**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра ВТ**

отчет

**по лабораторной работе №2**

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

Тема: Barrier

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

Санкт-Петербург

2020

**Цель работы.**

Изучить и реализовать Barrier 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 Simple barrier, Sense barrier и Tree barrier.

**Выводы.**

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

Приложение а

Код Simple barrier

public class SimpleBarrier implements Barrier {  
 int threadsNum;  
 final int size;  
 final Lock lock;  
 final Condition allThreadsArrived;  
  
 public SimpleBarrier(int size) {  
 this.threadsNum = 0;  
 this.size = size;  
 this.lock = new ReentrantLock();  
 this.allThreadsArrived = lock.newCondition();  
 }  
  
 @Override  
 public void await() {  
 lock.lock();  
  
 try {  
 if (++threadsNum == size) {  
 allThreadsArrived.signalAll();  
 } else {  
 while (threadsNum != size) {  
 this.allThreadsArrived.await();  
 }  
 }  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 } finally {  
 lock.unlock();  
 }  
 }  
}

Код Sense barrier

public class SenseBarrier implements Barrier {  
 final AtomicInteger count;  
 final int size;  
 volatile boolean sense;  
 final ThreadLocal<Boolean> threadSense;  
  
 public SenseBarrier(int size) {  
 this.count = new AtomicInteger(size);  
 this.size = size;  
 this.sense = false;  
 this.threadSense = ThreadLocal.*withInitial*(() -> !sense);  
 }  
  
 @Override  
 public void await() {  
 final Boolean mySense = threadSense.get();  
 final int position = count.getAndDecrement();  
  
 if (position == 1) {  
 count.set(size);  
 sense = mySense;  
 } else {  
 while (sense != mySense) {}  
 }  
 threadSense.set(!mySense);  
 }  
}

Код Tree barrier

public class TreeBarrier implements Barrier {  
 final int radix;  
 final Node[] leaf;  
 volatile int leaves;  
 final ThreadLocal<Boolean> threadSense;  
  
 public TreeBarrier(int n, int r) {  
 radix = r;  
 leaves = 0;  
 leaf = new Node[n / r];  
 int depth = 0;  
 threadSense = ThreadLocal.*withInitial*(() -> true);  
 // compute tree depth  
 while (n > 1) {  
 depth++;  
 n = n / r;  
 }  
 Node root = new Node();  
 build(root, depth - 1);  
 }  
 // recursive tree constructor  
 void build(Node parent, int depth) {  
 if (depth == 0) {  
 leaf[leaves++] = parent;  
 } else {  
 for (int i = 0; i < radix; i++) {  
 Node child = new Node(parent);  
 build(child, depth - 1);  
 }  
 }  
 }  
  
 public void await() {  
 int me = ThreadID.*get*();  
 Node myLeaf = leaf[me / radix];  
 myLeaf.await();  
 }  
  
 private class Node {  
 final AtomicInteger count;  
 Node parent;  
 volatile boolean sense;  
  
 public Node() {  
 sense = false;  
 parent = null;  
 count = new AtomicInteger(radix);  
 }  
  
 public Node(Node myParent) {  
 this();  
 parent = myParent;  
 }  
  
 public void await() {  
 boolean mySense = threadSense.get();  
 int position = count.getAndDecrement();  
 if (position == 1) { // I’m last  
 if (parent != null) { // Am I root?  
 parent.await();  
 }  
 count.set(radix);  
 sense = mySense;  
 } else {  
 while (sense != mySense) {}  
 }  
 threadSense.set(!mySense);  
 }  
 }  
  
 private static class ThreadID {  
 private static volatile int *nextID* = 0;  
  
 private static class ThreadLocalID extends ThreadLocal<Integer> {  
 protected synchronized Integer initialValue() {  
 return *nextID*++;  
 }  
 }  
  
 private static ThreadLocalID *threadID* = new ThreadLocalID();  
  
 public static int get() {  
 return *threadID*.get();  
 }  
  
 public static void set(int index) {  
 *threadID*.set(index);  
 }  
 }  
}

Код Performance test

public class PerformanceTest {  
 private static final int *NUM\_ITERATIONS* = 20;  
 private static final int *NUM\_OF\_FORKS* = 2;  
 private static final String *RES\_FILE\_PATH* = "max/src/test/resources/lab2/res.csv";  
  
 @State(Scope.*Benchmark*)  
 public abstract static class BarrierState {  
 @Contended Barrier barrier;  
  
 @Setup(Level.*Iteration*)  
 public abstract void init(BenchmarkParams params);  
 }  
  
 public static class SimpleBarrierState extends BarrierState {  
  
 @Override  
 public void init(BenchmarkParams params) {  
 this.barrier = new SimpleBarrier(params.getThreads());  
 }  
 }  
  
 public static class SenseBarrierState extends BarrierState {  
  
 @Override  
 public void init(BenchmarkParams params) {  
 this.barrier = new SenseBarrier(params.getThreads());  
 }  
 }  
  
 public static class TreeBarrierState extends BarrierState {  
  
 @Override  
 public void init(BenchmarkParams params) {  
 this.barrier = new TreeBarrier(params.getThreads(), 2);  
 }  
 }  
  
