Thread Methods

07 September 2020 07:48 PM

Interrupt() and InterruptedException.

The interrupt() method has two effects.

First, it causes any blocked method to throw an InterruptedException. In our example, the sleep() method is a blocking method. If the event-processing **(t2)** thread interrupts the RandomCharacterGenerator thread**(t1)** while that thread is executing the **sleep()** method, the sleep method immediately wakes up and throws an InterruptedException. Other methods that behave this way include the **wait()** method, the **join()** method, and methods that **read I/O**
Refer case 1.

The second effect is to set a flag inside the thread object that indicates the thread has been interrupted. To query this flag, use the isInterrupted() method. That method returns true if the thread has been interrupted.
This is when the Thread is not in waiting state
Refer case 4.

Documentation from Java Docs 8

public void interrupt()
Interrupts this thread.
Unless the current thread is interrupting itself, which is always permitted, the checkAccess method of this thread is invoked, which may cause a SecurityException to be thrown.

Case 1:
*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.***

Case 2: IO specific
*If this thread is blocked in an I/O operation upon an InterruptibleChannel then the channel will be closed, the thread's **interrupt status will be set**, and the thread **will receive a ClosedByInterruptException.***

Case 3: IO specific
*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.*

Case 4:
*If none of the previous conditions hold then this thread's **interrupt status will be set.***

Interrupting a thread that is not alive need not have any effect.

Throws:
SecurityException - if the current thread cannot modify this thread

Use checkAccess method on thread before interrupting it.

public final void checkAccess()

Determines if the currently running thread has permission to modify this thread.
If there is a security manager, its checkAccess method is called with this thread as its argument. This may result in throwing a SecurityException.

Throws:

SecurityException - if the current thread is not allowed to access this thread.

From <https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html#checkAccess-->

Examples on usage of interrupt method to stop the forever executing thread immediately even if they might be using blocking calls like wait(), sleep(some millis), join()

2nd Example for terminating forever running thread: Run method for thread t2.

```
public void run( ) {
        while (!isInterrupted( )) { // can call interrupt method because class has extended Thread class and not implemented
                                        Runnable Interface
                                        // Other alternatives
                                        // Thread.currentThread.isInterrupted();
                                        // Thread.interrupted() internally calls Thread.currentThread.isInterrupted()
                try {
                  Thread.sleep(time);
                }
                catch (InterruptedException) {
                        break/return;
                        Refer: Note 1
                }
        }
}
```

It's possible for the t1 to call the interrupt() method on t2 just after the t2 has called the isInterrupted() method. The t2 thread still executes the sleep() method, which won't be interrupted . **This is the problem not interrupting immediately**

Why skipped 1st example.
In 1st example we just use flag in the thread parent class extended child class instance object which will be set to true and false to notify the thread to terminate while loop.

Refer for java docs <https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html#interrupt-->
Statement referred from Java threads 3rd edition chapter 2 Thread creation and Management - Two Approaches to stopping the thread.

Note 1

In the catch statement here, you must jump out of the loop. According to the description of the interrupt method of the Thread class, if the Interrupt method of thread t1 is called, if t1 is in the sleep, wait, and join methods and is blocked, t1 The thread will receive an InterruptedException, but the interrupt state of t1 will be cleared (that is, isInterrupted() will return false)

currentThread() Method

Documentation from Java Docs 8

public static Thread currentThread()
Returns a reference to the currently executing thread object.

Returns:

the currently executing thread.

Usage

Example 1:
You can retrieve a reference to the current thread by calling the currentThread() method (a static method of the Thread class). Therefore, to see if code is being executed by an arbitrary thread (as opposed to the thread represented by the object), you can use this pattern:

```
public class MyThread extends Thread {
        public void run( ) {
                if (Thread.currentThread( ) != this) throw new IllegalStateException("Run method called by incorrect thread");
                 ... main logic ... }
                }
        }
…
}
```

Example 2:

Similarly, within an arbitrary object, you can use the currentThread() method to obtain a reference to a current thread. This technique can be used by a Runnable object to see whether it has been interrupted:

```
public class MyRunnable implements Runnable {
        public void run( ) {
                while (!Thread.currentThread( ).isInterrupted( )) {
                        ... main logic ...
                }
        }
}
```

Design questions

07 September 2020 09:08 PM

Extend thread class or implement Runnable interface ?

