Section 15: Concurrency in Java

15 July 2021 19:23

Section 15: Concurrency in Java

A **Process** is a unit of execution that as has its own memory space.

If one java application is running and we run another one, each application has its own memory space of **heap.**

A **Thread** is a unit of execution within a process. Each process can have multiple threads.

Each thread has what's called a thread stack, which is the memory that only that thread can access.

So, every Java application runs as a single process, and each process can have multiple threads. Every process has a heap, and every thread has a thread stack.

Concurrency, which refers to an application doing more than one thing at a time. Basically Concurrency means that one task doesn't have to complete before another can start.

Run another thread :-

1st way of using Threads : Using Sub-classes the thread class

First Create a class :

```
public class AnotherThread extends Thread {

   @Override
   public void run()
   {
      System.out.println(ANSI_BLUE + "Hello from " + currentThread().getName());

   }
```

Then create an instance and start it :

```
Thread anotherThread = new AnotherThread();
anotherThread.setName("== Another Thread ==");
anotherThread.start();
```

There is no order of thread, until we set their priority.

Creating Anonymous Class :-

```
new Thread(
 {
         public void run() {
            System.out.println(ANSI_GREEN + "Hello from the anonymous class thread");
         }
      }.start();
```

2nd way of using Threads : Using Runnable Interface

First Create a class :

```
public class MyRunnable implements Runnable {

   @Override
   public void run() {
      System.out.println(ANSI_RED + "Hello from MyRunnable's implementation of run()");
   }
}
```

Then create an instance and start it :

```
Thread myRunnableThread = new Thread(new MyRunnable());
myRunnableThread.start();
```

Creating Anonymous Class of Runnable:-

```
Thread myRunnableThread = new Thread(new MyRunnable() {
         @Override
         public void run() {
            System.out.println(ANSI_RED + "Hello from the anonymous class's implementation of
run()");
         }
      });

      myRunnableThread.start();
```

Most Developer use runnable as it's more flexible and recommended.

Don't call run method directly (Will call main run, not thread run), must call start method.

Sleeping a thread :-

```
try {
         Thread.sleep(3000);
      } catch(InterruptedException e) {
         System.out.println(ANSI_BLUE +  "Another thread woke me up");
      }
```

Interrupts : We interrupt a thread when we want it to stop what it was doing to do something else.

```
anotherThread.interrupt();
```

Joining threads :- When we join one thread to another thread and the join times out we have to handle that case in a real world application

```
anotherThread.join(); // written in public void run() {…}
```

Local Variables are stored in thread stack, that means that each thread has its own copy of a local variable in contrast the memory required to store an object instance value is allocated on the heap.

The process of controlling when threads execute code and therefore when they can access the heap is called **synchronization.** (To prevent race condition)
If a class has three synchronize methods then only one of these methods can ever run at a time and only on one thread.

To make synchronized :-

1) Add synchronized in function name :-

```
public synchronized  void doCountdown() { …. }
```

2) We can also synchronize a block of statements rather than an entire method :-
synchronized(this) { … }

As a general rule it's easy to just remember not to use local variables to synchronize.

Critical section just refers to the code that's referencing a shared resource like a variable only one thread at a time should be able to execute a critical section.

Thread-Safe : When a class or method is thread-safe what that means is that the developer has synchronized all the critical sections within the code so that we as a developer don't have to worry about the thread interference.

Deadlocks, wait, notify and notifyAll Methods :-
https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#synchronizedList(java.util.List)

Thread Interference :-
https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

Instead of using synchronization we can prevent thread interference using classes that implement the java.util.concurrent locks.lock interface

Use :-
ReentrantLock bufferLock = new ReentrantLock();
Replace synchronized block with :
bufferLock.lock();
…… block …….
bufferLock.unlock();

Put bufferLock.lock(); before try block and bufferLock.unlock(); in finally block.
https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/Lock.html

Thread Pools :-

Creating Executive Service :-

ExecutorService executorService = Executors.newFixedThreadPool(3); // where 3 = 3 number of threads.

New Thread(object).start() is replaced by executorService.execute(producer);

Also need to shut down the executive service using executorService.shutdown();

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html

Thread safe means that we can be confident that our call to one of the class methods will complete before another thread can run a method in the class.

