CS6235
Assignment 1

Rushabh Lalwani
CS18B046

The assignment expects that students will write their own code. All the Java codes snippets - should be valid Java code that is compilable using a Java compiler. Compile your code, test it and only then use it in the assignment.

1. [10+10] Write Java code snippets to show (i) data-races, (ii) how atomicity violations can exist despite race freedom (using synchronized blocks or methods).

i) A **data race** occurs when two or more threads access the same memory location simultaneously and at least one of them is a write to that location. This could lead into memory inconsistencies.

The program mentioned in **DataRaces.java** uses the implemented Runnable class Task from **Task.java**. The program is simple - consists of calculating array sum with different ways - sequentially and concurrently.

Before we begin any computation method, we first initialize the array elements to their corresponding index sequentially. Now we find the sum of all of them sequentially using the method *calculateArraySumSequential*() and note the time and result as reference of our further calculation.

Now we try to compute the same sum using multiple threads concurrently with the help of *ExecutorService* and *Executor* interface provided by Java. We need to create tasks (using Task class) and provide them as argument to execute in a *threadpool* with provided number of threads. We will discuss 3 types of compute() method in the following functions which is invoked from run() method of the Task class:

1. *calculateArraySumParallel*() : The compute method simply iterates over the range of indices given to the task and adds the value to the final sum (*parallelSum*). Here we have a data race on the location of the sum variable as multiple threads will try to write to it simultaneously and thus the final answer will be lesser than expected.

2. *calculateArraySumSynchronize*() : As there was data race on sum variable in the above method, we do this computation (adding of each value of array to the sum variable (s*ynchronizedSum*) in a synchronized method instead. Here we simply call the **synchronized** method *addSynchroizedElement*() for every element in each task. This will lead to correct answer without any memory inconsistency. But it is very costly and does nothing better than the sequential method as it uses lock on each index which makes it modifying the sum value in a lock for each index.

3. *calculateArraySumModifiedSynchronize*() : Here we are doing similar to the above method. Except instead of modifying the final sum variable, we maintain a local sum variable for each Task which computes the sum sequentially for its corresponding range and then after this adds this local result into the sum variable (*modifiedSynchronizedSum*) in a synchronized method *addSumModifiedSynchronized*(). This method is faster as we have only few synchronized calls (number of threads) .

For N = 10,000,000, we got the following results after running 10 times and averaging the time taken:

#Threads	Sequential method (ms)	Parallel method (ms)	Sync method (ms)	Modified sync method (ms)
10	520	1560	3075	67
20	536	1500	3075	37
50	516	1560	3090	41
100	517	1545	3137	49

ii) Atomic operations are instructions that execute atomically without any interruption avoiding data races on the writes to the variables in the atomic block.

The program mentioned in **Atomicity.java** uses the implemented Runnable class Task from **Task.java**. The program consists of series of deposits/withdrawals on the balance variable. We have assumed that any transaction will never lead to balance in account negative and hence we can perform all these operations concurrently. This example is quite similar to the example mentioned above for data races.

Before we begin any computation method, we first initialize the array elements (all the transactions) in the *setTransactions*() using a random generator in *generateRandomTransactions*() .We have taken care to avoid any set of transactions resulting into negative account balance in this initialization.

Now first we compute the final balance sequentially using *calculateFinalBalanceSequentially*(). We note the value and time and keep it as a reference. Further we have two more methods to compute this transaction. In both the methods to avoid data race on the final sum variable, we have updated the sum variable using synchronized methods.

1. *calculateFinalSumSynchronously*() : In this method, for each task, we first get the value of final sum static variable (*synchronousBalance*), modified it locally and then updated it back into static variable using synchronized method (*updateSynchronousBalance*()). So now we don't have any data race but the computation is still not atomic as within the time we read and update the balance variable, some other task would already read and modify it and our previous task update will shadow the changes by this other task leading to atomic violations.