Inheriting Thread class
allows you to leverage thread class functionality in the run method itself which is preferable depending on the implementation and design, But you loose the flexibility to extend other classes (as Java does not support multiple inheritance of classes).

Also some concurrency utilities in Java take Runnable instance as parameter instead of Thread instance.
These classes handle thread pooling, task scheduling, or timing issues. If you're going to use such a class, your task must be a Runnable object

Implementing Runnable interface
This allows you to separate the implementation of the task from the thread to run the task.

It’s a trade off between flexibility(Runnable) and simplicity(Thread).

Synchronized Methods, Explicit locking and Synchronized block

08 September 2020 11:50 PM

Synchronized Keyword (when used on methods)

it is simply prevents two or more threads from calling the methods of the same object at the same time.

Definition: Mutex Lock A mutex lock is also known as a mutually exclusive lock. This type of lock is provided by many threading systems as a means of synchronization. Only one thread can grab a mutex at a time: if two threads try to grab a mutex, only one succeeds. The other thread has to wait until the first thread releases
the lock before it can grab the lock and continue operation.

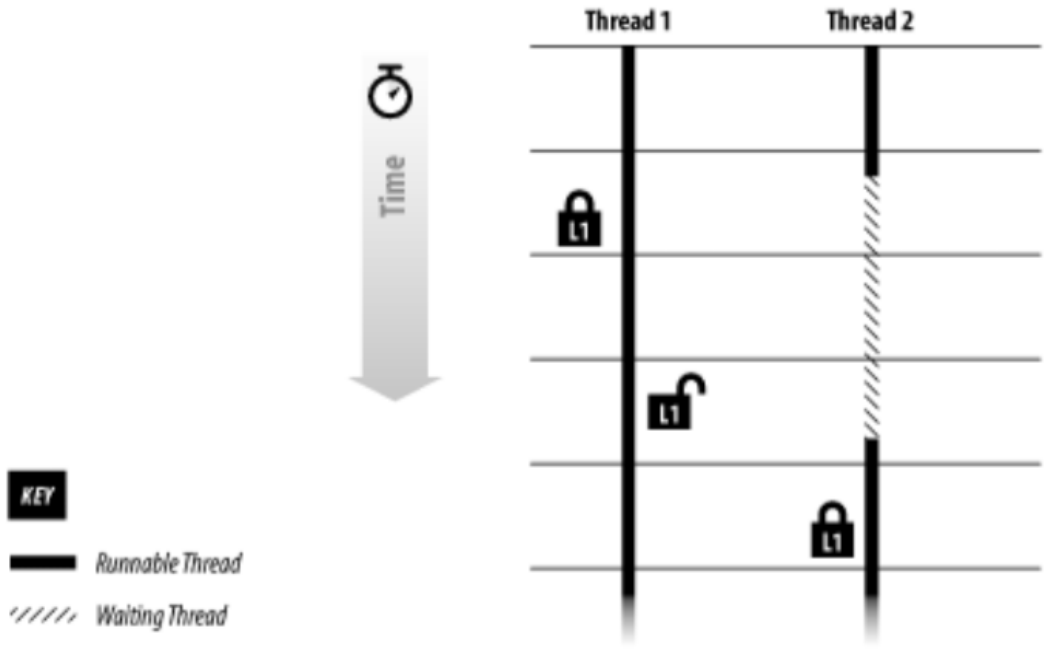
In Java, every object has an associated lock. When a method is declared synchronized, the executing thread must grab the lock associated with the object(**on which the method is called**) before it can continue. Upon completion of the method, the lock is automatically released.

Under the covers, the concept of synchronization is simple: when a method is declared synchronized, the thread that wants to execute the method must acquire a token, which we call a lock. Once the method has acquired (or checked out or grabbed) this lock, it executes the method and releases (or returns) the lock. **No matter how the method returns—including via an exception—the lock is released.** There is only one lock per object, so if two separate threads try to call synchronized methods of the same object, only one can execute the method immediately; the other has to wait until the first thread releases the lock before it can execute the method.

How can synchronizing two different methods prevent multiple threads calling those methods from stepping on each other?

