Parallel and MultiThreaded Programming

CSYE 7215

Homework 3

Due: October 4, 2020

Put all your java, compiled class files and documentation into a zip file named Homework3.zip and submit it via the dropbox on the Canvas before the END of due date. Put your name on all .java files. There will be a short quiz on this assignment.

1. Explain:

**Basic components of computer architecture ? Memory Heap?**

Memory heap is an area of pre-reserved computer [**main storage**](https://searchstorage.techtarget.com/definition/primary-storage)**(**[**memory**](https://searchstorage.techtarget.com/definition/memory-card)**)** that a program [process](https://whatis.techtarget.com/definition/process) can use to store data in some variable amount that won't be known until the program is running.

**When you compile and execute your program (eg: Hello.java),**

**what happens? Provide details of execution environment.**

**JVM memory architecture components?**

Let’s consider the following simple hello.java file

Public class hello{

Public static void main(String args[]){

// code

}

}

Execution steps

1. java compiler compile this file and convert into hello.class file and send to JVM.
2. Once the JVM receive this file it perform following steps
   1. It will load the class file in main memory which is know as dynamic class loading using following class loader
      1. Bootstrap classloader
         * + Responsible for load the class from bootstrap classpath
      2. Extension classloader
         * + Responsible for load the classes from ext folder
      3. Application classloader
         * + Responsible for load application level classpath
   2. Inside the linking phase JVM verify the weather the generated bytecode is proper or not then allocate the memory for static variables and resolve the symbolic memory reference with original reference from memory area.
3. After all this process Run time data area comes into picture. It is divided into five major parts .
   1. **Method area** : All the class-level data will be stored here, including static variables. There is only one method area per JVM, and it is a shared resourc
   2. **Heap Area** – All the Objects and their corresponding instance variables and arrays will be stored here. There is also one Heap Area per JVM. Since the Method and Heap areas share memory for multiple threads, the data stored is not thread-safe
   3. **Stack Area** – For every thread, a separate runtime stack will be created. For every method call, one entry will be made in the stack memory which is called Stack Frame. All local variables will be created in the stack memory
   4. **PC Registers** – Each thread will have separate PC Registers, to hold the address of current executing instruction once the instruction is executed the PC register will be updated with the next instruction.
   5. **Native Method stack**s – Native Method Stack holds native method information. For every thread, a separate native method stack will be created.
4. Once the bytecode assigned to run time data area , execution engine start executing bytecodes. JVM execution engine contains following components
   1. **Interpreter** – The interpreter interprets the bytecode faster but executes slowly. The disadvantage of the interpreter is that when one method is called multiple times, every time a new interpretation is required.
   2. **JIT Compiler** – The JIT Compiler neutralizes the disadvantage of the interpreter. The Execution Engine will be using the help of the interpreter in converting byte code, but when it finds repeated code it uses the JIT compiler, which compiles the entire bytecode and changes it to native code. This native code will be used directly for repeated method calls, which improve the performance of the system.
   3. **Garbage Collector**: Collects and removes unreferenced objects. Garbage Collection can be triggered by calling System.gc(), but the execution is not guaranteed. Garbage collection of the JVM collects the objects that are created.
5. JVM also contains JNI and native method library. JNI will interact with native method library and provides the native library required for the execution.

**JVM Heap?**

- All the Objects and their corresponding instance variables and arrays will be stored here. There is also one Heap Area per JVM. Since the Method and Heap areas share memory for multiple threads, the data stored is not thread-safe

- JVM heap is divided into 2 parts

1) Young generation

2) Old Generation

1) **Young generation**

- This is reserved for newly created object. Young Gen includes three parts Eden Memory and two Survivor Memory spaces (S0, S1)

- Most of the newly-created objects goes Eden space. When Eden space is filled with objects, Minor GC (a.k.a. Young Collection) is performed and all the survivor objects are moved to one of the survivor spaces.