2. *calculateFinalSumModifiedSynchronously*() : To avoid the above inconsistency, we used the fact that we don't need to read the previous value of balance. We can find the transaction sum first and then directly add this local value by updating the static sum variable in a synchronized method (*updateModifiedSynchronousBalance*()). This avoids all the atomic violations we had previously.

We kept the number of transactions very large (100,000,000) and used different number of threads and averaged the time over 10 times to get the following results:

#Threads	Sequential method (ms)	Sync method (ms)	Modified Sync method (ms)
10	990	120	55
20	880	64	64
50	920	82	62
100	838	88	75

2. [10+5] Write a Java program and use it to illustrate Amdahl's law. Show the execution time numbers to empirically establish the law. You may use any multi-core system (with at least 8 cores) for the experiment.

Amdahl's Law gives an upper bound on the overall performance improvement obtained from optimizing our program. The speedup is bounded by the ideal theoretical speedup value:

$$\text{Speedup} = \frac{1}{1 - f + \frac{f}{N}}$$

Where f = fraction of program which is parallelizable and N is the number of processors.

The program mentioned in **Amdahl.java** uses the implemented Runnable class Task from **Task.java**. The program is simple - to compute the sum of reciprocals of array elements.

Our arrays size is kept large - 100, 000, 000. We used a sequential and parallel way to compute this sum. Before we begin any computation method, we first initialize the array elements in the *initializeArraySequential*() method. We use a sequential sum method *repeatSequentialComputation*() to compute the sum sequentially for 10 times (*repeatComputation*) by repeatedly calling *calculateArraySumSequential*() and averaged the time taken. We used this as a reference sum to later confirm the result we obtained from parallel execution.

Similarly, our parallel method *repeatParallelComputation*() also computes the sum by repeatedly calling *calculateArraySumParallel*(). We pass a *numThreads* parameter to indicate the number of tasks used for execution. Each task computes the sum locally and later using a synchronized method adds this value in the final sum. This addition of final sum can be treated as sequential part of our parallel method execution. As the sequential part (equivalent to the number of threads) is very less than the parallel part we ignore this fraction simply for the sake of this example.

For e.g. at max when we use number of task (T) = 128 for the array size (S) = 100,000,000.
The parallel computation consists of summing S values. And sequential part consists of summing the later T values and few instructions setting up the task execution concurrently say c instructions where c << S. Hence, f = S/(S+T+c)
And 1-f = (T+c)/(S+T+c) ~ 10^{-6}.
Hence we ignore this very small value of (1-f) and say that ideal speedup = N.

In windows, by changing the affinity for eclipse.exe, we varied the number of cores (N) to use in the program. The results obtained for different number of processors are

N = 2 , Ideal speedup = 2, Average sequential time = 231ms.

#Tasks	Avg. Time taken (ms)	SpeedUp = Avg.Time taken / Avg. Sequential time
2	129	1.78
4	132	1.74
8	145	1.58
16	151	1.52
32	169	1.36
64	170	1.35
128	160	1.44

N = 4 , Ideal speedup = 4, Average sequential time = 226ms.

#Tasks	Avg. Time taken (ms)	SpeedUp = Avg.Time taken / Avg. Sequential time
2	74	3.03
4	73	3.08
8	77	2.94
16	76	2.96
32	84	2.70
64	88	2.56
128	83	2.71

N = 6 , Ideal speedup = 6, Average sequential time = 229ms.

#Tasks	Avg. Time taken (ms)	SpeedUp = Avg.Time taken / Avg. Sequential time
2	64	3.54
4	53	4.27
8	53	4.28
16	53	4.24
32	54	4.20
64	56	4.07
128	55	4.22

N = 8 , Ideal speedup = 8, Average sequential time = 230ms.

#Tasks	Avg. Time taken (ms)	SpeedUp = Avg.Time taken / Avg. Sequential time
2	92	2.97
4	65	4.21
8	46	5.92
16	49	5.50
32	46	5.91
64	56	4.85
128	62	4.40

3. [5] Write a Java program to prove that Java threads share the heap.