As stated earlier, synchronizing a method has the effect of serializing access to the method. This means that it is not possible to execute the same method in one thread while the method is already running in another thread. The implementation of this mechanism is done by a lock that is assigned to the object itself. The reason another thread cannot execute the same method at the same time is that the method requires the lock that is already held by the first thread. If two different synchronized methods of the same object are called,
they also behave in the same fashion because they both require the lock of the same object, and it is not possible for both methods to grab the lock at the same time. In other words, even if two or more methods are involved, they are never run in parallel in separate threads. This is illustrated in Figure 3-1. When thread 1 and thread 2 attempt to acquire the same lock (L1), thread 2 must wait until thread 1 releases the
lock before it can continue to execute.

Figure 3-1. Acquiring and releasing a lock



The point to remember here is that the lock is based on a specific instance of an object and not on any particular method or class. Assume that we have two different scoring components that score based on different formulas; we'll call these two ScoreLabel objects called objectA and objectB. One thread can execute the objectA.newCharacter() method while another thread executes the objectB.resetGenerator( ) method. These two methods can execute in parallel because the call to the objectA.newCharacter() method grabs the lock associated with instance variable objectA, and the call to the objectB.resetGenerator() method grabs the object lock associated with instance variable objectB. Since the two objects are different objects, two different locks are grabbed by the two threads: neither thread has to wait for the other.

How does a synchronized method behave in conjunction with an unsynchronized method?
To understand this, we must remember that all synchronizing does is to grab an object lock. This, in turn, provides the means of allowing only one synchronized method to run at a time, which in turn provides the data protection that solves the race condition. Simply put, a synchronized method tries to grab the object lock, and an unsynchronized method doesn't. This means that unsynchronized methods can execute at any time, by any thread, regardless of whether a synchronized method is currently running. At any given moment on any given object, any number of unsynchronized methods can be executing, but only one synchronized method can be executing.

What does synchronizing static methods do? And how does it work?
Throughout this discussion, we keep talking about "obtaining the object lock." But what about static methods? When a synchronized static method is called, which object are we referring to? A static method does not have a concept of the this reference. It is not possible to obtain the object lock of an object that does not exist. So how does synchronization of static methods work? To answer this question, we will introduce the concept of a class lock. Just as there is an object lock that can be obtained for each instance of a class (i.e., each object), there is a lock that can be obtained for each class. We refer to this as the class lock. In terms of implementation, there is no such thing as a class lock, but it is a useful concept to help us understand how all this works.

When a static synchronized method is called, the program obtains the class lock before calling the method. This mechanism is identical to the case in which the method is not static; it is just a different lock. And this lock is used solely for static methods. Apart from the functional relationship between the two locks, they are not operationally related at all. These are two distinct locks. The class lock can be grabbed and released independently of the object lock. If a nonstatic synchronized method calls a static synchronized method, it acquires both locks.

As we mentioned, a class lock does not actually exist. The class lock is the object lock of the Class object that models the class. Since there is only one Class object per class, using this object achieves the synchronization for static methods. For the developer, it is best envisioned as follows. Only one thread can execute a synchronized static method per class. Only one thread per instance of the class can execute a nonstatic synchronized method. Any number of threads can execute nonsynchronized methods — static or otherwise.

Explicit Locking (Refer Scope of the lock)

Although almost all the needs of data protection can be accomplished with synchronized keyword then why we need explicit locking ?

Synchronized keyword is too primitive when the need for complex synchronization arises. More complex cases can be handled by using classes that achieve similar functionality as the synchronized keyword.

The Lock Interface: Lock interface's lock() and unlock() methods:
Using the Lock interface is similar to using the synchronized keyword: we call the lock() method at the start of the method and call the unlock() method at the end of the method, and we've effectively synchronized the method. The lock() method grabs the lock. **The difference is that the lock can now be more easily envisioned: we now have an actual object that represents the lock. This object can be stored, passed around, and even discarded.** As before, if another thread owns the lock, ***a thread that attempts to acquire the lock waits until the other thread calls the unlock() method of the lock***. Once that happens, the waiting thread grabs the lock and returns from the lock( ) method. If another thread then wants the lock, it has to wait until the current thread calls the unlock() method.

Advantages of Lock.

