**Synchronization - Core Multi Threading in Java – 2021-2022**

**\*\* Situation \*\***

**Threa re two threads , there is only one class called "Common" which has only two synchronized menthods. The question is can two different threads access the different methods simultaneously ?**

There can be two scenarios

1. For Shared object

2. For Non-shared object

**For Shared object**

Let us consider for the shared object, it means the "Common" object has been shared by the two threads ie there is only one instance of the "Common" and it is used by the both the threads. Let us see what happens when two threads execute two different methods of the class "Common".

import java.util.concurrent.TimeUnit;

public class Common {

public synchronized void doTask1() {

try {

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

String s = String.format("%-10s %-10s - %10s %30s", "doTask1()",i,Thread.currentThread().getName(),System.nanoTime());

TimeUnit.SECONDS.sleep(1);

System.out.println(s);

}

}

catch (Exception e) {

e.printStackTrace();

}

}

public synchronized void doTask2() {

try {

for( int i = 0 ; i < 5 ; i++ )

{

String s = String.format("%-10s %-10s - %10s %30s", "doTask2()",i,Thread.currentThread().getName(),System.nanoTime());

TimeUnit.SECONDS.sleep(1);

System.out.println(s);

}

}

catch (Exception e) {

e.printStackTrace();

}

}

}

public class Thread1 implements Runnable {

private Common cmn;

public Thread1() {

}

public Thread1( Common cmn ) {

this.cmn = cmn;

}

@Override

public void run() {

cmn.doTask1();

}

}

public class Thread2 implements Runnable {

private Common cmn;

public Thread2() {

}

public Thread2( Common cmn ) {

this.cmn = cmn;

}

@Override

public void run() {

cmn.doTask2();

}

}

The test class is given below.

import java.util.concurrent.TimeUnit;

public class Test1 {

public static void doMultithreading1() {

**Common cmn = new Common();**

try {

long startTime = System.nanoTime();

**Thread th1 = new Thread( new Thread1(cmn) );**

th1.setName("T1");

**Thread th2 = new Thread( new Thread2(cmn) );**

th2.setName("T2");

th1.start();

th2.start();

boolean flag = true;

while( flag ) {

if( !th1.isAlive() && !th2.isAlive()) {

System.out.println("All threads are dead....");

flag = false;

long endTime = System.nanoTime();

long timeDiff = endTime - startTime;

System.out.println("Time Difference :::"+timeDiff+" nano seconds");

System.out.println("Time Difference :::"+TimeUnit.NANOSECONDS.toSeconds(timeDiff)+" seconds");

}

}

}

catch (Exception e) {

e.printStackTrace();

}

}

public static void main(String[] args) {

doMultithreading1();

}

}

If you run the above program many times, the average out will be like this.

RUNNABLE

doTask1() 0 - T1 10738597743678

doTask1() 1 - T1 10739605532732

doTask1() 2 - T1 10740620473032

doTask1() 3 - T1 10741621752298

doTask1() 4 - T1 10742623128005

doTask2() 0 - T2 10743624648779

doTask2() 1 - T2 10744625912646

doTask2() 2 - T2 10745640885363

doTask2() 3 - T2 10746641682422

doTask2() 4 - T2 10747655618192

All threads are dead....

Time Difference :::10076080852 nano seconds

Time Difference :::10 seconds

It means the first thread acquires the lock and until it release the lock, the other thread can not execute the other method because the concept is thread lock

per object. It is just like a sequential approach. Here it is also not required to use join() method of the thread. This is the only reason, it takes 10 seconds.

**For Non-shared object**

In this case let us modify the two threads like this.

public class Thread2 implements Runnable {

public Thread2() {

}

@Override

public void run() {

new Common().doTask2();

}

}

public class Thread1 implements Runnable {

public Thread1() {

}

@Override

public void run() {

new Common().doTask1();

}

}

The test class is given below.

import java.util.concurrent.TimeUnit;

public class Test1 {

public static void doMultithreading1() {

try {

long startTime = System.nanoTime();

Thread th1 = new Thread( new Thread1() );

th1.setName("T1");

Thread th2 = new Thread( new Thread2() );

th2.setName("T2");

th1.start();

th2.start();

boolean flag = true;

while( flag ) {

**if( !th1.isAlive() && !th2.isAlive())** {

System.out.println("All threads are dead....");

flag = false;

long endTime = System.nanoTime();

long timeDiff = endTime - startTime;

System.out.println("Time Difference :::"+timeDiff+" nano seconds");

System.out.println("Time Difference :::"+TimeUnit.NANOSECONDS.toSeconds(timeDiff)+" seconds");

}

}

}

catch (Exception e) {

e.printStackTrace();

}

}

public static void main(String[] args) {

doMultithreading1();

}

}

If you run the above program many times, the output may be like this.

doTask1() 0 - T1 11606743517968

doTask2() 0 - T2 11606743518778

doTask1() 1 - T1 11607761128233

doTask2() 1 - T2 11607762525822

doTask1() 2 - T1 11608775943321

doTask2() 2 - T2 11608776883016

All threads are dead...., Time Difference :::5081768864 nano seconds, Time Difference :::5 seconds

If you observe you will find that the time difference is 5 seconds, it means both the threads are able to execute their respective methods. It means for each thread we are creating a new instance of "Common", it is a new object for that particular thread. Since that object is not acquired by other thread, that particular thread will be able to execute the method.

**Now it creates another question.**

**\*\* Situation \*\***

There is one common class which has one synchronized method and one non-synchronized method.

Can two different threads can access both the methods simulataniously , both in case Shared and Non-shared object. Here the answer is yes in case of both shared and non-shared object. Let us see the code below.

import java.util.concurrent.TimeUnit;

public class Common {

public synchronized void doTask1() {

try {

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

String s = String.format("%-10s %-10s - %10s %30s", "doTask1()",i,Thread.currentThread().getName(),System.nanoTime());

TimeUnit.SECONDS.sleep(1);

System.out.println(s);

}

}

catch (Exception e) {

e.printStackTrace();

}

}

public void doTask2() {

try {

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

String s = String.format("%-10s %-10s - %10s %30s", "doTask2()",i,Thread.currentThread().getName(),System.nanoTime());

TimeUnit.SECONDS.sleep(1);

System.out.println(s);

}

}

catch (Exception e) {

e.printStackTrace();

}

}

}

The test class is given below.

public class Thread2 implements Runnable {

private Common cmn;

public Thread2( Common cmn ) {

this.cmn = cmn;

}

@Override

public void run() {

cmn.doTask2();

}

}

public class Thread1 implements Runnable {

private Common cmn;

public Thread1( Common cmn ) {

this.cmn = cmn;

}

@Override

public void run() {

cmn.doTask1();

}

}

import java.util.concurrent.TimeUnit;

public class Test1 {

public static void doMultithreading1() {

Common cmn = new Common();

try {

long startTime = System.nanoTime();

Thread th1 = new Thread( new Thread1(cmn) );

th1.setName("T1");

Thread th2 = new Thread( new Thread2(cmn) );

th2.setName("T2");

//Always write join() method after the start() method of a thread

th1.start();

th2.start();

boolean flag = true;

while( flag ) {

if( !th1.isAlive() && !th2.isAlive()) {

System.out.println("All threads are dead....");

flag = false;

long endTime = System.nanoTime();

long timeDiff = endTime - startTime;

System.out.println("Time Difference :::"+timeDiff+" nano seconds");

System.out.println("Time Difference :::"+TimeUnit.NANOSECONDS.toSeconds(timeDiff)+" seconds");

}

}

}

catch (Exception e) {

e.printStackTrace();

}

}

public static void main(String[] args) {

doMultithreading1();

}

}

So it is clear that in case of **sysnchronized method, the guard is for the whole method not for whole class**. In case of Java interview question, the interviewer may ask the question that in case of synchronized method, it has acquired the object lock, can another method access a non-synchronized method of that class. **You should give answer as yes because** one thread has acquired the lock to perform some operation for that particular method and not for the whole class.

