Parallel and MultiThreaded Programming

CSYE 7215­­­­

Homework 7

Due: October 31, 2020

Put all your java, compiled class files and documentation into zip file named Homework7.zip and submit it via the dropbox on Canvas before the END of due date. Put your name on

all .java files. There will be a short quiz on this assignment.

1. Provide detail descriptions:

**Blocking Algorithms**

A blocking concurrency algorithm is an algorithm which either:

* Performs the action requested by the thread – OR
* Blocks the thread until the action can be performed safely
* Blocking algorithms **block the thread until the requested action can be performed**.
* Usually it will be the actions of another thread that makes it possible for the first thread to perform the requested action.
* If for some reason that other thread is suspended (blocked) somewhere else in the application, and thus cannot perform the action that makes the first thread's requested action possible, the first thread remains blocked - either indefinitely, or until the other thread finally performs the necessary action.

**Non-Blocking Algorithms**

Non blocking algorithm use low-level atomic machine instructions such as compare and swap operation instead of locks to ensure data integrity under concurrent access

A non-blocking concurrency algorithm is an algorithm which either:

* Performs the action requested by the thread – OR
* Notifies the requesting thread that the action could not be performed
* Non blocking algorithms are considerably more complicated to design and implement than lock based alternatives, but they cab offer significant scalability and liveness advantages
* They coordinate at a finer level of granularity and can greatly reduce scheduling overhead because they don't block when multiple threads contend for the same data.
* In lock based algorithms, other threads cannot make progress if a thread goes to sleep or spins while holding a lock, whereas nonblocking algorithms are impervious to individual thread failure .

**CAS operation**

* CAS has **three operands** a memory location V on which to operate, the expected old value A, and the new value B. CAS atomically updates V to the new value B,but only if the value in V matches the expected old value A; otherwise it does nothing.
* it proceeds with the update in the hope of success, and can detect failure if another thread has updated the variable since it was last examined
* When multiple threads attempt to update the same variable simultaneously using CAS, one wins and updates the variable's value, and the rest lose. But the losers are not punished by suspension, as they could be if they failed to acquire a lock; instead, they are told that they didn't win the race this time but can try again.
* Because a thread that loses a CAS is not blocked, it can decide whether it wants to try again, take some other recovery action, or do nothing. This flexibility eliminates many of the liveness hazards associated with locking.
* The typical pattern for using CAS is first to read the value A from V, derive the new value B from A, and then use CAS to atomically change V from A to B so long as no other thread has changed V to another value in the meantime. CAS addresses the problem of implementing atomic read-modify-write sequences without locking, because it can detect interference from other threads.

2. Consider the following code segments:

a) Compile each Java code and run with “javap” to disassemble the code

b) Show Stack Frame for each code segment

c) Is each code segment Atomic? Yes/No? Explain step-by-step

A)

public class Test {

public static void main(String[] args) {

int i=0;

i++;

System.out.println(i);

}

}

Dissembled Code

Graphical user interface, application

Description automatically generated

Operations:

**Iconst\_0** 🡪 Push the int constant onto the operand stack

**Istore\_1** 🡪 Store the integer from LVA index 1 into local variable

**iinc 1 , 1** 🡪 increment the local variable with constant here first “1” is index of LVA and second “1” is constant

**getstatic** 🡪 initialize reference class PrintStream

**iload\_1** 🡪Load the integer value from LVA index 1 onto operand stack

**invokevirtual** 🡪 invoke println method

Current program is non-atomic as we can see in dissembled code that increment operation will first read the local variable value then increment the value and get the updated value using **iinc opcode.**

iinc operation is atomic in nature in terms of increment and save but in **multi- threaded environment read , increment and get operation can interleave over each other and give the incorrect result**.

B)

public class Test {

public static void main(String[] args) {

int i=1;

System.out.println(i);

}

}

Graphical user interface, website

Description automatically generated

Operations:

**Iconst\_0** 🡪 Push the int constant onto the operand stack

**Istore\_1** 🡪 Store the integer from LVA index 1 into local variable

**getstatic** 🡪 initialize reference class PrintStream

**iload\_1** 🡪Load the integer value from LVA index 1 onto operand stack

**invokevirtual** 🡪 invoke println method

The given program is atomic in nature as we are not modifying / updating the value of local variable. In multithreaded environment it will give same output.

