Deadlock in Java Multithreading

synchronized keyword is used to make the class or method thread-safe which means only one thread can have lock of synchronized method and use it, other threads have to wait till the lock releases and anyone of them acquire that lock.

It is important to use if our program is running in multi-threaded environment where two or more threads execute simultaneously. But sometimes it also causes a problem which is called **Deadlock**. Below is a simple example of Deadlock condition.

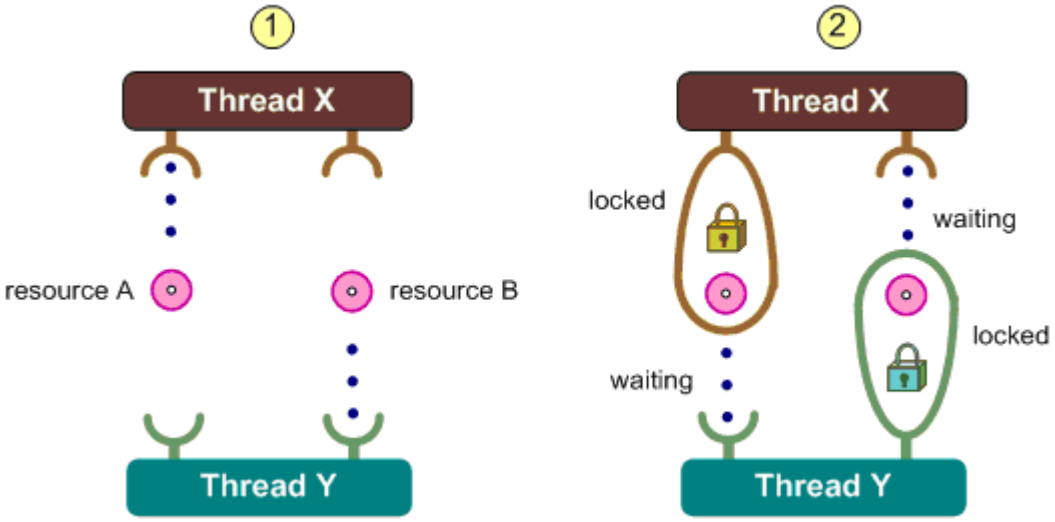

Image source: https://software.intel.com/en-us/articles/multi-threading-in-the-net-environment

```
// Java program to illustrate Deadlock
// in multithreading.
class Util
{
    // Util class to sleep a thread
    static void sleep(long millis)
    {
        try
        {
            Thread.sleep(millis);
        }
        catch (InterruptedException e)
```

```
        {
            e.printStackTrace();
        }
    }
}

// This class is shared by both threads
class Shared
{
    // first synchronized method
    synchronized void test1(Shared s2)
    {
        System.out.println("test1-begin");
        Util.sleep(1000);

        // taking object lock of s2 enters
        // into test2 method
        s2.test2(this);
        System.out.println("test1-end");
    }

    // second synchronized method
    synchronized void test2(Shared s1)
    {
        System.out.println("test2-begin");
        Util.sleep(1000);

        // taking object lock of s1 enters
        // into test1 method
        s1.test1(this);
        System.out.println("test2-end");
    }
}


class Thread1 extends Thread
{
    private Shared s1;
    private Shared s2;

    // constructor to initialize fields
    public Thread1(Shared s1, Shared s2)
    {
        this.s1 = s1;
        this.s2 = s2;
    }

    // run method to start a thread
    @Override
    public void run()
    {
        // taking object lock of s1 enters
        // into test1 method
        s1.test1(s2);
    }
}


class Thread2 extends Thread
{
    private Shared s1;
    private Shared s2;

    // constructor to initialize fields
    public Thread2(Shared s1, Shared s2)
    {
        this.s1 = s1;
        this.s2 = s2;
    }
```

```
    // run method to start a thread
    @Override
    public void run()
    {
        // taking object lock of s2
        // enters into test2 method
        s2.test2(s1);
    }
}


public class GFG
{
    public static void main(String[] args)
    {
        // creating one object
        Shared s1 = new Shared();

        // creating second object
        Shared s2 = new Shared();

        // creating first thread and starting it
        Thread1 t1 = new Thread1(s1, s2);
        t1.start();

        // creating second thread and starting it
        Thread2 t2 = new Thread2(s1, s2);
        t2.start();

        // sleeping main thread
        Util.sleep(2000);
    }
}
```

