**Project part B**

Karim Safarini

7391560

**Introduction:**

This purpose of project is to ensure that two threads, namely ‘A’ and ‘B’, execute a sequence of related methods. The implementation guarantees that every call to one of the functions will only continue executing if its dependency has been computed by that point. In this context, the report describes the synchronization mechanism employed, provides the code references to the lines of code where the synchronization is performed, and as well includes the synch testing.

**Shared Data Structure:**

Data class containing the global variables (dependent variables A1, A2, A3, B1, B2, B3) and Synchronized flags (goFunA2, goFunA3, goFunB2, goFunB3)

**Thread Creation:**

In the main() function, instances of ThreadA and ThreadB are created and started for each iteration.

**Synchronization Using wait() and notify():**

Each function call is synchronized using the wait() and notify() methods to ensure proper execution.

ThreadA: A1 notifies B2, A2 waits for B2 to complete, and A3 waits for B3 to complete.

ThreadB: B2 waits for A1 to complete, and B3 waits for A2 to complete.

**High Number of Iterations and Verification:**

In the main() function, the threads are run for 1000 iterations to make sure that the implementation of the threads is correct and properly synchronized. At the end of each iteration A3 is checked to ensure that it is correct.

**Testing Results:**

In the testing, the index was tested by running the threads for 1000 cycles. Every time round the calculation, the value for A3 was checked against the correct value to ensure correctness. The comparative was made through the formula that calculates the sum of the four hundred natural numbers plus the content in B3.

**Outcome:**

It was also successful for all the 1000 iterations and the value of A3 obtained was as per the expected value.

The verbose in the success message “successful operation!!!!!” assured that the synchronization and the order of execution was properly designed and followed.

**Conclusion:**

The project was able to achieve synchronization to be used for two coordinating threads with the use of wait(). The common variable count was correct in the needed order where numerous iterations provided the forked program’s correct and reliable performance. This helps to make sure that what thread is supposed to do kind of function sequence is done in the right order with correct function calls dependent on the other function.

Code: **package** project;

**class** Data {

**int** A1, A2, A3, B1, B2, B3;

**boolean** goFunA2 = **false**;

**boolean** goFunA3 = **false**;

**boolean** goFunB2 = **false**;

**boolean** goFunB3 = **false**;

}

**public** **class** Project {

**public** **static** **void** main(String[] args) {

**int** testSize = 1000; // High number of iterations

Data mySample = **new** Data();

**for** (**int** i = 0; i < testSize; i++) {

System.***out***.println("This iteration " + i);

mySample.goFunA2 = **false**;

mySample.goFunA3 = **false**;

mySample.goFunB2 = **false**;

mySample.goFunB3 = **false**;

ThreadA ta = **new** ThreadA(mySample);

ThreadB tb = **new** ThreadB(mySample);

ta.start();

tb.start();

**try** {

ta.join();

tb.join();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

// Verify the correctness of the implementation

**int** expectedA3 = mySample.B3 + (400 \* (400 + 1) / 2);

**if** (mySample.A3 != expectedA3) {

System.***out***.println("erorr in iteration: " + i + " A3 is incorrect");

**break**;

}

}

System.***out***.println("successful operation!!!!!.-:)");

}

}

**class** ThreadA **extends** Thread {

**private** Data sample;

**public** ThreadA(Data sample) {

**super**();

**this**.sample = sample;

}

**public** **void** run() {

**synchronized** (sample) {

**int** n = 500;

sample.A1 = n \* (n + 1) / 2;

System.***out***.println("A1 finished " + sample.A1);

sample.goFunB2 = **true**;

sample.notify();

// may need break

}

**synchronized** (sample) {

**while** (sample.goFunA2 == **false**) {

**try** {

sample.wait();

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

System.***out***.println("Waiting in A2 for B2 to be completed");

// may need break

}

**synchronized** (sample) {

**int** n = 300;

sample.A2 = sample.B2 + (n \* (n + 1) / 2);

System.***out***.println("A2 finished " + sample.A2);

sample.goFunB3 = **true**;

sample.notify();

// may need break

}

**synchronized** (sample) {

**while** (sample.goFunA3 == **false**) {

**try** {

sample.wait();

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

System.***out***.println("Waiting in A3 for B3 to be completed");

// may need break

}

}

**synchronized** (sample) {

**int** n = 400;

sample.A3 = sample.B3 + (n \* (n + 1) / 2);

System.***out***.println("A3 finished " + sample.A3);

// may need break

}

}

}

}

**class** ThreadB **extends** Thread {

**private** Data sample;

**public** ThreadB(Data sample) {

**super**();

**this**.sample = sample;

}

**public** **void** run() {

**synchronized** (sample) {

**while** (sample.goFunB2 == **false**) {

**try** {

sample.wait();

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

System.***out***.println("Waiting in B2 for A1 to be completed");

// may need break

}

}

**synchronized** (sample) {

**int** n = 200;

sample.B2 = sample.A1 + (n \* (n + 1) / 2);

System.***out***.println("B2 finished " + sample.B2);

sample.goFunA2 = **true**;

sample.notify();

// may need break

}

**synchronized** (sample) {

**while** (sample.goFunB3 == **false**) {

**try** {

sample.wait();

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

System.***out***.println("Waiting in B3 for A2 to be completed");

// may need break

}

}

**synchronized** (sample) {

**int** n = 400;

sample.B3 = sample.A2 + (n \* (n + 1) / 2);

System.***out***.println("B3 finished " + sample.B3);

sample.goFunA3 = **true**;

sample.notify();

// may need break

}

}

}