- Using a lock class, we can now **grab and release a lock whenever desired**.
- **We can test conditions before grabbing or releasing the lock.** And since the lock is no longer attached to the object whose method is being called, it is now possible for two objects to share the same lock.
- **It is also possible for one object to have multiple locks.**
- **Locks can be attached to data, groups of data, or anything else, instead of just the objects that contain the executing methods.**

```
private Lock customLock = new ReentrantLock( ); // A Class that implements Lock Interface.


public testMethod() {
        If (somecondition) {   // if somecondition is true then only acquire lock.
                try {
                        customLock.lock();
                } finally {
                        customLock.unlock();
                }

        }
}
```

Instead of declaring methods as synchronized, those methods now call the lock() method on entry and the unlock() method on exit. Finally, **the method bodies are now placed in try/finally clauses to handle possible runtime exceptions. With the synchronized keyword, locks are automatically released when the method exits. Using locks, we need to call the unlock() method: by placing the unlock() method call in a finally clause, we guarantee the method is called when the method exits, even if an unexpected runtime exception is thrown.**

How are the Lock classes related to static methods ?
As far as lock objects are concerned, it doesn't matter if the method being executed is static or not. As long as the method has a reference to the lock object, it can acquire
the lock. For complex synchronization that involves both static and nonstatic methods, it may be easier to use a lock object instead of the synchronized keyword.

Synchronized Blocks (Refer Scope of the lock)

Using Synchronized block we can achieve similar things as explicit locks, we can have synchronization at block of code instead at whole method (lock at scope smaller than method). We can have a different object altogether which will passed to synchronized block and this object can be used to obtain and release lock instead of the same object which contains the method.

```
synchronized (object) {
        …
        …
        …
}
```

Choosing a locking mechanism

1. Lock object has more overhead while implementation,
    - We need to create Lock objects.
    - We need to grab and release lock by calling lock and unlock method.
    - We need to have try and finally blocks to make sure we release the lock in case of any exception.

2. Using synchronized method is much more easier than Lock object but it lock scope spans unnecessary lines of code where synchronization is not needed. Hence it has larger lock scope than using Lock classes.

3. Synchronized block becomes more complex when you have too many object as locks. There is also a slight overhead in grabbing and releasing the lock, so it may be inefficient to free a lock just to grab it again a few lines of code later.

   It depends on case to case for example if we don’t have complex threading problem we go for synchronized methods or blocks. But if we do have complex threading problems, we should use explicit locks because of the additional functionalities that they provide, refer **More on Explicit Locking** section**.**

Scope of the lock

09 September 2020 12:08 AM

Definition: Scope of a Lock The scope of a lock is defined as the period of time between when the lock is grabbed and released.

For a synchronized method the scope of these locks is the period of time it takes to execute the methods. This is referred to as method scope. Which is very inefficient as we should limit the scope of lock which will not block other threads for executing the lines of method which does not need to be synchronized.

This inefficiency can be solved using synchronized blocks and explicit locks.

In case of explicit locks since we have different object altogether on which we have flexibility to call lock and unlock methods to grab and release the lock unlike the synchronized method which restricts to grab and release the lock at the method level. Due this we can move them anywhere, establishing any lock scope, from a single line of code to a scope that spans multiple methods and objects. By providing the means of specifying the scope of the lock, we can now move time-consuming and thread-safe code outside of the lock scope. And we can now lock at a scope that is specific to the program design instead of the object layout.

In case of synchronized block we can have scope on block of code inside the method, but still the flexibility of actually grabbing the lock and releasing it cannot be achieved here. Synchronized block allows to reduce the scope at block level.
Synchronized blocks also cannot establish a lock scope that spans multiple methods. Since the locking object is different from the object on which method is called and scope can be reduced to block we can do more flexible operations than Synchronized method.

Volatile Keyword

09 September 2020 12:26 AM

Java specifies that basic loading and storing of variables (except for long and double variables) is atomic. That means the value of the variable can't be found in an interim state during the store, nor can it be changed in the middle of loading the variable to
a register.

Unfortunately, Java's memory model is a bit more complex. Threads are allowed to hold the values of variables in local memory (e.g., in a machine register). In that case, when one thread changes the value of the variable, another thread may not see the changed variable. This is particularly true in loops that are controlled by a variable (like the done flag that we are using to terminate the thread): the looping thread may have already loaded the value of the variable into a register and does not necessarily notice when another thread changes the variable.