- Objects that are survived after many cycles of GC, are moved to the Old generation memory space. Usually it’s done by setting a threshold for the age of the young generation objects before they become eligible to promote to Old generation

2) **Old generation**

- This is reserved for containing long lived objects that could survive after many rounds of Minor GC

- When Old Gen space is full, Major GC is performed

**JVM Non-Heap?**

JVM non -heap include the permanent generation. It stores the per class structure such as runtime constant pool, field and method data and interned string.

**Cache?**

Cache is a special type of very high-speed memory. It is used to speed up and synchronizing with high-speed CPU. Cache memory is an extremely fast memory type that acts as a buffer between RAM and the CPU. It holds frequently requested data and instructions so that they are immediately available to the CPU when needed.

CPU

Cache

Main Memory

**Native Cache?**

The Just-In-Time (JIT) compiler stores the compiled code in an area called code cache. It is a special heap that holds the compiled code. This area is flushed if its size exceeds a threshold and these objects are not relocated by the GC

**Java Native Interface (JNI)?**

JNI will be interacting with the Native Method Libraries and provides the Native Libraries required for the Execution Engine

**Why JVM is threadSafe?**

In JVM for every thread, a separate **runtime stack will be created**. For every method call, one entry will be made in the stack memory which is called Stack Frame. All local variables will be created in the stack memory. The stack area is **thread-safe since it is not a shared resource.**

**2. How does Java Memory Model and Hardware Memory Architecture are different and how they work together?**

Diagram

Description automatically generated

* Java memory model is different than the Hardware memory architecture.

As we know that inside the JVM memory **divide s into thread stack and the heap while in hardware memory thread stack and heap both located in main memory.**

* Each thread running in the Java virtual machine has its own thread stack. The thread stack contains information about what methods the thread has called to reach the current point of execution
* While hardware memory **model has different layers of memory such as CPU register, CPU cache memory level 1 and level 2 and main memory**. part of the heap and thread stack is residing in each layer of memory.

**3. When objects and variables can be stored in various different memory areas in the computer, certain problems may occur. The two main problems are:**

**a) Visibility of thread updates (writes) to shared variables, provide an example? One way**

**to solve visibility issue is to use Volatile keyword, how does it work? Provide example.**

**b) Race conditions when reading, checking and writing shared variables.**

**Provide example of each, Explain as how these two problems can occur.**

**Visibility of thread update**

* If two or more threads are sharing an object, without the proper use of either volatile declarations or synchronization, updates to shared object made by one thread may not be visible to other threads.
* Imagine that shared object is initially stored in main memory. A thread running on CPU one then reads the shared object into its CPU cache. This makes a change to shared object.
* As long as the CPU cache has not been flushed back to main memory, the changed version of the shared object is not visible to threads running on other CPUs. This way each thread may end up with its own copy of the shared object, each copy sitting in a different CPU cache.
* The following diagram illustrates the sketched situation. One thread running on the left CPU copies the shared object into its CPU cache, and changes its count variable to 2.
* This change is not visible to other threads running on the right CPU, because the update to count has not been flushed back to main memory yet.
* To solve this problem you can use Java's volatile keyword. The volatile keyword can make sure that a given variable is read directly from main memory, and always written back to main memory when updated.

Diagram

Description automatically generated

Public class TestVisibility implements Runnable{

Int count =0; ***// we can solve the issue by making this volatile***

Public void run(){

Count++;

}

}

Public class main{

Public static void main(String args[]){

TestVisibility ts = new TestVisibility();

Thread t1 = new Thread(ts);

Thread t2 = new Thread(ts);

t1.start();

t2.start();

}

}

**Race conditions**

* if two or more threads share an object, and more than one thread updates variables in that shared object, race conditions may occur.
* Thread A reads variable count shared object into its CPU cache, and Thread B does the same but into a different CPU cache. Now thread A adds one to count, and thread B does the same. Now var1 has been incremented two times, once in each CPU cache.
* If these increments had been carried out sequentially, the variable count would be incremented twice and had the original value + 2 written back to main memory.
* However, the two increments have been carried out concurrently without proper synchronization. Regardless of which thread A and B that writes its updated version  
  of count back to main memory, the updated value will only be 1 higher than the original value, despite the two increments.