**\*\* Situation \*\***

Can one thread access one synchronized/non-sysnchronized method from another sysnchronized method of the same class ? The answer is yes. But there is a trick and it is a mistake also , people do many times for the purpose of optimization. Suppose we have a method and that method is not synchronized but some portion of the method is synchronized for optimization purpose and after

synchronize block if we invoke another synchronized method of the same class, there is no gurrantee that the method will be invoked by the same thread. The method structure is given below.

import java.util.concurrent.TimeUnit;

public class Common {

public void doTask1() {

synchronized(this) {

String s1 = String.format("%-20s %-10s %-20s %-30s", "Method name",

"Counter","Thread Name","Time in nanoseconds");

System.out.println(s1);

try {

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

String s = String.format("%-20s %-10s %-20s %-30s", "doTask1()",i,Thread.currentThread().getName(),System.nanoTime());

TimeUnit.SECONDS.sleep(1);

System.out.println(s);

}

}

catch (Exception e) {

e.printStackTrace();

}

}

**doTask2(); // Mark this line**

}

public synchronized void doTask2() {

try {

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

String s = String.format("%-20s %-10s %-20s %-30s", "doTask2()",i,Thread.currentThread().getName(),System.nanoTime());

TimeUnit.SECONDS.sleep(1);

System.out.println(s);

}

}

catch (Exception e) {

e.printStackTrace();

}

}

}

In the above method we are invoking another method called "doTask2()". In this case the first thread completes its operation till the end of the synchronized block and it may release the lock, the moment it releases the lock another thread may execute the another synchronized method of that class. Let us see the complete code structure and run it many times, you will realize the issue.

public class Thread2 implements Runnable {

private Common cmn;

public Thread2( Common cmn ) {

this.cmn = cmn;

}

@Override

public void run() {

cmn.doTask2();

}

}

public class Thread1 implements Runnable {

private Common cmn;

public Thread1( Common cmn ) {

this.cmn = cmn;

}

@Override

public void run() {

cmn.doTask1();

}

}

let us see the test class.

import java.util.concurrent.TimeUnit;

public class Test1 {

public static void doMultithreading1() {

Common cmn = new Common();

try {

long startTime = System.nanoTime();

Thread th1 = new Thread( new Thread1(cmn) );

th1.setName("T1");

Thread th2 = new Thread( new Thread2(cmn) );

th2.setName("T2");

th1.start();

th2.start();

boolean flag = true;

while( flag ) {

if( !th1.isAlive() && !th2.isAlive()) {

System.out.println("All threads are dead....");

flag = false;

long endTime = System.nanoTime();

long timeDiff = endTime - startTime;

System.out.println("Time Difference :::"+timeDiff+" nano seconds");

System.out.println("Time Difference :::"+TimeUnit.NANOSECONDS.toSeconds(timeDiff)+" seconds");

}

}

}

catch (Exception e) {

e.printStackTrace();

}

}

public static void main(String[] args) {

doMultithreading1();

}

}

If you run the above program, the output may be like the below case.

Method name Counter Thread Name Time in nanoseconds

doTask1() 0 T1 14640811164662

doTask1() 1 T1 14641798541385

doTask1() 2 T1 14642804192928

doTask1() 3 T1 14643804257357

doTask1() 4 T1 14644804830737

doTask2() 0 T2 14645819226021

doTask2() 1 T2 14646818909953

doTask2() 2 T2 14647819476040

doTask2() 3 T2 14648833500957

doTask2() 4 T2 14649833566602

doTask2() 0 T1 14650834163079

doTask2() 1 T1 14651848291327

doTask2() 2 T1 14652848209473

doTask2() 3 T1 14653848830264

doTask2() 4 T1 14654862872605

All threads are dead....

Time Difference :::15059518879 nano seconds

Time Difference :::15 seconds

**Always remember that using volatile variable, a thread can see the latest value at any point of time.**

**Thread Interruption**

According to JLS

**First the checkAccess method of this thread is invoked, which may cause a SecurityException to be thrown.**

**If this thread is blocked in an invocation of the wait(), wait(long), or wait(long, int) methods of the Object class, or of the join(), join(long), join(long, int), sleep(long), or sleep(long, int), methods of this class, then its interrupt status will be cleared and it will receive an InterruptedException.**

**If this thread is blocked in an I/O operation upon an interruptible channel then the channel will be closed, the thread's interrupt status will be set, and the thread will receive a ClosedByInterruptException.**

If this thread is blocked in a Selector then the thread's interrupt status will be set and it will return immediately from the selection operation, possibly with a non-zero value, just as if the selector's wakeup method were invoked.

If none of the previous conditions hold then this thread's interrupt status will be set.

**Interrupts**

**An interrupt is an indication to a thread that it should stop what it is doing and do something else.** It's up to the programmer to decide exactly how a thread responds to an interrupt, but it is very common for the thread to terminate. This is the usage emphasized in this lesson. A thread sends an interrupt by invoking interrupt on the Thread object for the thread to be interrupted. For the interrupt mechanism to work correctly, the interrupted thread must support its own interruption.

Supporting Interruption

How does a thread support its own interruption? This depends on what it's currently doing. If the thread is frequently invoking methods that throw InterruptedException, it simply returns from the run method after it catches that exception. For example, suppose the central message loop in the SleepMessages example were in the run method of a thread's Runnable object. Then it might be modified as follows to support interrupts:

for (int i = 0; i < importantInfo.length; i++) {

// Pause for 4 seconds

try {

Thread.sleep(4000);

} catch (InterruptedException e) {

// We've been interrupted: no more messages.

return;

}

// Print a message

System.out.println(importantInfo[i]);

}

**Many methods that throw InterruptedException, such as sleep, are designed to cancel their current operation and return immediately when an interrupt is received.**

What if a thread goes a long time without invoking a method that throws InterruptedException? Then it must periodically invoke Thread.interrupted, which returns true if an interrupt has been received. For example:

for (int i = 0; i < inputs.length; i++) {

heavyCrunch(inputs[i]);

if (Thread.interrupted()) {

// We've been interrupted: no more crunching.

return;

}

}

In this simple example, the code simply tests for the interrupt and exits the thread if one has been received. In more complex applications, it might make more sense to throw an InterruptedException:

if (Thread.interrupted()) {

throw new InterruptedException();

}

This allows interrupt handling code to be centralized in a catch clause.

The interrupt mechanism is implemented using an internal flag known as the interrupt status. Invoking Thread.interrupt sets this flag. When a thread checks for an interrupt by invoking the static method Thread.interrupted, interrupt status is cleared. The non-static isInterrupted method, which is used by one thread to query the interrupt status of another, does not change the interrupt status flag.

By convention, any method that exits by throwing an InterruptedException clears interrupt status when it does so. However, it's always possible that interrupt status will immediately be set again, by another thread invoking interrupt.

**Examples on Thread Interruption**

Let us consider a simple and easiest example on thread interruption. We will create a thread which will print 1 - 100 and if the value is 37, the thread should terminate. To see the output, we have provided a sleep method so that we can see the output for close observation.

Let us see the outcome from the following code.

import java.util.concurrent.TimeUnit;

public class Thread1 extends Thread {

@Override

public void run() {

String header = String.format("%-30s %-20s", "Thread Name","Counter");

System.out.println(header);

for( int i = 0 ; i < 100 ; i++ ) {

try {

Thread currentThread = Thread.currentThread();

String msg = String.format("%-30s %-20s", currentThread.getName(),i);

System.out.println(msg);

if( i == 37 ) {

System.out.println("\nNow the value is received and the loop should be stopped...");

currentThread.interrupt();

}

TimeUnit.MILLISECONDS.sleep(50);

}

catch( InterruptedException ie ) {

ie.printStackTrace();

//The interrupt status flag is cleared, so it will continue

System.out.println("Thread Interrupt status --->"+Thread.currentThread().isInterrupted());

}

catch (Exception e) {

System.out.println("Other Exception ....");

e.printStackTrace();

}

}

}

}

The test program is given below.

public class TestThread1 {

public static void main(String[] args) {

Thread th1 = new Thread1();

th1.setName("Thread-1");

th1.start();

}

}

If you run the above program, you will be surprised that the thread will not be stopped after the number 37 and it will throw exception and continue to print. So where is the problem, the problem is due to sleep() method and since it throws exception, the interruption status flag has been cleared of. You have to remember that you have to write your own logic for the thread interruption policy.

Let us see the modified program.

import java.util.concurrent.TimeUnit;

public class Thread1 extends Thread {

@Override

public void run() {

String header = String.format("%-30s %-20s", "Thread Name","Counter");

System.out.println(header);

for( int i = 0 ; i < 100 ; i++ ) {

try {

Thread currentThread = Thread.currentThread();

String msg = String.format("%-30s %-20s", currentThread.getName(),i);

System.out.println(msg);

if( i == 37 ) {

System.out.println("\nNow the value is received and the loop should be stopped...");

currentThread.interrupt();

}

TimeUnit.MILLISECONDS.sleep(50);

}

catch( InterruptedException ie ) {

ie.printStackTrace();

//The interrupt status flag is cleared, so it will continue

System.out.println("Thread Interrupt status --->"+Thread.currentThread().isInterrupted());

**Thread.currentThread().interrupt();**

System.out.println("Thread Interrupt status --->"+Thread.currentThread().isInterrupted());

**break;//This is my interruption policy**

// return;//This is my interruption policy

}

catch (Exception e) {

System.out.println("Other Exception ....");

e.printStackTrace();

}

}

}

}

This is the better program as we have provided our thread interruption policy and thread will terminate.