**If a variable is marked as volatile, every time the variable is used it must be read from main memory. Similarly, every time the variable is written, the value must be stored in main
memory.**

Volatile variables solve only the problem introduced by Java's memory model. **They can be used only when the operations that use the variable are atomic, meaning the methods that access the variable must use only a single load or store. If the method has other code, that code may not depend on the variable changing its value during its operation. For example, operations like increment and decrement (e.g., ++ and --) can't be used on a volatile
variable because these operations are syntactic sugar for a load, change, and a store.**

The requirements of using volatile variables seem overly restrictive. Are they really important? This question can lead to an unending debate. For now, it is better to think of the volatile keyword as a way to force the virtual machine not to make temporary copies of a variable.

There are some alternatives called Atomic Variables instead of volatile keyword.

How does volatile work with arrays? Declaring an array volatile makes the array reference itself volatile. The elements within the array are not volatile; the virtual machine may still store copies of individual elements in local registers.

Volatile Keywords restricts Java interpreter and JIT compiler to perform optimizations on the instructions by reordering them.

Race Condition and when it is a Problem

09 September 2020 11:19 PM

A race condition occurs when the order of execution of two or more threads may affect some variable or outcome in the program. It may turn out that all the different possible orders of thread execution have the same final effect on the program: the effect caused by the race condition may be insignificant and may not even be relevant.

Alternatively, the timing of the threading system may be such that the race condition never manifests itself, despite the fact that it exists in the code.
Race conditions can be considered harmless (or benign) **if you can prove that the result of the race condition is always the same.** This is a common technique in some of Java's core classes (**most commonly, the atomic classes**).

But in general, a race condition is a problem that is waiting to happen. Simple changes in the algorithm can cause race conditions to manifest themselves in problematic ways. Since different virtual machines have different ordering of thread execution, **the developer should never let a race condition exist even if it is currently not causing a problem on the development system.**

More on Explicit Locking

13 September 2020 12:58 PM

Through this Note, we will see that the functionality offered by the Lock interface exceeds the functionality offered by the synchronized keyword. By using explicit locks, the developer is free to address issues specific to his program instead of being swamped with concurrency issues.

The Lock Interface:

```
public interface Lock {

        void lock( );

        void lockInterruptibly( ) throws InterruptedException;

        boolean tryLock( );

        boolean tryLock(long time, TimeUnit unit) throws InterruptedException;

        void unlock( );

        Condition newCondition( );
}
```

What if we want to do other tasks if we can't obtain the lock? The Lock interface provides an option to try to obtain the lock:

Java 8 Docs
Method boolean tryLock():
Acquires the lock only if it is free at the time of invocation.
Acquires the lock if it is available and returns immediately with the value true. If the lock is not available then this method will return immediately with the value false.
A typical usage idiom for this method would be:

```
Lock lock = ...;
if (lock.tryLock()) {
  try {
    // manipulate protected state
  } finally {
    lock.unlock();
  }
} else {
  // perform alternative actions
}
```

This usage ensures that the lock is unlocked if it was acquired, and doesn't try to unlock if the lock was not acquired.

Returns:

true if the lock was acquired and false otherwise

From <https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/Lock.html>

By inspecting the return value, we can route the thread to separate tasks: if the value returned is false, for instance, we can route the thread to perform alternative tasks that do not require obtaining the lock.

What if we want to wait only for a specific period of time for a lock?

Java 8 Docs

Method boolean tryLock(long time, TimeUnit unit) throws InterruptedException

Acquires the lock if it is free within the given waiting time and the current thread has not been interrupted.

If the lock is available this method returns immediately with the value true. If the lock is not available then the current thread becomes disabled for thread scheduling purposes and lies dormant until one of three things happens:

- *The lock is acquired by the current thread; or*
- *Some other thread interrupts the current thread, and interruption of lock acquisition is supported; or*
- *The specified waiting time elapses*

If the lock is acquired then the value true is returned.

If the current thread:

- *has its interrupted status set on entry to this method; or*
- *is interrupted while acquiring the lock, and interruption of lock acquisition is supported,*

then InterruptedException is thrown and the current thread's interrupted status is cleared.

If the specified waiting time elapses then the value false is returned. If the time is less than or equal to zero, the method will not wait at all.

From <https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/Lock.html>

method is overloaded with a version that lets you specify the maximum time to wait. This method takes two parameters: one that specifies the number of time units and a TimeUnit object that specifies how the first parameter should be interpreted. For example, to specify 50 milliseconds, the long value is set to 50 and the TimeUnit
value is set to TimeUnit.MILLISECONDS.