Run on IDE

```
Output : test1-begin
test2-begin
```

It is not recommended to run the above program with online IDE. We can copy the source code and run it on our local machine. We can see that it runs for indefinite time, because threads are in deadlock condition and doesn't let code to execute. Now let's see step by step what is happening there.

1. Thread t1 starts and calls test1 method by taking the object lock of s1.
2. Thread t2 starts and calls test2 method by taking the object lock of s2.
3. t1 prints test1-begin and t2 prints test-2 begin and both waits for 1 second, so that both threads can be started if any of them is not.
4. t1 tries to take object lock of s2 and call method test2 but as it is already acquired by t2 so it waits till it become free. It will not release lock of s1 until it gets lock of s2.
5. Same happens with t2. It tries to take object lock of s1 and call method test1 but it is already acquired by t1, so it has to wait till t1 release the lock. t2 will also not release lock of s2 until it gets lock of s1.
6. Now, both threads are in wait state, waiting for each other to release locks. Now there is a race around condition that who will release the lock first.

7. As none of them is ready to release lock, so this is the Dead Lock condition.
8. When you will run this program, it will be look like execution is paused.

Detect Dead Lock condition

We can also detect deadlock by running this program on cmd. We have to collect Thread Dump. Command to collect depends on OS type. If we are using Windows and Java 8, command is jcmd $PID Thread.print
We can get PID by running jps command. Thread dump for above program is below:

```
5524:
2017-04-21 09:57:39
Full thread dump Java HotSpot(TM) 64-Bit Server VM (25.25-b02 mixed mode):

"DestroyJavaVM" #12 prio=5 os_prio=0 tid=0x0000000002690800 nid=0xba8 waiting on condition [0x0000
   java.lang.Thread.State: RUNNABLE

"Thread-1" #11 prio=5 os_prio=0 tid=0x0000000018bbf800 nid=0x12bc waiting for monitor entry [0x0000
   java.lang.Thread.State: BLOCKED (on object monitor)
        at Shared.test1(GFG.java:15)
        - waiting to lock  (a Shared)
        at Shared.test2(GFG.java:29)
        - locked  (a Shared)
        at Thread2.run(GFG.java:68)

"Thread-0" #10 prio=5 os_prio=0 tid=0x0000000018bbc000 nid=0x1d8 waiting for monitor entry [0x00000
   java.lang.Thread.State: BLOCKED (on object monitor)
        at Shared.test2(GFG.java:25)
        - waiting to lock  (a Shared)
        at Shared.test1(GFG.java:19)
        - locked  (a Shared)
        at Thread1.run(GFG.java:49)

"Service Thread" #9 daemon prio=9 os_prio=0 tid=0x000000001737d800 nid=0x1680 runnable [0x000000000
   java.lang.Thread.State: RUNNABLE

"C1 CompilerThread2" #8 daemon prio=9 os_prio=2 tid=0x000000001732b800 nid=0x17b0 waiting on condit
   java.lang.Thread.State: RUNNABLE

"C2 CompilerThread1" #7 daemon prio=9 os_prio=2 tid=0x0000000017320800 nid=0x7b4 waiting on conditi
   java.lang.Thread.State: RUNNABLE

"C2 CompilerThread0" #6 daemon prio=9 os_prio=2 tid=0x000000001731b000 nid=0x21b0 waiting on condit
   java.lang.Thread.State: RUNNABLE

"Attach Listener" #5 daemon prio=5 os_prio=2 tid=0x0000000017319800 nid=0x1294 waiting on conditior
   java.lang.Thread.State: RUNNABLE

"Signal Dispatcher" #4 daemon prio=9 os_prio=2 tid=0x0000000017318000 nid=0x1efc runnable [0x
   java.lang.Thread.State: RUNNABLE
```

```
"Finalizer" #3 daemon prio=8 os_prio=1 tid=0x0000000002781800 nid=0x5a0 in Object.wait() [0x0000000