Deadlock : Deadlock occurs when two or more thread are blocked on locks and every thread that's blocked is holding a lock that another block thread wants.
For example : Thread 1 is holding Lock 1 and waiting to acquire Lock 2 but Thread 2 is holding Lock 2 and waiting to acquire Lock 1. Because all the threads holding the locks are blocked they will never release the locks they're holding and so none of the waiting threads will actually even run.

How to prevent Deadlock :-
1) Use 1 lock, but that's not possible as many applications use multiple locks.
2) If we made both treads obtain the locks in the same order, a deadlock can't occur.
3) Another solution would be to use a lock object rather than using synchronized blocks.

Starvation :- When starvation occurs it's not that threads will never progress because they'll never get a lock but that they rarely have the opportunity to run and progress. So starvation often occurs due to thread priority when we assign a high priority to a thread we are suggesting to the operating system that it should try and run the thread before other waiting threads.

Setting Priority :-

```
Thread t1 = new Thread(new Worker(ThreadColor.ANSI_RED), "Priority 10");
t1.setPriority(10);
t1.start();

private static class Worker implements Runnable { …….. }
```

It's just priority, it still totality depend on operating system to run threads.

Setting priorities can make starvation more likely to happen, So how to prevent it.
While we are dealing in deadlocks, the order in which locks are require was important, but the starvation which thread gets to run when a lock becomes available is important

Alternative to Synchronous blocks is : Fair Locks and Live Locks

ReentrantLock :-

```
private static ReentrantLock lock = new ReentrantLock(true); // true means it's first come first served - FIFO
```

The only thing fair lock guarantees is the first come first served ordering for getting the lock. Secondly the try lock method doesn't honor the fairness settings so it will not be first come first served and lastly when using fair locks with a lot of threads keep in mind that performance will be impacted to ensure fairness, may things get slow down when there are lot of threads.

Using :-
Replace synchronized block with :-

```
lock.lock();
try {
            ……….……
              // execute critical section of code
finally {
              lock.unlock();
        }
```

Another problem we can have while working with threads is **live lock** which is similar to deadlock but instead of the threads been blocked they're actually constantly active and usually waiting for all the other threads to complete their tasks.

The next potential problem that can arise in a multi-threaded application is called a **slipped condition.** This is a specific type of race condition (aka thread interference).
It can occur when a thread can be suspended between reading a condition and acting on it.

Solution to slipped condition is the same as it is for any type of thread interference: use synchronized blocks or locks to synchronize the critical action of code.
If the code is already synchronized, then sometimes the placement of the synchronization may be causing the problem.
When using multiple locks, the order in which the locks can be acquired can also result in a slipped condition.

Thread Issues :-

An **Atomic Action** can't be suspended in the middle of being executed. It either completes, or it doesn't happen at all. Once a thread starts to run an atomic action, we can be confident that it can't be suspended until it has completed the action.

Atomic Action are as follows :-

1) Reading and writing reference variables. Statement myObject1 = myObject2
2) Reading and writing primitive variables, except those of type long and double.
    a. myInt = 10 is atomic action, but myDouble = 1.234 is not. (But we can use the AtomicLong and AtomicDouble classes to make theses operations atomic)
3) Reading and writing all variables declared volatile.
    a. Public volatile int counter;
    b. When we use a volatile variable, the JVM writed the value back to main memory immediately after a thread updates the value in its CPU cache.
    c. It also guarantees that every time a variable reads from a volatile, it will get the latest value.

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/package-summary.html
https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicInteger.html

JavaFX Background Tasks :-
https://docs.oracle.com/javase/8/javafx/api/javafx/concurrent/Task.html

If a variable is a local variable, it's already threadsafe. Local variables are stored on the thread stack, so each thread will have its own copy. Threads won't interface with each other when it comes to setting and getting its value.

When dealing with deadlock situation, look for the following :-

1) Is a set of locks being obtained in a different order by multiple threads. If so, can we force all threads to obtain the locks in the same order?
2) Are we over synchronizing the code?
3) Can we rewrite the code to break any circular call patterns?
4) Would using ReentrantLock object help?