If you look into the code, in the exception block I am again interrupting thread because in case of exception, the thread status has been cleared of and to

set once again, again I am going to interrupt the thread. Fine , where is my interruption policy ?

The following line is my interruption policy. **break;**

Interruption policy means to logically terminate the executing method or related artifacts. However it is the simplest program as our entire logic is inside the run() method of the thread. Now let us take an example of recursion in case of thread interruption. This is one of the difficult situations developers normally face to interrupt the thread.

Let us conside the program below.

import java.util.concurrent.TimeUnit;

public class Thread2 extends Thread {

@Override

public void run() {

recursiveIncrement(0);

}

public void recursiveIncrement( int num ) {

try {

Thread currentThread = Thread.currentThread();

num++;

if( num >= 100 ) return;

else{

String msg = String.format("%-30s %-20s", currentThread.getName(),num);

System.out.println(msg);

if( num == 37 ) {

System.out.println("\nNow the value is received and the loop should be stopped...");

currentThread.interrupt();

}

TimeUnit.MILLISECONDS.sleep(50);

recursiveIncrement( num );

}

}

catch( InterruptedException ie ){

ie.printStackTrace();

//The interrupt status flag is cleared, so it will continue

num = 110;

System.out.println("Now Num value---->"+num);

System.out.println("Thread Interrupt status --->"+Thread.currentThread().isInterrupted());

Thread.currentThread().interrupt();

System.out.println("Thread Interrupt status --->"+Thread.currentThread().isInterrupted());

**return;**

}

catch (Exception e) {

System.out.println("Other Exception ....");

e.printStackTrace();

}

}

}

The test program is given below.

public class TestThread1 {

public static void main(String[] args) {

Thread th2 = new Thread2();

th2.setName("Thread-2");

th2.start();

}

}

If you run the above program, you may get the result, but the thread will not be interrupted. It will continue to throw exception and after that it will stop. If you really terminate the thread, use your own thread termination policy ie use "return" after the logical condition.

**So the modified program is given below.**

import java.util.concurrent.TimeUnit;

public class Thread2 extends Thread {

@Override

public void run() {

recursiveIncrement(0);

}

public void recursiveIncrement( int num )

{

try {

Thread currentThread = Thread.currentThread();

num++;

if( num >= 100 ) return;

else{

String msg = String.format("%-30s %-20s", currentThread.getName(),num);

System.out.println(msg);

if( num == 37 ) {

System.out.println("\nNow the value is received and the loop should be stopped...");

**currentThread.interrupt();**

**return;//This is my interruption policy**

}

TimeUnit.MILLISECONDS.sleep(50);

recursiveIncrement( num );

}

}

catch( InterruptedException ie ){

ie.printStackTrace();

//The interrupt status flag is cleared, so it will continue

num = 110;

System.out.println("Now Num value---->"+num);

System.out.println("Thread Interrupt status --->"+Thread.currentThread().isInterrupted());

Thread.currentThread().interrupt();

System.out.println("Thread Interrupt status --->"+Thread.currentThread().isInterrupted());

return;

}

catch (Exception e) {

System.out.println("Other Exception ....");

e.printStackTrace();

}

}

}

**What about If I do not want to interrupt the thread. Let us see we want lauch a missile, once it starts it should not be interrupted, even if somebody is trying to interrupt the thread. In this case we have to extend the Thread class and we want to override the thread interrupt() method. Let us see the code below.**

import java.util.concurrent.TimeUnit;

public class MissileLaunchingThread **extends Thread** {

@Override

public void run() {

String header = String.format("%-30s %-20s", "Thread Name","Counter");

System.out.println(header);

for( int i = 0 ; i < 100 ; i++ ) {

try {

Thread currentThread = Thread.currentThread();

String msg = String.format("%-30s %-20s", currentThread.getName(),i);

System.out.println(msg);

if( i == 87 ) {

System.out.println("\nNow the value is received and the loop should be stopped...");

currentThread.interrupt();

System.out.println("Thread has been interrupted...");

System.out.println("Can I be stopped ?????????");

System.out.println("No way BOSS... I am going to be fired....");

}

TimeUnit.MILLISECONDS.sleep(10);

}

catch( InterruptedException ie ) {

ie.printStackTrace();

//The interrupt status flag is cleared, so it will continue

System.out.println("Thread Interrupt status --->"+Thread.currentThread().isInterrupted());

Thread.currentThread().interrupt();

System.out.println("Thread Interrupt status --->"+Thread.currentThread().isInterrupted());

break;//This is my interruption policy

// return;//This is my interruption policy

}

catch (Exception e) {

System.out.println("Other Exception ....");

e.printStackTrace();

}

}

}

**@Override**

**public void interrupt() {**

**System.out.println("\*\*\*\*\*\*\*\*\*\* I WILL NOT BE STOPPED OR INTERRUPTED \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");**

**System.out.println("\*\*\*\*\*\*\*\*\*\* I WILL NOT BE STOPPED OR INTERRUPTED \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");**

**System.out.println("\*\*\*\*\*\*\*\*\*\* I WILL NOT BE STOPPED OR INTERRUPTED \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");**

**}**

}

The test program is given below.

public class TestMissileLaucnhingThread {

public static void main(String[] args) {

Thread th1 = new MissileLaunchingThread();

th1.setName("Missile");

th1.start();

}

}

**This is the only condition where we need to extend the Thread class instead of implementing Runnable interface.**

Normally a typical question is asked to the developers that in which situation we have to go for Thread extension and Runnable implemenation. So the final answer will if we want to change the behaviour of the thread by overriding the thread methods, we have to extend the thread class. In normal situation we have to go for implementing Runnable interface.

**What will happen if there are two threads, one thread has to be terminated after a certain logical condition and other thread has to continue.** Let us see the code below.

import java.util.concurrent.TimeUnit;

public class MyUtil {

public synchronized void execute() {

recursiveIncrement(0);

}

public void recursiveIncrement( int num ) {

try {

Thread currentThread = Thread.currentThread();

num++;

if( num >= 100 ) return;

else{

String msg = String.format("%-30s %-20s", currentThread.getName(),num);

System.out.println(msg);

**if( num == 37 && currentThread.getName().equals("Thread-1"))** {

System.out.println("\nNow the value is received and the loop should be stopped...");

**currentThread.interrupt();**

**return;//This is my interruption policy**

}

TimeUnit.MILLISECONDS.sleep(50);

recursiveIncrement( num );

}

}

catch( InterruptedException ie ){

ie.printStackTrace();

//The interrupt status flag is cleared, so it will continue

num = 110;

System.out.println("Now Num value---->"+num);

System.out.println("Thread Interrupt status --->"+Thread.currentThread().isInterrupted());

**Thread.currentThread().interrupt();**

**System.out.println("Thread Interrupt status --->"+Thread.currentThread().isInterrupted());**

**return;**

}

catch (Exception e) {

System.out.println("Other Exception ....");

e.printStackTrace();

}

}

public class Thread3 implements Runnable {

private MyUtil util;

public Thread3( MyUtil util ) {

this.util = util;

}

@Override

public void run() {

util.execute();

}

}

public class Thread4 implements Runnable {

private MyUtil util;

public Thread4( MyUtil util ) {

this.util = util;

}

@Override

public void run() {

util.execute();

}

}

}

public class Test1 {

public static void main(String[] args) {

MyUtil util = new MyUtil();

Thread th3 = new Thread( new Thread3(util));

th3.setName("Thread-1");

Thread th4 = new Thread( new Thread4(util));

th3.setName("Thread-2");

th3.start();

th4.start();

}

}

**Starvation, LiveLock and** **DeadLock in Java**

**Starvation**

*Starvation* describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress. This happens when shared resources are made unavailable for long periods by "greedy" threads. For example, suppose an object provides a synchronized method that often takes a long time to return. If one thread invokes this method frequently, other threads that also need frequent synchronized access to the same object will often be blocked. ***Starvation describes a situation where a greedy thread holds a resource for a long time so other threads are blocked forever***.

