1.

a) Non-atomic:

public class JavaAtomic {

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

ProcessingThread pt = new ProcessingThread();

Thread t1 = new Thread(pt, "t1");

t1.start();

Thread t2 = new Thread(pt, "t2");

t2.start();

t1.join();

t2.join();

System.out.println("Processing count=" + pt.getCount());

}

}

class ProcessingThread implements Runnable {

private int count;

@Override

public void run() {

for (int i = 1; i < 5; i++) {

processSomething(i);

count++;

}

}

public int getCount() {

return this.count;

}

private void processSomething(int i) {

// processing some job

try {

Thread.sleep(i \* 1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

If you will run above program, you will notice that count value varies between 5,6,7,8. The reason is because count++ is not an atomic operation. So by the time one threads read it’s value and increment it by one, other thread has read the older value leading to wrong result.

Atomic:

import java.util.concurrent.atomic.AtomicInteger;

public class JavaAtomic {

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

ProcessingThread pt = new ProcessingThread();

Thread t1 = new Thread(pt, "t1");

t1.start();

Thread t2 = new Thread(pt, "t2");

t2.start();

t1.join();

t2.join();

System.out.println("Processing count=" + pt.getCount());

}

}

class ProcessingThread implements Runnable {

private AtomicInteger count = new AtomicInteger();

@Override

public void run() {

for (int i = 1; i < 5; i++) {

processSomething(i);

count.incrementAndGet();

}

}

public int getCount() {

return this.count.get();

}

private void processSomething(int i) {

// processing some job

try {

Thread.sleep(i \* 1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

Benefits of using Concurrency classes for atomic operation is that we don’t need to worry about synchronization. This improves code readability and chance of errors are reduced. Also atomic operation concurrency classes are assumed to be more efficient that synchronization which involves locking resources.

b) The root cause is data race.

Any time two threads operate on a shared variable concurrently, and one of those operations performs a write, both threads must use atomic operations.

If you violate this rule, and either thread uses a non-atomic operation, you’ll have what the C++ standard refers to a data race, which results in torn reads and torn writes.

Example: the same as the non-atomic example in the question (a)

2.

pthread\_mutex\_t mutex1 = PTHREAD\_MUTEX\_INITIALIZER;

void functionC()

{

pthread\_mutex\_lock( &mutex1 );

       DO SOMETHING

pthread\_mutex\_unlock( &mutex1 );

}

3. The program has two child threads: t1 and t2. The function setDatatoReady notifies – using the condition variable condVar – that it is done with the preparation of the work condVar.notifyone(). While holding the lock, thread t2 is waiting for its notification: condVar.wait(lck);

In fact, the spurious wakeup can happen, that the receiver finished its task before the sender has sent its notification. The receiver is susceptible for spurious wakeups. So the receiver wakes up, although no notification happens.

5.

JAVA and C++ executes similarly. However, the result shows that C++ executes faster than JAVA. It may be caused by the operating mechanism. JAVA executes on JVM but C++ works directly on Windows. That may cause the difference of executing speed.