 @Fork(*NUM\_OF\_FORKS*)  
 @Warmup(iterations = *NUM\_ITERATIONS*)  
 @Measurement(iterations = *NUM\_ITERATIONS*)  
 @BenchmarkMode(Mode.*SingleShotTime*)  
 @OutputTimeUnit(TimeUnit.*MILLISECONDS*)  
 public static class SimpleBarrierAccelerationTest {  
  
 @Threads(2)  
 @Benchmark  
 public void await\_2\_thread(final SimpleBarrierState state, Blackhole bh) {  
 final Barrier barrier = state.barrier;  
  
 barrier.await();  
 bh.consume(barrier);  
 }  
  
 @Threads(4)  
 @Benchmark  
 public void await\_4\_thread(final SimpleBarrierState state, Blackhole bh) {  
 final Barrier barrier = state.barrier;  
  
 barrier.await();  
 bh.consume(barrier);  
 }  
  
 @Threads(16)  
 @Benchmark  
 public void await\_16\_thread(final SimpleBarrierState state, Blackhole bh) {  
 final Barrier barrier = state.barrier;  
  
 barrier.await();  
 bh.consume(barrier);  
 }  
  
 @Threads(32)  
 @Benchmark  
 public void await\_32\_thread(final SimpleBarrierState state, Blackhole bh) {  
 final Barrier barrier = state.barrier;  
  
 barrier.await();  
 bh.consume(barrier);  
 }  
 }  
  
 @Fork(*NUM\_OF\_FORKS*)  
 @Warmup(iterations = *NUM\_ITERATIONS*)  
 @Measurement(iterations = *NUM\_ITERATIONS*)  
 @BenchmarkMode(Mode.*SingleShotTime*)  
 @OutputTimeUnit(TimeUnit.*MILLISECONDS*)  
 public static class SenseBarrierAccelerationTest {  
  
 @Threads(2)  
 @Benchmark  
 public void await\_2\_thread(final SenseBarrierState state, Blackhole bh) {  
 final Barrier barrier = state.barrier;  
  
 barrier.await();  
 bh.consume(barrier);  
 }  
  
 @Threads(4)  
 @Benchmark  
 public void await\_4\_thread(final SenseBarrierState state, Blackhole bh) {  
 final Barrier barrier = state.barrier;  
  
 barrier.await();  
 bh.consume(barrier);  
 }  
  
 @Threads(16)  
 @Benchmark  
 public void await\_16\_thread(final SenseBarrierState state, Blackhole bh) {  
 final Barrier barrier = state.barrier;  
  
 barrier.await();  
 bh.consume(barrier);  
 }  
  
 @Threads(32)  
 @Benchmark  
 public void await\_32\_thread(final SenseBarrierState state, Blackhole bh) {  
 final Barrier barrier = state.barrier;  
  
 barrier.await();  
 bh.consume(barrier);  
 }  
 }  
  
 @Fork(*NUM\_OF\_FORKS*)  
 @Warmup(iterations = *NUM\_ITERATIONS*)  
 @Measurement(iterations = *NUM\_ITERATIONS*)  
 @BenchmarkMode(Mode.*SingleShotTime*)  
 @OutputTimeUnit(TimeUnit.*MILLISECONDS*)  
 public static class TreeBarrierAccelerationTest {  
  
 @Threads(2)  
 @Benchmark  
 public void await\_2\_thread(final TreeBarrierState state, Blackhole bh) {  
 final Barrier barrier = state.barrier;  
  
 barrier.await();  
 bh.consume(barrier);  
 }  
  
 @Threads(4)  
 @Benchmark  
 public void await\_4\_thread(final TreeBarrierState state, Blackhole bh) {  
 final Barrier barrier = state.barrier;  
  
 barrier.await();  
 bh.consume(barrier);  
 }  
  
 @Threads(16)  
 @Benchmark  
 public void await\_16\_thread(final TreeBarrierState state, Blackhole bh) {  
 final Barrier barrier = state.barrier;  
  
 barrier.await();  
 bh.consume(barrier);  
 }  
  
 @Threads(32)  
 @Benchmark  
 public void await\_32\_thread(final TreeBarrierState state, Blackhole bh) {  
 final Barrier barrier = state.barrier;  
  
 barrier.await();  
 bh.consume(barrier);  
 }  
 }  
  
 public static void main(String[] args) throws Exception {  
 Options opt =  
 new OptionsBuilder()  
 .include(PerformanceTest.class.getName())  
 .jvmArgsAppend("-XX:-RestrictContended")  
 .syncIterations(true)  
 .timeout(TimeValue.*seconds*(5))  
 .shouldFailOnError(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();  
 }  
 });  
 }  
}