Lock Class

A java.util.concurrent.locks.Lock is a thread synchronization mechanism just like synchronized blocks. A Lock is, however, more flexible and more sophisticated than a synchronized block.

Since Lock is an interface, you need to use one of its implementations to use a Lock in your applications.

First a Lock is created. Then it's lock() method is called. Now the Lock instance is locked. Any other thread calling lock() will be blocked until the thread that locked the lock calls unlock(). Finally unlock() is called, and the Lock is now unlocked so other threads can lock it.

The java.util.concurrent.locks package has the following implementations of the Lock interface:

ReentrantLock

The Lock interface has the following primary methods:

lock()

lockInterruptibly()

tryLock()

tryLock(long timeout, TimeUnit timeUnit)

unlock()

The lock() method locks the Lock instance if possible. If the Lock instance is already locked, the thread calling lock() is blocked until the Lock is unlocked.

The lockInterruptibly() method locks the Lock unless the thread calling the method has been interrupted. Additionally, if a thread is blocked waiting to lock the Lock via this method, and it is interrupted, it exits this method calls.

The tryLock() method attempts to lock the Lock instance immediately. It returns true if the locking succeeds, false if Lock is already locked. This method never blocks.

The tryLock(long timeout, TimeUnit timeUnit) works like the tryLock() method, except it waits up the given timeout before giving up trying to lock the Lock.

The unlock() method unlocks the Lock instance. Typically, a Lock implementation will only allow the thread that has locked the Lock to call this method. Other threads calling this method may result in an unchecked exception (RuntimeException)

If **fairness** is a requirement, then a fair lock can be created by:

new ReentrantLock(true);

SampleLockCode:

Lock lock = new ReentrantLock();

lock.lock();

//critical section

lock.unlock();

Race Condition

Race condition in software is an undesirable event that can happen when multiple entities access or modify shared resources in a system. The system behaves correctly when these entities use the shared resources as expected. But sometimes due to uncontrollable delays, the sequence of operations may change due to relative timing of events. When this happens, the system may enter a state not designed for and hence fail. The "race" happens because this type of failure is dependent on which entity gains access to the shared resource first.

The usual **solution** to avoid race condition is to serialize access to the shared resource. If one process gains access first, the resource is "locked" so that other processes have to wait for the resource to become available.

DeadLocks

Deadlock describes a situation where two or more threads are blocked forever, waiting for each other.

Deadlock is a common problem in multiprocessing systems, parallel computing, and distributed systems, where software and hardware locks are used to handle shared resources and implement process synchronization.

For parallel processes to co-operate correctly and efficiently using shared data we need four conditions to hold:

* + No two processes may be simultaneously inside their critical sections;
  + No assumptions may be made about the speeds or number of CPUs;
  + No process running outside its critical section may block other processes;
  + No process should have to wait forever to enter its critical section.

Fork Join & Work Stealing

This is a recursive algorithm that divides a task until it is manageable and then solves the sub task. To do this it recursively forks new tasks and then waits (join) for all sub-tasks to complete. The underlying tasks are managed by a threadpool that uses work stealing. The ThreadPoolExecutor had a central inbound queue for new tasks (Runnables or Callables), which is shared by all worker threads.

It takes the form:

Result solve(Problem problem) {

if (problem is small)

directly solve problem

else {

split problem into independent parts

fork new subtasks to solve each part

join all subtasks

compose result from subresults

}

}

Nondeterministic vs deterministic

**Nondeterminism** means that the path of execution isn't fully determined by the specification of the computation, so the same input can produce different outcomes

**Deterministic** execution is guaranteed to be the same, given the same input.

Volatile variables

The Java volatile keyword guarantees visibility of changes to variables across threads.

The Java volatile keyword is used to mark a Java variable as "being stored in main memory". More precisely that means, that every read of a volatile variable will be read from the computer's main memory, and not from the CPU cache, and that every write to a volatile variable will be written to main memory, and not just to the CPU cache.

However synchronization ensures both atomicity and visibility (volatile does not ensure atomicity).

Fine Grained Locking

Code that synchronizes in such a way as to allow as much concurrency as possible is said to be fine-grained.