This diagram illustrates an occurrence of the problem with race conditions:

Diagram

Description automatically generated

* To solve race condition problem we can use a Java **synchronized block**. A synchronized block guarantees that **only one thread can enter a given critical section of the code at any given time**. Synchronized blocks also guarantee that all variables accessed inside the synchronized block will be **read in from main memory, and when the thread exits the synchronized block, all updated variables will be flushed back to main memory again**, regardless of whether the variable is declared volatile or not.

**Program :**

Public class TestVisibility implements Runnable{

Int count =0; // we can solve the issue by making this **volatile**

Public void run(){

synchronized(this)

{

Count++;

}

}

}

Public class main{

Public static void main(String args[]){

TestVisibility ts = new TestVisibility();

Thread t1 = new Thread(ts);

Thread t2 = new Thread(ts);

t1.start();

t2.start();

}

}

4. In Homework2, you defined Student class, and created 25 student threads and created one GraderThread. Change previous homework, and have GraderThread to write final grade to file “FinalGrades”. This is how things going to work:

A) Create 40 student threads, each thread records 3 scores (homework, midterm-exam, final-exam), and each score is randomly generated between 70 to 100 with incremental 1second interval. Use a List to store scores and is set to zeros initially. Use Map with key/value, (List, threadId).

B) Each student thread first generates homework score within 1 second, stores score in List [0], and then writes List to map. Next, it generates midterm-exam score within the next second, stores score to List[1] and writes List to map. Next, it generates final-exam score within the next second, stores score in List[2] and writes List to map.

C) GraderThread randomly reads scores from Map, calculates final grade for Letter

A, B, C, D, F, and writes the final grade to file “FinalGrades”.

D) Each student thread, Reads final grade from file “FinalGrades” and reports its grade,

Notes: Things to think about: 1) How to protect Map and File? 2) student Threads writing to Map and Reading from File, 3) GraderThread is reading Map and writing to File. 4) student threads write List to Map with incremental List update for each score. Grader thread reads Map for each student scores, the Grader can move to calculate letter grade calculation and write the grade to the file?, the Grader thread does not need to wait for all student threads, once each student completes scores, grader thread can move to calculate grade. 5) Student threads often read file to see if final grade is submitted by the grader thread, think thread safety for all reads and writes. Carefully read steps described above.

Note: Because there are many read and write operations to the Map and File, between student threads and grader thread, there is possibility for Race Condition, How does that is possible?, Explain.

**Possible race condition Explanation:**

* In a given problem we are creating 40 different threads using student class.
* Here each thread will generate different score for homework, midterm and final exam in difference of 1 second and store that score inside the List and store that list inside the Map collection.
* Concurrently grader thread will try to access that Map object and list of all the grades to generate Letter grade. In this case race condition wilk occur All the student thread try to write data in map object and we need to make sure that grader thread will access the data once the student thread complete their write operation.
* We have to writes program in such way that the grader thread should wait until student thread complete their write operation. We can achieve this using synchronized block and wait() and notifyall() **functions**
* **Second** part of the problem is once the Grader thread calculate the letter grade it will write that data in grade.txt file and all the student thread will try to read that file and get the updated grade. As this two operations occurs concurrently race condition may occur to get access the grade.txt file
* We can also handle this situation using synchronized block , wait() and notifyall() method
* For implementation detail you can check the code files.

Text

Description automatically generated

Text

Description automatically generated

**5. Reverse engineer the following JVM memory model explain what object it represents in detail?**

Diagram

Description automatically generated

**Aforementioned Java model represents the following code.**

class Person {

int pid;

String personName;

// constructor, setters/getters

Void setPersonName(String name)

{

this. personName = name ;

}

}

public class Driver {

public static void main(String[] args) {

int id = 23;

String pName = "Jon";

Person p = null;

p = new Person(id, pName);