**3. Conversion of Number system:**

a) 1011010010110101 convert to HEX —> convert to Decimal

Convert to hex

1011 = 1\* 2^3+ 0\*2^2 +1\*2^1 +1\*2^0 = 11= B

0100 = 0\* 2^3+ 1\*2^2 +0\*2^1 +0\*2^0 = 4= 4

1011 = 1\* 2^3+ 0\*2^2 +1\*2^1 +1\*2^0 = 11= B

0101 = 0\* 2^3+ 1\*2^2 +0\*2^1 +1\*2^0 = 5= 5

Hex = B4B5

HEX 🡪 Decimal

B4B5 = 11 \* 16^3 + 4 \* 16^2 + 11 \* 16^1+ 5\* 16^0

= 46261

b) 0xABCD Convert to Binary —> convert to Decimal

0x ABCD

A = 1010

B = 1011

C = 1100

D = 1101

Binary = 1010101111001101

Binary to Decimal

1010101111001101

1\*2^15+0^2^14+1\*2^13+0\*2^12 +1\*2^11+0^2^10+1\*2^9+1\*2^8+1\*2^7+1\*2^6+0\*2^5+0\*2^4+1\*2^2+1\*2^2+0\*2^1+1\*2^0

=43981

c) 11010010110101 Convert to HEX. —> convert to Decimal

0011 =1\*2^1+1\*2^0 = 3

0100 =0\*2^3 + 1\*2^2 + +0\*2^1+ +0\*2^0 =4

1011 =1\*2^3 + 0\*2^2 + +1\*2^1+ +1\*2^0 =B

0101 =0\*2^3 + 1\*2^2 + +0\*2^1+ +1\*2^0 =5

Hex = 34B5

Hex -> Decimal

34B5= 3\*16^3 +4\* 16^2+11\* 16^1+5\*16^0

=13493

4. The architecture of your machine is 32-bits. Consider the following code and its

compiled disassembly:

uint64\_t sharedValue = 0;

void storeValue() {

sharedValue = 0x100000002;

}

$ gcc -O2 -S -masm=intel test.c

$ cat test.s

...

mov DWORD PTR sharedValue, 2

mov DWORD PTR sharedValue+4, 1

ret

...

In given assembly code, the compiler implemented the 64-bit assignment using two separate machine instructions. The first instruction **sets the lower 32 bits to** **0x00000002**, and the second sets the upper 32 bits **to 0x0000000**1. Clearly, **this assignment operation is not atomic**

sharedValue = 0x100000002 =**1** **00000000000000000000000000000010**

00000000000000000000000000000010 – first 32 bits

00000000000000000000000000000001—second 32 bits

a) If sharedValue = 0x100000004; What changes in assembly code?

sharedValue = 0x100000004= **1** **00000000000000000000000000000100**

00000000000000000000000000000100 – first 32 bit =4

00000000000000000000000000000001 – second 32 bits =1

So assembly code would be like

**mov DWORD PTR sharedValue, 4**

**mov DWORD PTR sharedValue+4, 1**

b) What are the root causes of Torn-read or Torn-write? Identify Thread

scenarios which causes torn-read or torn-write cases.

* If the reading and writing operations are not atomic then in concurrent reading and writing operation can lead the torn-read and torn write
* As we know that the assignment operation in current program is non atomic so if sharedValue is accessed by concurrently by different threads server things can go wrong
* If a thread calling **storeValue** is preempted between the two machine instructions, it will leave the value of 0x0000000000000002 in memory as **torn write**. At this point, if another thread reads sharedValue, it will receive this completely bogus value which nobody intended to store.
* Reading concurrently from sharedValue brings its own set of problems while reading the data concurrently compiler will implement the load operation using two machine instructions the first read the lower 32 bits into eax and the second reads the upper 32 bit into edx.
* In this case if a concurrent store to sharedValue become visible between the two instructions it will result to **torn read even if the concurrent store was atomic**

5. Consider the following two classes:

public class SimulatedCAS {

private int value;

public synchronized int getValue() { return value; }

public synchronized int compareAndSwap(int expectedValue, int newValue) {

int oldValue = value;

if (value == expectedValue)

value = newValue;

return oldValue;

}

}

public class CasCounter {

private SimulatedCAS value;

public int getValue() {

return value.getValue();

}

public int increment() {

int oldValue = value.getValue();

while (value.compareAndSwap(oldValue, oldValue + 1) != oldValue)

oldValue = value.getValue();

return oldValue + 1;

}

}

1. **Explain each class**

* Simulated CAS class has two synchronized method one will helps us to retrieve the latest value of value variable
* Second method is Compare and Swap method which compare the old value with expected value(expected old values) just to verify that no other thread has updated the value if so then function return the updated value only and doesn’t perform any swap.
* If the old value is not changed then the function will swap the new value with old value and return the updated value.
* Compare and Swap is performing the atomic operation.
* CASCounter class implements a **thread safe counter using CAS**. The increment operation follows the canonical form fetch the old value, transform it to the new value (adding one), and use CAS to set the new value.
* If the CAS fails, the operation is immediately retried. CasCounter does not **block, though it may have to retry several times if other threads are updating the counter at the same time**

1. **What are the differences between two classes?**

* CAS Counter class use the simulated CAS class to perform the increment operation
* Both the class perform the atomic operation only but simulated CAS class has synchronized method to perform atomic operation.