There are degrees of access and the more access allowed the more fine-grained the locking becomes.

A fine-grained implementation optimizing thread access could use an array of Object to provide a lock for each element in an array. Multiple threads accessing the same element are forced to wait on a given element lock. This guarantees thread safety while optimizing thread access.

Coarse Grained Locking

The more a program synchronizes in such a way as to inhibit concurrency, the more *coarse-grained* it is said to be

Intrinsic locking in the case of classes is coarse-grained because it only permits a single thread at a time access to the resource. This, of course, is necessary to protect atomic actions on the data.

A coarse-grained solution simply makes each method synchronized. Then when multiple threads wish to access different elements of data, one is given access and all the rest are needlessly held up.

Intrinsic Locking

Intrinsic locking means that you don't have to create an object to synchronize your methods on. It enforces exclusive access to an object's state and establishes happens-before relationships that are essential to visibility.

In java intrinsic locking can be achieved by using the synchronized keyword.

Thread Safety

Thread safety is a property that allows code to run in multithreaded environments by re-establishing some of the correspondences between the actual flow of control and the text of the program, by means of synchronization.

An alternative to Thread Safety is immutable classes.

CountDownLatch

The CountDownLatch class is a flexible latch implementation that allows one or more threads to wait for a set of events to occur. The CountDownLatch is initialized with a positive number, which represents the number of events to wait for. Once an event has occurred the count is decremented. When all the events have occurred the count will be zero (terminal state) and the waiting threads can proceed to execute concurrently.

It has 2 important methods:

* await
* countdown

CyclicBarrier

Barriers are similar to latches in that they block a group of threads until some event

has occurred. However **all the threads must come together, at a barrier point, at the**

**same time, in order to proceed**. CyclicBarriers allow a fixed number of threads to

rendezvous repeatedly at a barrier point. A barrier can be reset and used again if

required. CyclicBarriers are useful in programs involving a fixed sized party of

threads that must occasionally wait for each other. The barrier is called cyclic because

it can be re-used after the waiting threads are released.

Streams

A stream is a pipeline linking data **suppliers** with data **consumers**

Streams can be connected to data suppliers but are only activated by data consumers

A stream requests data from a supplier in response to a request from a consumer

It provides a channel or channels through which the data flows

It can actively monitor the flow of data and in certain circumstances transform it

Streams are **lazy** in the sense that they only respond to requests from consumers

Thread States

A thread executes in the context of the virtual machine. During its lifetime it may pass through a number of different states because it shares the processor with other threads including main. A thread can only be in one state at any given time and it changes state under a given set of pre-defined rules. The list of possible states is given in the following table. The names of the thread states form an enumerated class and, hence, are written with capital letters.

|  |  |
| --- | --- |
| **Name** | **State Description** |
| NEW | A thread that has not yet started |
| RUNNABLE | A thread executing in the Java virtual machine. A thread in this state may be waiting for resources from the underlying operating system. For example, it might be on a ready-to-run queue waiting to be allocated a processor. |
| BLOCKED | A thread that is blocked waiting for a monitor lock. (We will discuss this state when we come to dealing with sharing resources.) |
| WAITING | A thread that is waiting for another thread to perform a particular action. (We will discuss this state when we come to dealing with sharing resources.) |
| TIMED\_WAITING | A thread that is waiting for another thread to perform an action, for up to a specified waiting time. For example, a thread is in this state when it is sleeping for a given period of time. |
| TERMINATED | A thread that has exited |

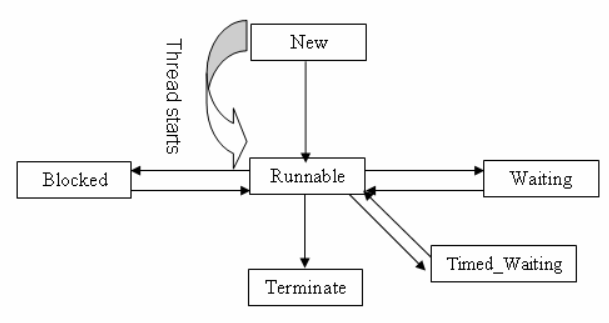
The state of a thread may be go by calling the getState method like:

myThread.getState();

You may need to get the current thread first using Thread.currentThread();

To wait for a thread to finish use the join method like so: myThread.join();

Join throws an InterruptedException so must be in a try and catch.