}

}

**Explanation:**

* Inside the call stack we can see that when we start the program it will call the main() function then Person() and at last setPersonName() function.
* Stack memory shows the how the object are created. In first block we can **see that Pname string is directly pointing to string constant pool** while int is **primitive type it store it’s value inside the memory stack and person object refers to Null object.**
* After that we are calling person() constructor method and assign string value using setPerName() methed

**6. Provide Call Stack, Stack Memory, and Heap Space for the following code:**

class MyThread implements Runnable {

String name;

Thread t;

MyThread String thread){

name = thread;

t = new Thread(this, name);

System.out.println("New thread: " + t);

t.start();

}

public void run() {

try {

for(int i = 5; i > 0; i--) {

System.out.println(name + ": " + i);

Thread.sleep(1000);

threadname

}

}catch (InterruptedException e) {

System.out.println(name + "Interrupted");

}

System.out.println(name + " exiting.");

}

}

class MultiThread {

public static void main(String args[]) {

new MyThread("One");

new MyThread("Two");

new NewThread("Three");

try {

Thread.sleep(10000);

} catch (InterruptedException e) {

System.out.println("Main thread Interrupted");

}

System.out.println("Main thread exiting.");

}

Above program creates the three instance of the MyThread class

And it also contain the thread object which start in object initialization

Call Stack :

|  |
| --- |
| Run() |
| Run() |
| Run() |
| Start() |
| new Thread() |
| MyThread(“three”) |
| Start() |
| new Thread() |
| MyThread(“two”) |
| Start() |
| new Thread() |
| MyThread(“One”) |
| Main() |

// Run method are depended on how operating system assign the memory to particular thread to execute that method

* Call stack store the current point of execution it perform the operation in **LIFO (Last in First out)** manner

As we are creating separate three threads we JVM creates **three different Stack for each thread**

Thread One Thread Two Thread three

Stack Frame

Stack Frame

Stack Frame

Stack Frame

Stack Frame

Stack Frame

Stack Frame

Stack Frame

Stack Frame

Stack Frame

Stack Frame

Stack Frame

Stack Frame

Stack Frame

Stack Frame

It contains

1. Local variable array
2. Operand stack
3. Constant pool reference

In Above figure you can find the thread different stack thread for each thread and each thread stack contains the stack frame.

So MyThread class have two reference variable one is String name and another is thread t so stackframe will store the reference of both. The actual address of String and Thread instance is inside the JVM heap.

Call Stack

Thread three

Stack Frame

Thread two

Stack Frame

Thread one

Stack Frame

JVM HEAP

Thread t

String name

**7. Consider the following code segments:**

**A) Compile the following code segments**

**B) Use Javap to disassemble code, report**

**Javap -l -v -c WaitNotifyTest.class**

**C) Explain the report**

package com.journaldev.concurrency;

public class Message {

private String msg;

public Message(String str){

this.msg=str;

}

public String getMsg() {

return msg;

}

public void setMsg(String str) {

this.msg=str;

}

}

package com.journaldev.concurrency;

public class Notifier implements Runnable {

private Message msg;

public Notifier(Message msg) {

this.msg = msg;

}

@Override

public void run() {

String name = Thread.currentThread().getName();

System.out.println(name+" started");

try {

Thread.sleep(1000);

synchronized (msg) {

msg.setMsg(name+" Notifier work done");

msg.notify();

// msg.notifyAll();

}

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}}

package com.journaldev.concurrency;

public class WaitNotifyTest {

public static void main(String[] args) {

Message msg = new Message("process it");

Waiter waiter = new Waiter(msg);

new Thread(waiter,"waiter").start();

Waiter waiter1 = new Waiter(msg);

new Thread(waiter1, "waiter1").start();

Notifier notifier = new Notifier(msg);

new Thread(notifier, "notifier").start();

System.out.println("All the threads are started");

}

}

**Javap command disassembles the one more classfile.**

Please find the attach screenshot of Javap -l -v -c WaitNotifyTest.class

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Java compile file consist following structure

**Screenshot 1 shows all this details.**

**magic,minor\_version,major\_version** 🡪 specifies information about the version of the class and the version of the JDK this class was compiled for.