Nested Locks

Nested locking is referred to the scenario where system keeps track of the nested methods being called which requires same lock acquired by the first method call.

Example: These two methods belong to the same class. When the first thread calls methodOne it grabs the lock associated with the object and when other two nested calls happen to methodTwo and methodThree the thread does not wait to get the lock free or even to grab the lock, it just goes and executes these methods.

```
synchronized methodOne() {
        methodtwo();
}
```

```
synchronized methodTwo() {
        methodThree();
}

synchronized methodThree() {
}
```

The system is smart enough not to free the lock if it is completed with call to either methodTwo or methodThree.

This works because the system keeps track of the number of recursive acquisitions of the lock, finally freeing the lock upon exiting the first method (or block) that acquired the lock. **This functionality is called nested locking.**

How Nested locks are supported by ReentrantLock Class?

Nested locks are also supported by the ReentrantLock class—the class that implements the Lock interface that we have been using so far. If a lock request is made by the thread that currently owns the lock, the ReentrantLock object just increments an internal count of the number of nested lock requests. Calls to the unlock() method decrement the count. The lock is not freed until the lock count reaches zero. This implementation allows these locks to behave exactly like the synchronized keyword. Note, however, that **this is a specific property of the ReentrantLock class and not a general property of classes that implement the Lock interface.**

What if java does not support nested Locking ? Example on Cross-calling methods.
Point 1: So if we don't keep track of acquisitions of lock then we have to release the lock if we are calling nested synchronized methods which requires same lock. Why? As we don't track acquisitions we don't know if the thread stack unrolls when to release the lock.

So taking into consideration Point 1.
Example: grab lock in methodOne, before calling methodTwo release the lock so that methodTwo can garb it and continue the work.

Now consider we have two objects objectA with synchronized method objectAMethod and objectB with synchronized method objectBMethod.

If t1 thread calls objectA's objectAMethod acquiring lock A.
This objectAMethod calls objectB's objectBMethod acquiring lock B.

Now if objectBMethod calls objectA's objectAMethod it will be deadlock.

As thread t1 will wait on lock A to be released, but it won't get release since the method execution acquiring lock A is not completed and will never be completed, As the same execution is halt due to wait on same object acquired by the same method.

So nested Locking is important to support nested synchronized method calls.

ReentrantLocks extra functionalities:

The Lock interface does not provide a means to detect the number of nested acquisitions.
However, that functionality is implemented by the ReentrantLock class:

```
public class ReentrantLock implements Lock {
        public int getHoldCount( );
        public boolean isLocked( );
        public boolean isHeldByCurrentThread( );
        public int getQueueLength( );
}
```

The **getHoldCount()** method returns the number of acquisitions that the current thread has made on the lock. A return value of zero means that the current thread does not own the lock: it does not mean that the lock is free.

To determine if the lock is free—not acquired by any thread—the **isLocked()** method may be used.

The **isHeldByCurrentThread()** method is used to determine if the lock is owned by the current thread.

The **getQueueLength()** method can be used to get an estimate of the number of threads waiting to acquire the lock. This value that is returned is only an estimate due to the race condition that exists between the time that the value is calculated and the time that the value is used after it has been returned.

Lock Fairness

What if we want locks to be issued in a fair fashion? What does it mean to be fair? The ReentrantLock class allows the developer to request that locks be granted fairly. This just means that locks are granted in as close to arrival order as possible. While this is fair for the
majority of programs, the definition of "fair" can be much more complex.

Fairness is subjective as the program needs differ, So simply the fairness depends on the algorithm of the program and minimally based on the synchronization constructs that the program uses. The best that the threading library can accomplish is to grant locks in a fashion that is specified and consistent.

We can declare the lock like this instead:
private Lock scoreLock = new ReentrantLock(true);

The ReentrantLock class provides an option in its constructor to specify whether to issue locks in a "fair" fashion. In this case, the definition of "fair" is first-in-first-out. This means that when many lock requests are made at the same time, they are granted very close to the order in which they are made. At a minimum, this prevents
lock starvation from occurring.

Deadlock

13 September 2020 03:23 PM

Deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.

Deadlock can arise if following four conditions hold simultaneously (Necessary Conditions)

Mutual Exclusion: One or more than one resource are non-sharable (Only one process can use at a time)

Hold and Wait: A process is holding at least one resource and waiting for resources.