## **Worker.java**

**import** java.io.BufferedWriter;  
**import** java.io.FileWriter;  
**import** java.io.IOException;  
**public class Worker** {  
 **public synchronized void** work() {  
 String name = Thread.*currentThread*().getName();  
 String fileName = name + **".txt"**;  
  
 **try** (BufferedWriter writer = **new** BufferedWriter(**new** FileWriter(fileName));) {  
 writer.write(**"Thread "** + name + **" wrote this mesasge"**);  
 } **catch** (IOException ex) {  
 ex.printStackTrace();  
 }  
  
 **while** (**true**) {  
 System.***out***.println(name + **" is working"**);  
 }  
 }  
}

## **StarvationExample.java** **public class** StarvationExample { **public static void** main(String[] args) { Worker worker = **new** Worker(); **for** (**int** i = 0; i < 10; i++) { **new** Thread(**new** Runnable() { **public void** run() { worker.work(); } }).start(); } } }

A solution to solve this starvation problem is to make the current thread waits for a specified amount of time so other threads have chance to acquire the lock on the Worker object:

while (true) {

    System.out.println(name + " is working");

    try {

**wait(1000);**

    } catch (InterruptedException ex) {

        ex.printStackTrace();

    }

}

**Livelock**

A thread often acts in response to the action of another thread. If the other thread's action is also a response to the action of another thread, then *livelock* may result. As with deadlock, livelocked threads are unable to make further progress. However, the threads are not blocked — they are simply too busy responding to each other to resume work. This is comparable to two people attempting to pass each other in a corridor: Alphonse moves to his left to let Gaston pass, while Gaston moves to his right to let Alphonse pass. Seeing that they are still blocking each other, Alphone moves to his right, while Gaston moves to his left. They're still blocking each other, so... Livelock conditions can arise when two or more tasks depend on and use the some resource causing a circular dependency condition where those tasks continue running forever. **One example will be both wife and husband want eat soup using one spoon, they say, you first and you first**.

***Livelock describes situation where two threads are busy responding to actions of each other***. They keep repeating a particular code so the program is unable to make further progress:

Thread 1 acts as a response to action of thread 2

Thread 2 acts as a response to action of thread 1

Unlike deadlock, threads are not blocked when livelock occurs. They are simply too busy responding to each other to resume work. In other words, the program runs into an infinite loop and cannot proceed further.

**A Livelock Example:**

Let’s see an example: a criminal kidnaps a hostage and he asks for ransom in order to release the hostage. A police agrees to give the criminal the money he wants once the hostage is released. The criminal releases the hostage only when he gets the money. Both are waiting for each other to act first, hence livelock.

## **Criminal.java**

**public class** Criminal {  
 **private boolean hostageReleased** = **false**;  
 **public void** releaseHostage(Police police) {  
 **while** (!police.isMoneySent()) {  
 System.***out***.println(**"Criminal: waiting police to give ransom"**);  
 **try** {  
 Thread.*sleep*(1000);  
 } **catch** (InterruptedException ex) { ex.printStackTrace(); }  
 }  
 System.***out***.println(**"Criminal: released hostage"**);  
 **this**.**hostageReleased** = **true**;  
 }  
  
 **public boolean** isHostageReleased() {  
 **return this**.**hostageReleased**;  
 }  
}

## **Police.java**

**public class** Police {  
 **private boolean moneySent** = **false**;  
 **public void** giveRansom(Criminal criminal) {  
 **while (!criminal.isHostageReleased())** {  
 System.***out***.println(**"Police: waiting criminal to release hostage"**);  
 **try** {  
 Thread.*sleep*(1000);  
 } **catch** (InterruptedException ex) {  
 ex.printStackTrace();  
 }  
 }  
 System.***out***.println(**"Police: sent money"**);  
 **this**.**moneySent** = **true**;  
 }  
  
 **public boolean** isMoneySent() {  
 **return this**.**moneySent**;  
 }  
}

## **HostageRescueLiveLock.java**

**public class** HostageRescueLivelock {  
 **static final** Police ***police*** = **new** Police();  
 **static final** Criminal ***criminal*** = **new** Criminal();  
 **public static void** main(String[] args) {  
 Thread t1 = **new** Thread(**new** Runnable() {  
 **public void** run() {  
 ***police***.giveRansom(***criminal***);  
 }  
 });  
 t1.start();  
  
 Thread t2 = **new** Thread(**new** Runnable() {  
 **public void** run() {  
 ***criminal***.releaseHostage(***police***);  
 }  
 });  
 t2.start();  
 }  
}

**DEADLOCK** Deadlock is a condition in which a task waits indefinitely for conditions that can never be satisfied - task claims exclusive control over shared resources - task holds resources while waiting for other resources to be released - tasks cannot be forced to relinguish resources - a circular waiting condition exists.

# Understanding Deadlock

***Deadlock describes a situation where two more threads are blocked because of waiting for each other forever***. When deadlock occurs, the program hangs forever and the only thing you can do is to kill the program.

## **BusinessTest1.java**

**public class** BusinessTest1 {  
 **public static void** main(String[] args) {  
 **final** Business business = **new** Business();  
  
 Thread t1 = **new** Thread(**new** Runnable() {  
 **public void** run() {  
 business.foo();  
 }  
 });  
 t1.start();  
  
 Thread t2 = **new** Thread(**new** Runnable() {  
 **public void** run() {  
 business.bar();  
 }  
 });  
 t2.start();  
 }  
}

## **Business.java** **public class** Business { **private** Object **lock1** = **new** Object(); **private** Object **lock2** = **new** Object(); **public void** foo() { **synchronized** (**lock1**) { **synchronized** (**lock2**) { System.***out***.println(**"foo"**); } } } **public void** bar() { **synchronized** (**lock2**) { **synchronized** (**lock1**) { System.***out***.println(**"bar"**); } } } }

**How Avoid deadlock**

public void bar() {

    synchronized (lock1) {

        synchronized (lock2) {

            System.out.println("bar");

        }

    }

}

public void foo() {

    synchronized (lock1) {

        synchronized (lock2) {

            System.out.println("foo");

        }

    }

}

**Notes**

- **Deadlock**:  All threads are blocked, the program hangs forever.

- **Livelock**: No threads blocked but they run into infinite loops. The program is still running but unable to make further progress.

- **Starvation**: Only one thread is running, and other threads are waiting forever.

**DeadLock**

**Simplest Type - 1**

**Use join() in the run method of both the threads.**

public class Thread2 implements Runnable {

@Override

public void run() {

System.out.println("Running thread 2 ....");

try {

Thread.currentThread().join();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

public class Thread1 implements Runnable {

@Override

public void run() {

System.out.println("Running thread 1 ....");

try {

Thread.currentThread().join();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

public class TestDeadLock1 {

public static void main(String[] args) {

Thread th1 = new Thread( new Thread1());

Thread th2 = new Thread( new Thread2());

th1.start();

th2.start();

}

}

**Type – 2: Using shared resources**

public class Thread1 extends Thread {

private String resource1;

private String resource2;

public Thread1(String resource1, String resource2) {

this.resource1 = resource1;

this.resource2 = resource2;

}

@Override

public void run() {

**synchronized (resource1)** {

System.out.println("Thread 1: locked resource 1");

try {

Thread.sleep(100);

} catch (Exception e) {

e.printStackTrace();

}

**synchronized (resource2)** {

System.out.println("Thread 1: locked resource 2");

}

}

}

}

public class Thread2 extends Thread {

private String resource1;

private String resource2;

public Thread2( String resource1 , String resource2 ) {

this.resource1 = resource1;

this.resource2 = resource2;

}

@Override

public void run() {

**synchronized (resource2) {**

System.out.println("Thread 1: locked resource 2");

try {

Thread.sleep(100);

} catch (Exception e) {

e.printStackTrace();

}

**synchronized (resource1)** {

System.out.println("Thread 1: locked resource 1");

}

}

}

}

public class TestDeadLock {

public static void main(String[] args) {

String resource1 = "resource1";

String resource2 = "resource2";

Thread th1 = new Thread1(resource1,resource2);

Thread th2 = new Thread2(resource1,resource2);

th1.start();

th2.start();

}

}

[How does JVM handle synchronization?](http://javawithneo.blogspot.com/2011/08/how-does-jvm-handle-synchronization.html)

**Introduction**  
We know that each thread has its own stack and that all the threads in the program share the heap. The heap contains all the objects created within the program including the thread. The method area contains all the class /static variables of a classes used by the program. These variables are available to all the threads in the program.  
  
Now we know that we have two data areas which contains data shared by all threads.  
1. the heap and 2. the method area  
  
**Monitors**  
So if two threads try to access the objects or class variables in these areas concurrently, then the data need to be properly managed else we will end up with inconsistent data. This situation can be handled through synchronization and java manages this through the use of '*monitor'*. Hence monitor acts as a guardian over a piece of code, so that no two threads execute that code simultaneously.  
Java's monitor supports two kinds of synchronization: *mutual exclusion*and *cooperation*

* **Mutual exclusion** is achieved in JVM through the use of object or class blocks , they enable multiple threads to work independently on shared data without interfering with  each other.
* While **Cooperation** in JVM is achieved through the use of object wait, notify and notifyAll methods.