Java threads provide an abstraction of concurrency and parallelism. They are managed by ExecutorService and Executor interface. Threads share heap data but have their own control and data stack. Hence they are lightweight.

In our example in **Shareheap.java**, we have two private classes inside Shareheap class to illustrate the example where the threads share heap data.

1. <u>SampleArray</u> class: This class is a simple abstraction of a Java integer array with only add and print methods. The constructor defines and allocates memory for given number of elements. It defaults to 10 elements if non-positive array size is asked to construct. Our add method (*addElement*()) adds an element into the array. If size gets full, we update and allocate the array with double of its original size. This method is synchronized as multiple threads would potentially try to add an element into this array. And we have a *print*() method to print the elements of the array.

2. <u>MyRunnable</u> class: This class implements the Runnable interface to create tasks. We provide the SampleArray object to the task while construction as a parameter and the task instance **points** to this local (or not?, is it pass by reference or value? We assume pass by value) copy of the SampleArray object. In the *run*() method we ask this implementation to add an element to it.

Now, our main() function creates an object (*sharedObject*) of class SampleArray of array size 10. We create 100 threads (*numThreads*) and provide a task to each of these threads to add a random element in this object *sharedObject*. All the threads are independent of each other. Also we have passed the sharedObject as a parameter to the task creation. We expect that a local copy would have been created in the task object.

Now we *run*() and *join*() the threads. As local copy would have been created in the thread and updated there locally, our *main*() function's *sharedObject* object should not be modified. But when we print it, there are exactly 100 elements in the array field.

1. Java objects are created in Heap. Stack only contains references to this heap. So our main function object *sharedObject* is actually inside the Java heap.
2. Java always uses pass-by-value. Even when we are passing object arguments, the reference to those objects (present in the stack) are passed-by-value. Hence if the function invoked has and object parameter passed to it and if it tries to modify it, the changes are reflected in the object and thus are seen in the original caller (here Main).

So in the MyRunnable task object, when we pass the sharedObject reference as value its instance variable *instance* stores the value of this reference and hence points to the original object (present in heap). Now we allocate this task to a thread and when the thread runs it modifies this same object present in the heap.

4. [10+10] For the code written in Q1(i) and Q1(ii): show the static Happens Before (HB) relation between the different Java statements. You would need to add a line number to each line in the Java code to illustrate the HB relation.
Note:

A) If two statements S1 and S2 may run in parallel with each other - they have no HB relation. Else, either S1 HB S2 and/or S2 HB S1.

B) While analyzing a program statically, unlike the actual execution, there will be cases where we may say that two statements S1 and S2, may have HB relation with each other in both directions (that is, S1 HB S2 and S2 HB S1).

Static happens before relationship

1. **DataRaces.java:**

a) Main(): It is executed in a single thread called main thread. We invoke certain methods from this function which create certain threads, run/execute them and after termination and some instructions return back to the caller main function. All the reads of the sum variables happen after successful termination of the threads getting created before. Hence they have a static Happen Before relation with the thread termination which is discussed below. Including these edges, all the statements inside the main function have static Happens Before relation among them consecutively (lines 190->..->216).

b) initializeArraySequential(): This method is similar to main function and runs in the same main thread. The for loop also executes sequentially. There is a static Happens Before relationship between every line. (lines 25->..->30)

c) calculateArraySumSequential(): Similar argument to the above function. As the *sequentialSum* is getting written sequentially in the *for* loop, its write is visible to the immediate next write. (So its like a self edge on that statement - not really sure how is it drawn/represented because self loop signifies reflexivity but Happens Before is irreflexive). The last write to the variable *sequentialSum* is read in the main function. There is a relation between 41 to 207,210,213. All the statements have consecutive statics Happen Before relation (lines 39-44).

d) executeTasks(): The pool invokes the tasks sequentially (similar to the above mentioned *for* loop) - hence a self edge. All other statements execute sequentially. Hence all the statements have consecutive statics Happen Before relation (lines 56-74).