No Preemption: A resource cannot be taken from a process unless the process releases the resource.

Circular Wait: A set of processes are waiting for each other in circular form.

Methods for handling deadlock.

We can have deadlock prevention and avoidance (Banker's algorithm,).

We can have deadlock detection and recovery (Detecting cycle in graph - graph theory).

In java deadlock might occur if two or more threads are waiting on two or more locks to get free and the circumstances is that the locks will never get freed.

Example:

```
methodOne() {
        acquireLock(A)
        acquireLock(B)
        …
        …
        releaseLock(A)
        releaseLock(B)
}

methodTwo() {
        acquireLock(B)
        acquireLock(A)
        …
        …
        releaseLock(A)
        releaseLock(B)
}
```

If method One is called by thread t1 and it acquires lock A, after that CPU processing time is assigned to thread t2 so thread t1 goes into wait. Now thread t2 calls methodTwo and acquires lock B. Now the situation is t1 and t2 have lock A and lock B and both of them in their next line of execution sequence will try to acquire each other's locks and Deadlock will occur.

Interestingly, it is possible to deadlock even if no synchronization locks are involved.

A deadlock situation involves threads waiting for conditions; this includes waiting to acquire a lock and also waiting for variables to be in a particular state.

Example:
One thread checks done variable to exits its run method each time in a loop. Run method itself is synchronized since done variable is a shared resource. Other thread is responsible to call setDone method which is also synchronized.
This is deadlock because thread t1 at start itself acquires the lock and keeps on checking done variable to be set to true.
As thread t2 tries to call setDone it goes in waiting/blocked state due lock acquired by t1 thread and hence done will never be set to true.

On the other hand, it is not possible to deadlock if only one thread is involved, as Java allows nested lock acquisition. If a single user thread deadlocks, a system thread must also be involved.

Wait and Notify - has the condition occurred ?

15 September 2020 10:40 PM

Note: This notes are all related to synchronized methods and synchronized blocks

We've seen that every Java object has a lock (Monitor). In addition, every object also provides a mechanism that allows it to be a waiting area; this mechanism aids communication between threads.

The idea behind the mechanism is simple:

one thread needs a certain condition to exist and assumes that another thread will create that condition. When another thread creates the condition, it notifies the first thread that has been waiting for the condition

What is the purpose of the wait-and-notify mechanism, and how does it work?

The wait-and-notify mechanism is a synchronization mechanism. However, it is more of a **communication mechanism**:
it allows one thread to communicate to another thread that a particular condition has occurred. The wait-and-notify mechanism **does not specify what the specific condition is.**

Can the wait-and-notify mechanism be used to replace the synchronized mechanism?

Actually, the answer is no; wait-and-notify **does not solve the race condition problem that the synchronized mechanism solves**. As a matter of fact, wait-and-notify must be used in conjunction with the synchronized lock to **prevent a race condition in the wait-and-notify mechanism itself**.

How we communicate between threads without Wait and Notify?

Without wait and notify in one thread we might need to use loops and variables(booleans) to know if the condition has occurred or not and if the condition does not occur, we spin in the loop. We might also use sleep method so that it does not eats the precious processing time. Once the other thread sets the condition, first thread will go ahead.

The one thing about sleep method is that if sleep method is called within a synchronized code it does not release the monitor lock it **sleeps while holding the lock.** is it similar to hold and wait ? - *one of the condition for deadlock.*

How to use Wait and Notify ?

When thread sees that a particular condition has not occurred it calls the wait() method (with no arguments). The thread waits (or blocks) in that method until another thread calls the notify method, at which point it restarts its execution again.

Why wait is better that sleep ?

method: **void wait(long timeout)**

Waits for a condition to occur. However, if the notification has not occurred in timeout milliseconds, it returns anyway.

Wait method is actually used to stop processing by thread and hold till a condition has occurred and other thread will notify that the condition has occurred. We can use this version to wait for some timeout and if the condition has not occurred then carry some other work again.

The differences between the wait() and sleep() methods is that unlike the sleep( ) method, the wait() method requires that the thread own the synchronization lock of the object (more about this later). **When the wait() method executes, the synchronization lock is released (internally by the virtual machine itself). Upon receiving the notification, the thread needs to reacquire the synchronization lock before returning from the wait() method.**

While reacquiring the monitor or lock again the thread does not get any special privileges it does have to compete for lock again.