Each monitor is associated with an object reference. Whenever a thread reaches the code which is synchronized, it must obtain a lock on the referenced object failing which it will have to wait (block on synchronization state). Once the thread obtains the lock the JVM increases the count of the number of times an object has been locked. The same thread can lock the same object multiple times. Whenever the thread releases/relinquishes the lock the count is decremented. When there are no more locks on the object i.e count returns to zero, then any other thread can obtain the lock on the object if needed.  
  
Synchronization is supported by the java language  in two ways

1. **synchronized method :**Whenever the JVM encounters a symbolic reference to the method and realizes its a synchronized method, then it tries to obtains the lock from the monitor. If the lock is obtained then the synchronized method is executed and once all the statements are processed or the code throws an exception the lock is released. For an instance method, a lock is obtained on the object, of which synchronized method is invoked. In case of static synchronized method the lock is obtained on the class object. The JVM does not use special op-codes to invoke or return from method level synchronization.
2. **synchronized statement (Block of code):**The JVM uses two special op codes **monitorenter** and **monitorexit** whenever a thread enters or exits the synchronized block of code. When the JVM's encounters *monitorenter*  it tries to obtain a lock on the object referred to by objectref on the stack. If the lock is already obtained the the count is incremented by one and whenever the *monitorexit*  is encountered the count is decremented by one. When the count reaches zero the lock is released.

**How JVM handles thread synchronization?**

JVM associates a lock with an object or a class to achieve mutilthreading. A lock is like a token or privilege that only one thread can "possess" at any one time. When a thread wants to lock a particular object or class, it asks the JVM.JVM responds to thread with a lock maybe very soon, maybe later, or never. When the thread no longer needs the lock, it returns it to the JVM. If another thread has requested the same lock, the JVM passes the lock to that thread.If a thread has a lock,no other thread can access the locked data until the thread that owns the lock releases it.

The JVM uses locks in conjunction with monitors. A monitor is basically a guardian in that it watches over a sequence of code, making sure only one thread at a time executes the code.Each monitor is associated with an object reference. It is the responsibility of monitor to watch an arriving thread must obtain a lock on the referenced object.

When the thread leaves the block,it releases the lock on the associated object.A single thread is allowed to lock the same object multiple times.JVM maintains a count of the number of times the object has been locked. An unlocked object has a count of zero. When a thread acquires the lock for the first time, the count is incremented to one. Each time the thread acquires a lock on the same object, a count is incremented. Each time the thread releases the lock, the count is decremented. When the count reaches zero, the lock is released and made available to other threads.

In Java language terminology, the coordination of multiple threads that must access shared data is called synchronization. The language provides two built-in ways to synchronize access to data: with synchronized statements or synchronized methods. The JVM does not use any special opcodes to invoke or return from synchronized methods. When the JVM resolves the symbolic reference to a method, it determines whether the method is synchronized. If it is, the JVM acquires a lock before invoking the method. For an instance method, the JVM acquires the lock associated with the object upon which the method is being invoked. For a class method, it acquires the lock associated with the class to which the method belongs. After a synchronized method completes, whether it completes by returning or by throwing an exception, the lock is released.

**Two opcodes, monitorenter and monitorexit are used by JVM for accomplishing this task.**

When monitorenter is encountered by the Java virtual machine, it acquires the lock for the object referred to by objectref on the stack. If the thread already owns the lock for that object, a count is incremented. Each time monitorexit is executed for the thread on the object, the count is decremented. When the count reaches zero, the monitor is released.

**Difference between Hashtable-ConcurrentHashMap-Collections.synchronizedMap(Map)**

The one feature offered by the synchronized Map or HashTable implementations but not by ConcurrentHashMap is the ability to lock the map for exclusive access. With Hashtable and synchronizedMap, acquiring the Map lock prevents any other thread from accessing it. This might be necessary in unusual cases such as adding several mappings atomically, or iterating the

Map several times and needing to see the same elements in the same order. Where as concurrent collections should be expected to change their contents continuously. This is actually a very good point. One common use case is to grab the lock (on the instance of the collection) prior to iterating through the collection; another is to do the same when multiple operations on the collection is needed; this works because the synchronized Map and/or HashTable uses the same lock for its implementation. With the ConcurrentHashMap, this is not the case -- and in fact, a common lock is not available. The ConcurrentHashMap uses a reader-writer lock to allow parallel reads, and uses segmentation to allow parallel writes, and there isn't a global/common lock to grab.

**Hashtable**

**Collections.synchronizedMap(Map)**

**ConcurrentHashMap**

For your needs, use ConcurrentHashMap. It allows concurrent modification of the Map from several threads without the need to block them. Collections.synchronizedMap(map) creates a blocking Map which will degrade performance, albeit ensure consistency (if used properly).

Use the second option if you need to ensure data consistency, and each thread needs to have an up-to-date view of the map. Use the first if performance is critical, and each thread only inserts data to the map, with reads happening less frequently. The "scalability issues" for Hashtable are present in exactly the same way in Collections.synchronizedMap(Map) - they use very simple synchronization, which means that only one thread can access the map at the same time. This is not much of an issue when you have simple inserts and lookups (unless you do it extremely intensively), but becomes a big problem when you need to iterate over the entire Map, which can take a long time for a large Map - while one thread does that, all others have to wait if they want to insert or lookup anything. The ConcurrentHashMap uses very sophisticated techniques to reduce the need for synchronization and allow parallel read access by multiple threads without synchronization and, more importantly, provides an Iterator that requires no synchronization and even allows the Map to be modified during interation (though it makes no guarantees whether or not elements that were inserted during iteration will be returned).

HashTable and the synchronized wrapper class provide basic thread-safety by only allowing one thread at a time to access the map, this is not 'true' thread-safety since many compound operations still require additional synchronization, for example:

synchronized (records) {

Record rec = records.get(id);

if (rec == null) {

rec = new Record(id);

records.put(id, rec);

}

return rec;

}

In case of multiple reader and Single writer ConcurrentHashMap is best choice.

Since ConcurrentHashMap indroduced concept of segmentation , how large it becomes only certain part of it get locked to provide thread safety so many other readers can still access map without waiting for iteration to complete. In Summary ConcurrentHashMap only locked certain portion of Map while Hashtable lock full map while doing iteration.

**Conditional thread-safety**

The synchronized collections wrappers, synchronizedMap and synchronizedList, are sometimes called conditionally thread-safe -- all individual operations are thread-safe, but sequences of operations where the control flow depends on the results of previous operations may be subject to data races. The first snippet in Listing 1 shows the common put-if-absent idiom -- if an entry does not already exist in the Map, add it. Unfortunately, as written, it is possible for another thread to insert a value with the same key between the time the containsKey() method returns and the time the put() method is called. If you want to ensure exactly-once insertion, you need to wrap the pair of statements with a synchronized block that synchronizes on the Map m.

The other examples in Listing 1 deal with iteration. In the first example, the results of List.size() could become invalid during the execution of the loop, because another thread could delete items from the list. If the timing was unlucky, and an item was deleted by another thread just after entering the last iteration of the loop, List.get() will return null, and doSomething() will likely throw a NullPointerException. What can you do to avoid this? If another thread may be accessing a List while you are iterating through it, you must lock the entire List while you are iterating by wrapping it with a synchronized block, synchronizing on the List l. This addresses the data race, but has further costs for concurrency, since locking the entire List while iterating could block other threads from accessing the list for a long time.

The Collections framework introduced iterators for traversing a list or other collection, which optimizes the process of iterating through the elements in a collection. However, the iterators implemented in the java.util Collections classes are fail-fast, which means that if one thread changes a collection while another thread is traversing it through an Iterator, the next Iterator.hasNext() or Iterator.next() call will throw ConcurrentModificationException. Just as with the previous example, if you want to prevent ConcurrentModificationException, you must lock the entire List while you are iterating by wrapping it with a synchronized block that synchronizes on the List l. (Alternatively, you can invoke List.toArray() and iterate on the array without synchronization, but this could be expensive if the list is large.)

Listing 1. Common race conditions in synchronized maps

Map m = Collections.synchronizedMap(new HashMap());

List l = Collections.synchronizedList(new ArrayList());

// put-if-absent idiom -- contains a race condition

// may require external synchronization

if (!map.containsKey(key))

map.put(key, value);

// ad-hoc iteration -- contains race conditions

// may require external synchronization

for (int i=0; i<list.size(); i++) {

doSomething(list.get(i));

}

// normal iteration -- can throw ConcurrentModificationException

// may require external synchronization

for (Iterator i=list.iterator(); i.hasNext(); ) {

doSomething(i.next());

}

**False sense of confidence**

The conditional thread safety provided by synchronizedList and synchronizedMap present a hidden hazard -- developers assume that because these collections are synchronized, they are fully thread-safe, and they neglect to synchronize compound operations properly. The result is that while these programs appear to work under light load, under heavy load they may start throwing NullPointerException or ConcurrentModificationException.