e) calculateArraySumParallel(): The variable *parallelSum* is static. So it is stored in Heap (PermGen section). Now when threads are created they have reference to this static variable and they directly modify it in the heap using reference they have in their stack. We haven't guarded this static variable properly which leads to data race. As different threads manipulate it concurrently with no guard, we don't really have any static Happen Before relation among them. Finally all the threads terminate and the change in parallelSum variable is read later in Main. As we really don't know which is the last instruction (thread executed), we keep the await call as termination and accumulation of all the thread changes. Hence there is static Happen Before relation between 67 (94) to 209 and 210. All other statements have consecutive statics Happen Before relation (82->84->*for*->100->102).

f) calculateArraySumSynchronize(): Again we have static variable *synchronizeSum*. Also we have a guard on it using the synchronize method *addSynchronizedElement*(). So we don't have any data race. Also there is a static May Happen Before relation among all the threads because of this synchronized method. If one thread executes (writes to the variable *synchronizeSum*) it before others, the changes done by it will be visible to other threads. Hence self loop on line 109. Also the last write to the *synchronizeSum* variable will be visible to the read in main function read. Hence there is a static Happen Before relation between 67 (109) to 212 and 213. All other statements have consecutive statics Happen Before relation (118->120->*for*->136->138).

g) calculateArraySumModifiedSynchronize(): Very similar argument to the above function. The write to the task local variable *tempResult* is read in the synchronized method *addModifiedSynchronizedElement*(). Hence a relation on line 168->146. Then because of synchronized method we have self loop on line 146. Then due to read in main, we have a relation 67 (146) to 215 and 216. All other statements have consecutive statics Happen Before relation (155->157->*for*->175->177).

2. **Atomicity.java:**

a) main(): It is executed in a single thread called main thread. We invoke certain methods from this function which create certain threads, run/execute them and after termination and some instructions return back to the caller main function. All the reads of the sum variables happen after successful termination of the threads getting created before. Hence they have a static Happen Before relation with the thread termination which is discussed below. Including these edges, all the statements inside the main function have static Happens Before relation among them consecutively (lines 182->..->201).

b) setTransactions(): This method is similar to main function and runs in the same main thread. The for loop also executes sequentially. It also calls the *generateRandomTransaction*() method sequentially in each iteration. There is a static Happens Before relationship between every line. (lines 45-50, 49->35-37->49).

c) calculateFinalBalanceSequentially(): Similar argument to the above function. As the *sequentialBalance* is getting written sequentially in the *for* loop, its write is visible to the immediate next write. (So its like a self edge on that statement - not really sure how is it drawn/represented because self loop signifies reflexivity but Happens Before is irreflexive). The last write to the variable *sequentialBalance* is read in the main function. There is a relation between 61 to 195,198,201. All the statements have consecutive statics Happen Before relation (lines 59-64).

d) executeTasks(): The pool invokes the tasks sequentially (similar to the above mentioned *for* loop) - hence a self edge. All other statements execute sequentially. Hence all the statements have consecutive statics Happen Before relation (lines 74-92).

e) calculateFinalBalanceSynchronously(): The variable s*ynchronousBalance* is static. So it is stored in Heap (PermGen section). Now when threads are created they have reference to this static variable and they directly modify it in the heap using reference variable they have in their stack. We are reading this variable in the stack, modifying it locally in the stack and then updating it back in the Heap using local reference. Although we are updating the variable in a synchronized method not leading to data race, some other thread would have read and modified the same variable and the current update will shadow the changes done by it. Hence we have memory inconsistency as the update depends on scheduling of threads and we can have any update in any order. Still we do have a HB relation among the threads because of the synchronized method. Self loop on line 100 (126). The write to the task local variable *tempBalance* is read in the synchronized method *updateSynchronousBalance*(). Hence a relation on line 124->100. Finally all the threads terminate and the change in s*ynchronousBalance* variable is read later in *Main*. As we really don't know which is the last instruction (thread executed), we keep the await call as termination and accumulation of all the thread changes. Hence there is static Happen Before relation between 85(100) to 197 and 198. All other statements have consecutive statics Happen Before relation (111->113->*for*->131->133).

f) calculateFinalBalanceModifiedSynchronously(): Very similar argument for the HB relation from the above function. The write to the task local variable *addBalance* is read in the synchronized method *updateModifiedSynchronousBalance*(). Hence a relation on line 165->142. Then because of synchronized method we have self loop on line 142. Then due to read in main, we have a relation 85 (142) to 200 and 201. All other statements have consecutive statics Happen Before relation (152->154->*for*->172->174).

