizedQueue;  
 }  
  
 @Threads(3)  
 @Benchmark  
 public Pool<Integer> enq\_3\_thread(final SynchronizedQueueAddState state, final Blackhole bh) {  
 final Pool<Integer> synchronizedQueue = state.pool;  
 final AtomicInteger i = state.i;  
  
 synchronizedQueue.enq(i.getAndIncrement());  
 return synchronizedQueue;  
 }  
  
 @Threads(4)  
 @Benchmark  
 public Pool<Integer> enq\_4\_thread(final SynchronizedQueueAddState state, final Blackhole bh) {  
 final Pool<Integer> synchronizedQueue = state.pool;  
 final AtomicInteger i = state.i;  
  
 synchronizedQueue.enq(i.getAndIncrement());  
 return synchronizedQueue;  
 }  
  
 @Threads(10)  
 @Benchmark  
 public Pool<Integer> enq\_10\_thread(final SynchronizedQueueAddState state, final Blackhole bh) {  
 final Pool<Integer> synchronizedQueue = state.pool;  
 final AtomicInteger i = state.i;  
  
 synchronizedQueue.enq(i.getAndIncrement());  
 return synchronizedQueue;  
 }  
 }  
  
 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();  
 }  
 });  
 }  
}