   java.lang.Thread.State: WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        - waiting on  (a java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(Unknown Source)
        - locked  (a java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(Unknown Source)
        at java.lang.ref.Finalizer$FinalizerThread.run(Unknown Source)

"Reference Handler" #2 daemon prio=10 os_prio=2 tid=0x000000000277a800 nid=0x15b4 in Object.wait()
   java.lang.Thread.State: WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        - waiting on  (a java.lang.ref.Reference$Lock)
        at java.lang.Object.wait(Unknown Source)
        at java.lang.ref.Reference$ReferenceHandler.run(Unknown Source)
        - locked  (a java.lang.ref.Reference$Lock)

"VM Thread" os_prio=2 tid=0x00000000172e6000 nid=0x1fec runnable

"GC task thread#0 (ParallelGC)" os_prio=0 tid=0x00000000026a6000 nid=0x21fc runnable

"GC task thread#1 (ParallelGC)" os_prio=0 tid=0x00000000026a7800 nid=0x2110 runnable

"GC task thread#2 (ParallelGC)" os_prio=0 tid=0x00000000026a9000 nid=0xc54 runnable

"GC task thread#3 (ParallelGC)" os_prio=0 tid=0x00000000026ab800 nid=0x704 runnable

"VM Periodic Task Thread" os_prio=2 tid=0x0000000018ba0800 nid=0x610 waiting on condition

JNI global references: 6


Found one Java-level deadlock:
=============================
"Thread-1":
  waiting to lock monitor 0x0000000018bc1e88 (object 0x00000000d5d645a0, a Shared),
  which is held by "Thread-0"
"Thread-0":
  waiting to lock monitor 0x0000000002780e88 (object 0x00000000d5d645b0, a Shared),
  which is held by "Thread-1"

Java stack information for the threads listed above:
===================================================
"Thread-1":
        at Shared.test1(GFG.java:15)
        - waiting to lock  (a Shared)
        at Shared.test2(GFG.java:29)
        - locked  (a Shared)
        at Thread2.run(GFG.java:68)
"Thread-0":
        at Shared.test2(GFG.java:25)
        - waiting to lock  (a Shared)
        at Shared.test1(GFG.java:19)
        - locked  (a Shared)
```

```
        at Thread1.run(GFG.java:49)

Found 1 deadlock.
```

As we can see there is clearly mentioned that found 1 deadlock. It is possible that the same message appears when you try on your machine.

Avoid Dead Lock condition

We can avoid dead lock condition by knowing its possibilities. It's a very complex process and not easy to catch. But still if we try, we can avoid this. There are some methods by which we can avoid this condition. We can't completely remove its possibility but we can reduce.

- **Avoid Nested Locks :** This is the main reason for dead lock. Dead Lock mainly happens when we give locks to multiple threads. Avoid giving lock to multiple threads if we already have given to one.
- **Avoid Unnecessary Locks :** We should have lock only those members which are required. Having lock on unnecessarily can lead to dead lock.
- **Using thread join :** Dead lock condition appears when one thread is waiting other to finish. If this condition occurs we can use Thread.join with maximum time you think the execution will take.

Important Points :

- If threads are waiting for each other to finish, then the condition is known as Deadlock.
- Deadlock condition is a complex condition which occurs only in case of multiple threads.
- Deadlock condition can break our code at run time and can destroy business logic.
- We should avoid this condition as much as we can.

This article is contributed by **Vishal Garg**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Java Java-Multithreading

Login to Improve this Page

Please write to us at contribute@geeksforgeeks.org to report any issue with the above content.