5. [10+10] Write a Java program that leads to a deadlock due to parallelism related constructs: (i) uses threads, and synchronized methods. (ii) uses threads and cyclic-barriers.

Deadlocks are the situation when two or more threads are simultaneously waiting for each other resulting in blocking of them and no progress.

i) Synchronized method are the methods which are executed only one at a time by a single thread. The lock used by this method is the locking provided by the object which the method is invoked from. Each object maintains a single lock for all the methods it has. So we cannot access the same method using the same object if we have multiple requests and they will have to wait one after another.

In the example **DiceDeadlock.java**, we have created only one such synchronized method per private internal class Athread and Bthread. These classes correspond to the players A and B who both has one dice each with them - die1 and die2 respectively.

The event is that both of them have to throw these dices and compute the sum and check how often these sum is equal. We have n=10 trials. As the die are private so to access them they have to call the throw method on the class object of each other. These throw methods are synchronized hence only one player can throw a die at a given instant. But they can throw different die at the same instant.

For e.g. A has die1, B has die2

For A to throw both die, it will call Athread.throw1(true) and similarly for B to throw both die it will call Bthread.throw2(true) . They will throw their die parallely and now to throw each others die A will call Bthread.throw2(false) and B will call Athread.throw1(false).

Here Athread and Bthread are (static) class object and are accessible outside the private class. We know that each object has only lock and hence to resolve multiple request on the same lock, it needs to be unlocked first.

Here thread1 runs and invokes Athread.throw1(true) and gets the lock of the class object Athread. Similarly, thread2 runs and invokes Bthread.throw2(true) and gets the lock of the class object Bthread. Now when both the locks are locked, thread1 invokes Bthread.throw2(false) and tries to get the lock of the class object Bthread. And similarly thread2 invokes Athread.throw1(false) and tries to get the lock of the class object Athread. This leads to DEADLOCK!! Because both the threads are waiting for each other to unlock the class object lock and none of them will release it unless it first gets access to other lock.

ii) Cyclic barriers provide an abstraction of a barrier and asks the threads to wait until a given number of threads are have not given signal to wait. It maintains a count which keeps incrementing as more threads invoke the *await*() function.When the required number of threads invoke this function, the barrier resolves itself and allows the threads to execute further and sets the count to 0 back again (cyclic).

In our example in **BarrierDeadlock.java**, we again have two friends A and B. They are given/allotted two die. They have to throw the die and report the sum they get later. We are interested to find how frequent do they get the same sum of both die they throw.

At any instant, each one of them will either throw same die which they were allotted or different die than allotted. But they cannot throw a same die because that is not possible. Hence to avoid this, we create two barriers in each event. The threads will wait on this barrier and then correspondingly either throw same allotted or different allotted die.

Our Runnable implemented CyclicBarrierRunnable task class provides this mechanism of two deadlocks and execute the throwing event in between them. The values of die1 and die2 are stored globally in a static variable. We have also created the optional task which runs at the barrier point indicating whether same die are going to be thrown or different die.

Now in our main, we create this two barriers task for each player (thread). Player A creates a barrier such that in the event first the players will throw same allotted die and then the different ones. And due to some miscommunication, player B creates a barrier such that in the event first the players will throw different allotted die and then the same ones.

This will result in player A (threadA) waiting at the barrier s*ameAllottedDiceBarrier* and player B (threadB) waiting at the barrier *differentAllottedDiceBarrier*. This leads to a deadlock as both of them are waiting for each other at a different barriers and no barrier can get resolved unless one of them resolves.