**Scalability problems**

Scalability describes how an application's throughput behaves as its workload and available computing resources increase. A scalable program can handle a proportionally larger workload with more processors, memory, or I/O bandwidth. Locking a shared resource for exclusive access is a scalability bottleneck -- it prevents other threads from being able to access that resource, even if idle processors are available to schedule those threads. To achieve scalability, we must eliminate or reduce our dependence on exclusive resource locks. The bigger problem with the synchronized Collections wrappers, and the earlier Hashtable and Vector classes, is that they synchronize on a single lock. This means that only one thread may access the collection at once, and if one thread is in the process of reading from a Map, all other threads that want to either read from it or write to it must wait. The most common Map operations, get() and put(), may involve more computation than is obvious -- when traversing a hash bucket to find a specific key, get() may have to call Object.equals() on a large number of candidates. If the hashCode() function used by the key class does not spread values evenly over the hash range or has a large number of hash collisions, certain bucket chains may be much longer than others, and traversing a long hash chain and calling equals() on some percentage of its elements could be slow. The problem with get() and put() being expensive under these conditions is not only that access will be slow, but that all other threads are locked out from accessing the Map while that hash chain is being traversed. The fact that get() may take significant time to execute in some cases is made significantly worse by the conditional thread safety problem discussed above. The race conditions illustrated in Listing 1 often make it necessary to hold the single collection lock for much longer than it takes to execute a single operation. If you are going to hold the lock on the collection during an entire iteration, then other threads may be stalled waiting for the collection lock for a long time.

**Example: A simple cache**

One of the most common applications for Map in server applications is to implement a cache. Server applications may cache file contents, generated pages, results of database queries, DOM trees associated with parsed XML files, and many other types of data. The primary purpose of a cache is to reduce service time and increase throughput by reusing the results of a previous computation. A typical characteristic of cache workload is that retrievals are much more common than updates, so (ideally) a cache would offer very good get() performance. A cache that impedes application performance is worse than no cache at all.

If you use synchronizedMap to implement a cache, you are introducing a potential scalability bottleneck into your application. Only one thread can access the Map at once, and this includes all the threads that might be retrieving a value out of the Map as well as threads that want to install a new (key, value) pair into the map.

**Reducing lock granularity**

One approach to improving the concurrency of a HashMap while providing thread safety is to dispense with the single lock for the entire table, and use a lock for each hash bucket (or more commonly, a pool of locks where each lock protects several buckets). This means that multiple threads can access different portions of the Map simultaneously, without contending for the single collection-wide lock. This approach immediately improves the scalability of insertion, retrieval, and removal operations. Unfortunately, this concurrency has a cost -- it becomes harder to implement methods that operate on the collection as a whole, such as size() or isEmpty(), because this may require acquiring many locks at once or risk returning an inaccurate result. However, for situations such as implementing caches, this is a very sensible trade-off -- retrieval and insertion operations are frequent, and size() and isEmpty() operations are considerably less frequent.

**ConcurrentHashMap**

The ConcurrentHashMap class from util.concurrent (which will also appear in the java.util.concurrent package in JDK 1.5) is a thread-safe implementation of Map that offers far better concurrency than synchronizedMap. Multiple reads can almost always execute concurrently, simultaneous reads and writes can usually execute concurrently, and multiple simultaneous writes can often execute concurrently. (The related ConcurrentReaderHashMap class offers similar multiple-reader concurrency, but allows only a single active writer.) ConcurrentHashMap is designed to optimize retrieval operations; in fact, successful get() operations usually succeed with no locking at all. Achieving thread-safety without locking is tricky and requires a deep understanding of the details of the Java Memory Model. The ConcurrentHashMap implementation, along with the rest of util.concurrent, has been extensively peer-reviewed by concurrency experts for correctness and thread safety. We will look at the implementation details of ConcurrentHashMap in next month's article. ConcurrentHashMap achieves higher concurrency by slightly relaxing the promises it makes to callers. A retrieval operation will return the value inserted by the most recent completed insert operation, and may also return a value added by an insertion operation that is concurrently in progress (but in no case will it return a nonsense result). Iterators returned by ConcurrentHashMap.iterator() will return each element once at most and will not ever throw ConcurrentModificationException, but may or may not reflect insertions or removals that occurred since the iterator was constructed. No table-wide locking is needed (or even possible) to provide thread-safety when iterating the collection. ConcurrentHashMap may be used as a replacement for synchronizedMap or Hashtable in any application that does not rely on the ability to lock the entire table to prevent updates. These compromises enable ConcurrentHashMap to provide far superior scalability over Hashtable, without compromising its effectiveness in a wide variety of common-use cases, such as shared caches.

**How much better?**

Table 1 gives a rough idea of the scalability differences between Hashtable and ConcurrentHashMap. In each run, n threads concurrently executed a tight loop where they retrieved random key values values from either a Hashtable or a ConcurrentHashMap, with 80 percent of the failed retrievals performing a put() operation and 1 percent of the successful retrievals performing a remove(). Tests were performed on a dual-processor Xeon system running Linux. The data shows run time in milliseconds for 10,000,000 iterations, normalized to the 1-thread case for ConcurrentHashMap. You can see that the performance of ConcurrentHashMap remains scalable up to many threads, whereas the performance of Hashtable degrades almost immediately in the presence of lock contention.

The number of threads in this test may look small compared to typical server applications. However, because each thread is doing nothing but repeatedly hitting on the table, this simulates the contention of a much larger number of threads using the table in the context of doing some amount of real work.

Table 1. Scalability of Hashtable versus ConcurrentHashMap

Threads ConcurrentHashMap Hashtable

1 1.00 1.03

2 2.59 32.40

4 5.58 78.23

8 13.21 163.48

16 27.58 341.21

32 57.27 778.41

**CopyOnWriteArrayList**

The CopyOnWriteArrayList class is intended as a replacement for ArrayList in concurrent applications where traversals greatly outnumber insertions or removals. This is quite common when ArrayList is used to store a list of listeners, such as in AWT or Swing applications, or in JavaBean classes in general. (The related CopyOnWriteArraySet uses a CopyOnWriteArrayList to implement the Set interface.)

If you are using an ordinary ArrayList to store a list of listeners, as long as the list remains mutable and may be accessed by multiple threads, you must either lock the entire list during iteration or clone it before iteration, both of which have a significant cost. CopyOnWriteArrayList instead creates a fresh copy of the list whenever a mutative operation is performed, and its iterators are guaranteed to return the state of the list at the time the iterator was constructed and not throw a ConcurrentModificationException. It is not necessary to clone the list before iteration or lock it during iteration because the copy of the list that the iterator sees will not change. In other words, CopyOnWriteArrayList contains a mutable reference to an immutable array, so as long as that reference is held fixed, you get all the thread-safety benefits of immutability without the need

**Summary**

The synchronized collections classes, Hashtable and Vector, and the synchronized wrapper classes, Collections.synchronizedMap and Collections.synchronizedList, provide a basic conditionally thread-safe implementation of Map and List. However, several factors make them unsuitable for use in highly concurrent applications -- their single collection-wide lock is an impediment to scalability and it often becomes necessary to lock a collection for a considerable time during iteration to prevent ConcurrentModificationExceptions. The ConcurrentHashMap and CopyOnWriteArrayList implementations provide much higher concurrency while preserving thread safety, with some minor compromises in their promises to callers. ConcurrentHashMap and CopyOnWriteArrayList are not necessarily useful everywhere you might use HashMap or ArrayList, but are designed to optimize specific common situations. Many concurrent applications will benefit from their use.

http://dmy999.com/article/34/correct-use-of-concurrenthashmap

ConcurrentHashMap has been pitched as a simple alternative for HashMap, eliminating the need for a synchronized blocks. I had some simple event counting code that created count records on the fly. Although I could have used synchronized blocks for safety I used ConcurrentHashMap for this situation, partly for efficiency but mostly for the exercise. Going through this made me realize how carefully ConcurrentHashMap must be used for your code to work correctly and efficiently.