Why wait and notify methods (*wait(), wait(long timeout), notify(), notifyAll()*) needs to be called when thread holds lock or monitor, why we can't call them without holding the lock?

This technique is needed due to a race condition that would otherwise exist between setting and sending the notification and testing and getting the notification. In general, a thread that uses the wait() method confirms that a condition does not exist (typically by checking a variable) and then calls the wait() method. When another thread establishes the condition (typically by setting the same variable), it calls the notify() method.

A race condition occurs when:

1. The first thread tests the condition and confirms that it must wait.
2. The second thread sets the condition.
3. The second thread calls the notify() method; this goes unheard since the first thread is not yet waiting.
4. The first thread calls the wait() method.

If the wait() and notify() mechanism were not invoked while holding the synchronization lock, there would be no way to guarantee that the notification would be received. And if the wait() method did not release the lock prior to waiting, it would be impossible for the notify() method to be called (as it would

be unable to obtain the lock). This is also why we had to use the wait() method instead of the sleep() method; if the sleep( ) method were used, the lock would never be released.

It is not possible to solve the race condition without integrating the lock into the wait-and-notify mechanism. This is why it is mandatory for the wait() and notify() methods to hold the locks for the object on which they are operating.

How does the potential Race Condition gets solved ?

In order to call the wait() or notify() methods, we must have obtained the lock for the object on which we're calling the method. This is mandatory; the methods do not work properly and generate an exception **(IllegalMonitorStateException)** condition if the lock is not held.

Is there a race condition during the period that the wait() method releases and reacquires the lock?

The wait() method is tightly integrated with the lock mechanism. The object lock is not actually freed until the waiting thread is already in a state in which it can receive notifications. This would have been difficult, if not impossible, to accomplish if we had needed to implement the wait() and notify() methods ourselves.

What happens when notify() is called and no thread is waiting?

if the notify() method is called when no other thread is waiting, notify() simply returns and the notification is lost. A thread that later executes the wait() method has to
wait for another notification to occur.

If a thread receives a notification, is it guaranteed that the condition is set correctly?

Simply, no. Prior to calling the wait() method, a thread should always test the condition while holding the synchronization lock. Upon returning from the wait() method, the thread should always retest the condition to determine if it should wait again. This is because another thread t2 can also test the condition and determine that a wait is not necessary — processing the valid data that was set by the notification thread.

Example: If there are three threads one is producer and there are two different consumer threads, the producer produce only one value at a time. If consumers are in wait and after notification one of the consumer gets the lock, it consumes the value and again goes in wait and now the other consumer gets the lock, it should not simply go ahead in an assumption that the condition will still be holding for it.

Can a thread that called wait() can be awakened without notify?

Yes if the thread is interrupted by some other thread, In that case, processing is application-specific, depending on how the algorithm needs to handle the interruption

What happens when more than one thread is waiting for notification? Which threads actually get the notification when the notify() method is called?

It depends: the Java specification doesn't define which thread gets notified. Which thread actually receives the notification varies based on several factors, including the **implementation of the Java virtual machine** and scheduling and timing issues during the execution of the program. There is no way to determine, even on a single processor platform, which of multiple threads receives the notification.

Another method of the Object class assists us when multiple threads are waiting for a condition:

The notifyAll() method is similar to the notify() method except that all of the threads that are waiting on the object are notified instead of a single arbitrary thread. Just like the notify() method, the notifyAll() method does not allow us to decide which thread gets the notification: they all get notified. All of the waiting threads wake up, but they still have to reacquire the object lock. So the threads do not run in parallel: they must each wait for the object lock to be freed.
Thus, only one thread can run at a time,

Notify and NotifyAll method does not release the lock when called with in synchronized block, it just makes sure to wake the threads that are waiting for the condition to occur on a particular object or monitor.

Why would you want to wake up all of the threads?

There are a few reasons. For example, there might be more than one condition to wait for. Since we cannot control which thread gets the notification, it is entirely possible that a notification wakes up a thread that is waiting for an entirely different condition. **Refer Topic for condition Interface there we discuss options to allow multiple condition variables to coexist. This allows different threads to wait for different conditions efficiently.**

Condition Interface - create different conditions

16 September 2020 12:38 AM