1. **Does one benefit over other? Explain details**

* CAS Counter class use the simulatedCAS class to perform the automic increment operation so it definitely increase the performance and also avoid the unnecessary blocking issues.

**d) Does one perform better than other? Why?**

* CAS based counters significantly outperform lock based counters if there is even a small amount of contention, and often even if there is no contention.
* The fast path for uncontended lock acquisition typically requires at least one CAS plus other lock related housekeeping, so more work is going on in the best case for a lock based counter than in the normal case for the CASbased counter.
* Since the CAS succeeds most of the time (assuming low to moderate contention), the hardware will correctly predict the branch implicit in the while loop, minimizing the overhead of the more complicated control logic.
* Executing a CAS from within the program involves no JVM code, system calls, or scheduling activity. What looks like a longer code path at the application level is in fact a much shorter code path when JVM and OS activity are taken into account.

6. Consider the following code using AtomicLong

a) Compile code and Run

b) Rewrite the code as BlockingAlgorithm

c) Compare performance (a) and (b)

import java.util.concurrent.atomic.AtomicLong;

public class TestThread {

static class Counter {

private AtomicLong c = new AtomicLong(0);

public void increment() {

c.getAndIncrement();

}

public long value() {

return c.get();

}

}

public static void main(final String[] arguments) throws InterruptedException {

final Counter counter = new Counter();

//1000 threads

for(int i = 0; i < 1000 ; i++) {

new Thread(new Runnable() {

public void run() {

counter.increment();

}

}).start();

}

Thread.sleep(6000);

System.out.println("Final number (should be 1000): " + counter.value());

}

}

a) Compile code and Run

A screenshot of a computer screen

Description automatically generated

b) Rewrite the code as BlockingAlgorithm

Graphical user interface, text

Description automatically generated

c) Compare performance (a) and (b)

In (a). we have used the non blocking algorithm to increment the value of variable using AtomicLong while in (b) we have used the synchronized methods to implement blocking

If we check the performance of (a) and (b) we find that the in **moderate contention non-blocking algorithm performs much better than blocking because the CAS succeeds** on first try and the penalty for contention when it does occur does not involve thread suspension and context switching, just a few more iterations of the loop.

Under high contention when many threads are pounding on a single memory location lock-based **algorithms start to offer better throughput than nonblocking ones because when a thread blocks, it stops pounding and patiently waits its turn, avoiding further contention**.

|  |  |  |
| --- | --- | --- |
| No of Threads | Time in Blocking Algorithms in ms | Time in non blocking algorithm in ms |
| 100000 | 3978 | 3877 |
| 1000000 | 35643 | 36119 |

7. Java does not support “AtomicDouble”. There are two code segments: 1) creates an instance of Double, and 2) uses AtomicReference to wrap “Double”. Compile each code segment and compare performance.

package com.logicbig.example;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.TimeUnit;

public class NoAtomicReferenceExample {

private static Double sum;

public static void main(String[] args) throws InterruptedException {

for (int k = 0; k < 5; k++) {

sum=0d;

ExecutorService es = Executors.newFixedThreadPool(50);

for (int i = 1; i <= 50; i++) {

int finalI = i;

es.execute(() -> {

sum+=Math.pow(1.5, finalI);

});

}

es.shutdown();

es.awaitTermination(10, TimeUnit.MINUTES);

System.out.println(sum);

}

}

}

package com.logicbig.example;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.TimeUnit;

import java.util.concurrent.atomic.AtomicReference;

public class AtomicReferenceExample {

private static AtomicReference<Double> sum = new AtomicReference<>();

public static void main(String[] args) throws InterruptedException {

for (int k = 0; k < 5; k++) {

sum.set(0d);

ExecutorService es = Executors.newFixedThreadPool(50);

for (int i = 1; i <= 50; i++) {

int finalI = i;

es.execute(() -> {

sum.accumulateAndGet(Math.pow(1.5, finalI),

(d1, d2) -> d1 + d2);

});

}

es.shutdown();

es.awaitTermination(10, TimeUnit.MINUTES);

System.out.println(sum.get());

}

}

}

Atomic Reference example:

In atomic reference every time we will get the same output.

Text

Description automatically generated

Non Atomic reference example output :

**In Non atomic result would not be same every time**

Text

Description automatically generated

* In NoAtomicReference class we get the different output if we run the program multiple times this thing happens because we are not protecting the shared instance of Double , also we have not implemented any locking mechanism.
* Hence we can not compare the performance of NoAtomic reference class and Atomic Reference class because NoAtomic Reference class is not accurate.
* If we want to compare the performance of these two class then we need to implement any locking mechanism in NoAtomicReference class to get accurate output.
* If we implement the locking mechanism in NoAtomicReference class then in high contention level it will perform better then AtomicReference class, on the other hand Atomic reference class will perform very well in low and moderate contention level.