When using a HashMap, the standard idiom to add a value if it doesn’t exist is to use code that looks something like this:

synchronized (this) {

Record rec = records.get(id);

if (rec == null) {

rec = new Record(id);

records.put(id, rec);

}

return rec;

}

If you were to simply replace HashMap with ConcurrentHashMap and remove the synchronized keyword your code would be exposed to a race condition. If a new Record was put into the map just after the call to get returned null the put operation would overwrite the value. You could add synchronized back in but this defeats the purpose of using ConcurrentHashMap. To safely create values on demand you must use putIfAbsent (and avoid making extra calls to get in the process). First check to see if a value with the key already exists in the map and use this value if it does. Otherwise, create a new value for the map and add it with putIfAbsent. putIfAbsent returns any existing value if there is one, otherwise null (this is why ConcurrentHashMap can’t contain null values).

private ConcurrentMap<String, Record> records =

new ConcurrentHashMap<String, Record>();

private Record getOrCreate(String id) {

Record rec = records.get(id);

if (rec == null) {

// record does not yet exist

Record newRec = new Record(id);

rec = records.putIfAbsent(id, newRec);

if (rec == null) {

// put succeeded, use new value

rec = newRec;

}

}

return rec;

}

If putIfAbsent does return a value, it’s the one that must be used. It may have already been used by other threads at this point. The new value created must be abandoned. Although it sounds wasteful this case should happen very infrequently.

I’ve seen other code on the net that ignores the return value of putIfAbsent and makes another call to get at the end to figure out which value made it into the map (the new value created or a value from another thread). Although this will work it introduces an unnecessary lookup.

http://javarevisited.blogspot.in/2011/04/difference-between-concurrenthashmap.html

So what is the difference between Hashtable and ConcurrentHashMap , both can be used in multithreaded environment but once the size of Hashtable becomes considerable large performance degrade because for iteration it has to be locked for longer duration.

Since ConcurrentHashMap introduced concept of segmentation , how large it becomes only certain part of it get locked to provide thread safety so many other readers can still access map without waiting for iteration to complete.

In Summary ConcurrentHashMap only locked certain portion of Map while Hashtable lock full map while doing iteration.

http://vchenna.wordpress.com/2012/03/16/difference-between-concurrenthashmap-and-hashtable/

ConcurrentHashMap is offering all the features of Hashtable with a performance almost as good as a HashMap. ConcurrentHashMap’s accomplish this by a very simple mechanism. Instead of a map wide lock, the collection maintains a list of 16 locks by default, each of which is used to guard (or lock on) a single bucket of the map. This effectively means that 16 threads can modify the collection at a single time (as long as they’re all working on different buckets). Infact there is no operation performed by this collection that locks the entire map. The concurrency level of the collection, the number of threads that can modify it at the same time without blocking, can be increased. However a higher number means more overhead of maintaining this list of locks.

http://apurvagnihotri.blogspot.in/2010/06/hashmap-vs-hashtable.html

HashMap and HashTable both provide key-value access to data.

The Hashtable is among the original collection classes in Java.Hashtable extends the Dictionary class, which as the Javadocs state, is obsolete and has been replaced by the Map interface. HashMap is part of the new Collections Framework, added with Java 2.

The key difference between the two is that access to the Hashtable is synchronized on the table while access to the HashMap is not synchronized.This makes HashMap better for non-threaded applications, as unsynchronized Objects typically perform better than synchronized ones

HashMap has a more complex hashing algorithm then Hashtable. It takes the hash value from the key and then hashes it again (double hashing). This can improve the distribution of the keys and hence the performance of the Map.

Another difference is that iterator in the HashMap is fail-safe while the enumerator for the Hashtable isn't. If we change the map while iterating, it will throw exception.

Third difference is that HashMap permits null values in it, while Hashtable doesn't.Also note that only one NULL value is allowed as a key in HashMap. HashMap does not allow multiple keys to be NULL. Nevertheless, it can have multiple NULL values.

Using ConcurrentHashmap: for having a thread safe map we can use ConcurrenthashMap(from java5 onwards) as well instead of Hashtable which has become obsolete.

**private Map myConcMap = new ConcurrentHashMap();**

Now The question arises why ConcurrentHashMap and not HashTable or just have a synchronised access to HasMap.

So the major advantage of using ConcurrentHashMap is "performance" as the lock is not applied on wholeMap as is the case with a Synchronised access to hashmap or Hashtable.

As we know that hash maps store their data in a series of separate buckets, it is possible to lock only the portion of the map that is being accessed.ConcurrentHashMap uses this to provide us a highly optimized synchronised way of accessing HashMap.ConcurrentHash hash map follows following to provide a concurrent access:

1. Writing to a ConcurrentHashMap locks only a portion of the map

2. Reads can occur without locking.

Some disadvantges of ConcurrentHashMap:

ConcurrentHashMap will generally take up more memory.

it cannot take null as a key.

http://www.techlabs4u.com/2012/05/concurrenthashmap-in-java-example.html

Some of the drawbacks of Synchronized collection such as HashTable , Collections.synchronizedMap are as follows .

Synchronized collection classes such as Hashtable and the synchronized wrapper classes created by the Collections.synchronizedMap are thread safe with poor concurrency, less performance and scalabilty .

1. Poor concurrency : When these collections are accessed by two or more threads, they achieve thread safety by making the collection's data private and synchronizing all public methods so that only one thread at a time can access the collection (hashtable / synchronizedMap ) data. This leads to poor concurrency. As Single lock is used for the whole collection , multiple threads struggle for the collection wide lock which reduces the performance

2. ConcurrentModificationException :

When one thread is traversing the hashtable / Collections.synchronizedMap through an Iterator , while another thread changes it by mutative operations (put, remove , etc) , iterator implemented in the java.util collections classes fails by throwing ConcurrentModificationException . The exception occurs when the hasNext() or next() method of Iterator class is called. The same error also occurs (See Code Part 1 : ) , when elements are added in hashtable or synchronizedMap , once the iterator is constructed. While iterating the collection (hashtable) through iterator , collection / table- wide locking is required , otherwise ConcurrentModificationException is occured .

3. Scalabilty Issues :

Scalabilty is the major issue when we use synchronized collections . When the workload of the application increases , increasing the resources like processor , memory should also increase the throughtput of the application. Unfortunately , it does not happen . A scalable program can handle a proportionally larger workload with more resources. As synchronized collections synchronize on a single common lock , it restricts access to a single thread at a time, other threads are restricted to access that collections , even if the resources are available to schedule those threads.

4. Some of the common sequences of operations , such as put-if-absent (to check if an element is in the collection before adding it) or iteration , require external synchronization (i.e. client side locking ) (See Code Part 3 ) to avoid data races .

//Map hm=Collections.synchronizedMap(new HashMap());

Map hm=new Hashtable(new HashMap());

//ConcurrentHashMap hm=new ConcurrentHashMap();

hm.put(1, "Blue");

hm.put(2, "Green");

hm.put(3, "Yellow");

Iterator entries = hm.entrySet().iterator();

hm.put(4, "Red");

hm.put(5, "Orange");

while (entries.hasNext()) {

Map.Entry entry = (Map.Entry) entries.next();

Integer key = (Integer)entry.getKey();

String value = (String)entry.getValue();

System.out.println("Key = " + key + ", Value = " + value);}

To overcome the above issues with the synchronized collections , a new version of HashMap with concurrent access has been designed that is ConcurrentHashMap.

The main purpose to create ConcurrentHashMap is to provide

1. better concurrency

2 high scalability

3. thread safe

and it supports

1. full concurrency of retrievals. Allows all readers to read the table concurrently . No lock is used for retrival operations.

2. concurrency for writes . Allows a limited number of writers to update the table concurrently

3. full thread safe .

ConcurrentHashMap can be used where more read operation is required ( i.e. traversal is the dominant operation )

How a ConcurrentHashMap is implemented ? or How it works? or how concurrency is achieved?

Volatile fields and lock striping plays major role for to achieve concurrency .

Lock striping : Synchronizing every method on a single lock, restricts access to a single thread at a time. Instead of using single lock , ConcurrentHashMap uses different locking mechanism called lock striping to access the shared collection concurrently which increases the scalabilty and performance . Using different locks to allow different threads to operate on different portions of the same data structure called lock striping. Splitting the lock into more than one improves the scalability . For example two locks allow two threads to execute concurrently instead of one.

Lock splitting can sometimes be extended to partition locking on a variablesized set of independent objects, in which case it is called lock striping.

Now let see that how lock striping mechanism is applied to ConcurrentHashMap . The strategy is to subdivide the collection (hashtable) into independent subsets called segments each guarded by a lock sothat each subset (itself a hashtable) can be accessed concurrently. It uses an array of 16 locks each of which guards 1/16 of the hash buckets. N/16 locks are used for a hashtable having N hash buckets. Hash bucket N is guarded by lock N mod 16. 16 locks allow maximum of 16 threads to modify the hashtable at same time. Mutative operations such as put() and remove() use locks where as read operation does not use locks .

Note : The number of locks can be increased

Volatile Fields : Some of the volatile fileds declared in the ConcurrentHashMap are

transient volatile int count;

static final class HashEntry<K,V> {

final K key;

final int hash;

volatile V value;

volatile HashEntry<K,V> next;

HashEntry(K key, int hash, HashEntry<K,V> next, V value) {

.....