**constant\_pool** 🡪 The JVM maintains a per-type constant pool, a run time data structure that **is** similar to a symbol table although it contains more data. Byte codes in Java require data, often this data is too large to store directly in the byte codes, instead it is stored in the constant pool and the byte code contains a reference to the constant pool.

**access\_flags** 🡪 provides the list of modifiers for this class.

**this**\_**class**🡪 index into the constant\_pool providing the fully qualified name of this class

**super**\_**class**🡪 index into the constant\_pool providing a symbolic reference to the super class

**interfaces**🡪 array of indexes into the constant\_pool providing a symbolic references to all interfaces that have been implemented.

**Fields**🡪 array of indexes into the constant\_pool giving a complete description of each field.

**Attributes** 🡪 array of different value that provide additional information about the class including any annotations with RetentionPolicy.CLASS or RetentionPolicy.RUNTIME

**Methods** 🡪 array of indexes into the constant\_pool giving a complete description of each method signature, if the method is not abstract or native then the bytecode is also present.

Each method contains 4 areas

* signature and access flags
* byte code
* LineNumberTable – this provides information to a debugger to indicate which line corresponds to which byte code instruction, for example line 6 in the Java code corresponds to byte code 0 in the sayHello method and line 7 corresponds to byte code 8.
* LocalVariableTable – this lists all local variables provided in the frame, in both examples the only local variable is this.

Please find following screenshot for reference.

Text

Description automatically generated

**Following operand code used in this report**

**new** 🡪 Create new object

**dup**. 🡪 Duplicate the top operand stack value

**ldc** 🡪 push item from run-time constant pool

**invokespecial** 🡪 Invoke instance method; special handling for superclass, private, and instance initialization method invocations

**astore**\_1. 🡪 Store reference into local variable

**aload**\_1-🡪 Load reference from local variable

**invokevirtual** 🡪 Invoke a class (static) method

**getstatic** 🡪 Get static field from class

**8. Write a Java Test program that creates a Thread and has this “calculate” method.**

**public int calculate (int i, int j, int k) {**

**int a = i \* j \* k + k \* i + 10 ;**

**return a;**

**}**

**A) Compile Java code**

**B) Run “Javap -l -v -c Test.class”**

**C) Discuss the report, the output, Local Variable Array and**

**Operand Stack**

**D) Provide a drawing to show JVM step-by-step Execution of Local**

**Variable Array (LVM) and Operand Stack**

Report screenshot

Text

Description automatically generated

According to give report we can see that there are 5 local variables mentioned in local variable table

A picture containing text

Description automatically generated

**LOCAL VARIABLE ARRAYA**

|  |  |
| --- | --- |
| **Index** | **Variable** |
| 0 | this |
| 1 | Int i |
| 2 | Int j |
| 3 | Int k |
| 4 | Int a |

Operand Stack

0: iload\_1. 🡪 Load the value of index 1 which is **i**

1: iload\_2. 🡪 // Load the value of index 2 which is **j**

2: imul 🡪 perform multiplication

3: iload\_3 🡪 Load the value of index 3 which is **k**

4: imul. 🡪 perform multiplication

5: iload\_3 🡪 Load the value of index 3 which is **k**

6: iload\_1 🡪 Load the value of index 1 which is **i**

7: imul 🡪 perform multiplication

8: iadd 🡪 perform addition

9: bipush 10 🡪 push the 10 into operand stack

11: iadd 🡪 perform addition

12: istore 4 🡪 store the result value at index 4

14: iload 4 🡪 Load the value of index 4 which is **j**

16: ireturn 🡪 return that value

**Operand stack operation :**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Before  Loading | iload\_1 | iload\_2 | imul | iload\_3 | imul | iload\_3 | iload\_1 | imul | iadd | bipush |
|  | i | i | I\*j | k | I\*j\*k | k | k | K\*i | I\*j\*k  +  K\*i | 10 |
|  |  | j |  |  |  |  | i |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
| iadd | istore | ireturn |
| i \* j \* k + k \* i + 10 | Store a in LVA at 4th index | a |