}

transient volatile HashEntry<K,V>[] table;

The above small piece of code is from the source of ConcurrentHashMap

As we know , volatile field ensures visibilty i.e. one thread reads the most up-to-date value written by another thread . For example count is the volatile field which is used to track the number of elements . When one thread adds an element to the table , the count is increased by one , Similarly when one thread removes an element from the table , the count is decreased by one . Now the other threads doing many read operations get the count variable's most recent updated value.

Similarly HashEntry<K,V>[] table , value , volatile HashEntry<K,V> next fileds are declared as volatile. This ensures that all the threads see the the most recent written value of those fields at all times.

When iterating the collection (hashtable) through iterator , it does not throw ConcurrentModificationException, but the elements added or removed after the iterator was constructed may or may not be reflected . No collection / table- wide locking is required while iterating the collection.

Issue with ConcurrentHashMap: How to protect / lock the entire collection? . There is no support for locking the entire table in a way that prevents all access. Then One way is to acquire all of the locks recursively which is costlier than using a single lock .

ConcurrentHashMap provides three new update methods:

putIfAbsent(key, value) - check if the key is in the collection before adding the specified key and associate it with the given value

replace( key, value) - Replace the existing key with given key , only if the key is mapped to given value.

remove(key, value) - Remove the key only if the key is mapped to given value.

The following program using ConcurrentHashMap helps to keep the accessed files in a cache .

Code Part 2 :

import java.util.\*;

import java.util.concurrent.ConcurrentHashMap;

import java.io.\*;

public class CacheUsingMap2 {

ConcurrentHashMap cache;

public CacheUsingMap2() {

cache = new ConcurrentHashMap();

}

public String getFile2(String fname) {

myFile fileObj=(myFile)cache.get(fname);

if (fileObj==null)

{

myFile fileObjNew=(myFile) readFile(fname);

fileObj= (myFile) cache.putIfAbsent(fname, fileObjNew );

if (fileObj==null) fileObj=fileObjNew;

}

return fileObj.getFileData();

}

public myFile readFile(String name)

{

File file = new File(name);

String fileData="";

try {

Scanner scan = new Scanner(file);

scan.useDelimiter("\\Z");

fileData = scan.next();

} catch (FileNotFoundException e){

System.out.println(e);

}

catch ( IOException e) {

System.out.println(e);

}

return (new myFile( fileData));

}

public static void main(String args[]) {

CacheUsingMap2 cache=new CacheUsingMap2();

String filePath="D:/Files/";

System.out.println( cache.getFile2(filePath+"k.txt"));

System.out.println( cache.getFile2(filePath+"k1.txt"));

System.out.println( cache.getFile2(filePath+"k.txt"));

System.out.println( cache.getFile2(filePath+"k1.txt"));

}

}

class myFile {

String fileData;

public myFile(String data)

{

fileData=data;

}

public String getFileData() {

return fileData;

}

}

Code Part 3 :

Sample code to createt cache using Hashtable (implements put-if-absent operation) which requires client side locking

Hashtable cache =new Hashtable();

---

public String getFile(String fname) {

// if (cache.get(fname)==null)

if (!cache.containsKey(fname))

{

synchronized(cache)

{

cache.put(fname, readFile(fname));

}

}

return ((myFile)cache.get(fname)).getFileData(); }

**How to know how many default threads are running when a program starts** .

import java.lang.management.ManagementFactory;

import java.lang.management.ThreadMXBean;

public class ShowAllDefaultThreads

{

public static void main(String[] args)

{

ThreadMXBean bean = ManagementFactory.getThreadMXBean();

long[] thIds = bean.getAllThreadIds();

String header = String.format("%-30s %-30s", "Thread Name","Thread State");

System.out.println(header);

for( long l : thIds )

{

String format = String.format("%-30s %-30s", bean.getThreadInfo(l)

.getThreadName(), bean.getThreadInfo(l).getThreadState());

System.out.println(format);

}

}

}

Output

**Thread Name Thread State**

**Attach Listener RUNNABLE**

**Signal Dispatcher RUNNABLE**

**Finalizer WAITING**

**Reference Handler WAITING**

**main RUNNABLE**

When a program starts in java, it creates 5 threads by default. The 5 threads are

**Attach Listener**

**Signal Dispatcher**

**Finalizer**

**Reference Handler**

**main**

**Dynamic Attach or Attach Listener Thread**

This is a Sun private mechanism that allows an external process to start a thread in HotSpot that can then be used to launch an agent to run in that HotSpot, and to send information about the state of HotSpot back to the external process.

**Signal Dispatcher**

When the OS raises a signal to the JVM, the signal dispatcher thread will pass the signal to the appropriate handler. The signal may be shutdown, Exceptions, Errors, Interrupts etc. Signal Dispatcher is a thread that handles the native signals send by the OS to your jvm.

**Reference Handler**

Reference Handler is a high-priority thread to enqueue pending References. its defined in java.lang.ref.References.java. The OpenJDK source states its is a High-priority thread to enqueue pending References. The GC creates a simple linked list of references which need to be processed and this thread quickly adds them to a proper queue. The reason this is done in two phases is that the GC does nothing but find the References, this thread calls code which handles those references e.g. Call Cleaners, and notifies ReferenceQueue listeners.

**How will you know that all threads have completed the tasks**

There will be some situations where you want to track whether all the threads have completed or not. Sometimes it is also required to check the time taken for the whole operations. Let us consider the small program.

import java.util.concurrent.TimeUnit;

public class MyThread1 extends Thread {

private long waitTime;

public MyThread1(String name,long waitTime) {

setName(name);

this.waitTime = waitTime;

}

@Override

public void run() {

try {

System.out.println(Thread.currentThread().getName()+" running ...");

TimeUnit.SECONDS.sleep(waitTime);

System.out.println(Thread.currentThread().getName()+" completed ...");

}

catch (Exception e) {

e.printStackTrace();

}

}

}

public class TestThread {

public static void main(String[] args) {

Thread th1 = new MyThread1("T1",2l);

Thread th2 = new MyThread1("T2",5l);

th1.start();

th2.start();

boolean flag = true;

while( flag ) {

if( !th1.isAlive() & !th2.isAlive() )

flag = false;

}

System.out.println("All threads completed ...");

}

}

In the above program, we have to handle like this

**boolean flag = true;**

**while( flag ) {**

**if( !th1.isAlive() & !th2.isAlive() )**

**flag = false;**

**}**

**System.out.println("All threads completed ...");**

The above solution is fine, what about the below program ?

**public class TestThread {**

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

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

**{**

**Thread th = new MyThread1("T"+i, 2);**

**th.start();**

**}**

**}**

**}**

In the above how will you track that all the above user threads have been completed.

You have to use the following piece of code.

**ThreadMXBean bean = ManagementFactory.getThreadMXBean();**

**boolean flag = true;**

**while( flag ) {**

**int threadCount = bean.getThreadCount();**

**if( threadCount == 5 )**

**{**

**flag = false;**

**}**

**}**

Now see the complete program.

import java.lang.management.ManagementFactory;

import java.lang.management.ThreadMXBean;

public class TestThread {

public static void main(String[] args) {

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

Thread th = new MyThread1("T"+i, 2);

th.start();

}

ThreadMXBean bean = ManagementFactory.getThreadMXBean();

boolean flag = true;

while( flag ) {

int threadCount = bean.getThreadCount();

if( threadCount == 5 )

{

flag = false;

}

}

System.out.println("All threads completed ...");

}

}

**Difference between synchronized method and synchronized block**

1) **Method locks**. When a method is synchronized then the lock is applied to the instance of the object in non-static context, and to the Class itself in static context. But in both cases, when you lock the method, you lock all other threads out of any and all synchronized methods (well...the same context, static or non-static being kept in mind) and they are locked out for the duration of the method, including any method calls invoked from within this method. That adds up to increased time spent "thumb-twiddling" by waiting threads. How much of that method's code really genuinely NEEDS to be done under lock protection? If you code well and make use of local variables you might be able to minimize the area that needs to be locked but, again, with method locks you lock the entire object(in non-static)/class(in static) for the duration of the method.

2) **Block locks**. Synchronizing a block of code allows two things: a) finer granularity of the area locked, and, b) selection/specification of a lock object rather than "this"(non-static) or the class(in static). The finer granularity part means that you only lock the code region that really needs to be locked. That means you have to code with a finer eye (any instance, static variables are treated as read-only, the working variables are local(on the stack ... so each thread has it's own), but that means threads can at least be concurrently active through portions of the called methods and contention for locks is minimized to just the required areas. Selection of a lock object means that you can have several locks protecting a different resource/set of resources. IN other words ... on any given thread execution though a particular method the resource might need to be synchronized, but, if the methods utilize different resources than those used by another method.... use a different lock. Any simple Object can be a lock.