Process

A process, in the simplest terms, is an executing program. One or more threads run in the context of the process. A thread is the basic unit to which the operating system allocates processor time

Processes v Threads

Both processes and threads are independent sequences of execution. The typical difference is that threads (of the same process) run in a shared memory space, while processes run in separate memory spaces.

*Process*

Each process provides the resources needed to execute a program. A process has a virtual address space, executable code, open handles to system objects, a security context, a unique process identifier, environment variables, a priority class, minimum and maximum working set sizes, and at least one thread of execution. Each process is started with a single thread, often called the primary thread, but can create additional threads from any of its threads.

*Thread*

A thread is the entity within a process that can be scheduled for execution. All threads of a process share its virtual address space and system resources. In addition, each thread maintains exception handlers, a scheduling priority, thread local storage, a unique thread identifier, and a set of structures the system will use to save the thread context until it is scheduled. The thread context includes the thread's set of machine registers, the kernel stack, a thread environment block, and a user stack in the address space of the thread's process. Threads can also have their own security context, which can be used for impersonating clients

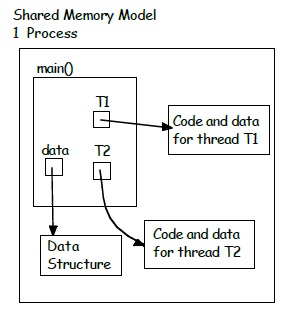
Thread v Runnable

A thread can be defined in two ways. First, by extending a Thread class that has already implemented a Runnable interface. Second, by directly implementing a Runnable interface. When you define a thread by extending Thread class you have to override the run() method in Thread class. When you define a thread implementing a Runnable interface you have to implement the only run() method of Runnable interface. The basic difference between Thread and Runnable is that each thread defined by extending Thread class creates a unique object and get associated with that object. On the other hand, each thread defined by implementing Runnable interface shares the same object.

|  | **Thread** | **Runnable** |
| --- | --- | --- |
| Basic | Each thread creates a unique object and gets associated with it. | Multiple threads share the same objects. |
| Memory | As each thread create a unique object, more memory required. | As multiple threads share the same object less memory is used. |
| Extending | In Java, multiple inheritance not allowed hence, after a class extends Thread class, it cannot extend any other class. | If a class define thread implementing the Runnable interface it has a chance of extending one class. |
| Use | A user must extend thread class only if it wants to override the other methods in Thread class. | If you only want to specialize run method then implementing Runnable is a better option. |
| Coupling | Extending Thread class introduces tight coupling as the class contains code of Thread class and also the job assigned to the thread | Implementing Runnable interface introduces loose coupling as the code of Thread is separate from the job of Threads. |

Shared memory model

All the threads share common memory in the context of a single process. This makes thread communication simple because threads can share access to data by sharing reference variables.



Run Method

All Java code that runs Threads must implement the Runnable interface or extend the Thread class (which implements the Runnable interface)

The Runnable interface has 1 method call run. When implement it should contain the code executed in the thread.

The Thread class itself implements Runnable, though its run method does nothing. An application can subclass Thread, providing its own implementation of run.

In both cases you invoke Thread.start to start the new thread.

Subclasses of Thread and Runnable must override the run method.

Synchronous vs Asynchronous

When you execute something synchronously, you wait for it to finish before moving on to another task. When you execute something asynchronously, you can move on to another task before it finishes.

A synchronous operation blocks a process until the operation completes. An asynchronous operation is non-blocking and only initiates the operation. The caller could discover completion by some other mechanism discussed later.

Note that synchronous/asynchronous implies blocking/not blocking but not vice versa, that is, not every blocking operation is synchronous and not every non-blocking operation is asynchronous.

Atomic Action

In programming, an atomic action is one that effectively happens all at once. An atomic action cannot stop in the middle: it either happens completely, or it doesn't happen at all. No side effects of an atomic action are visible until the action is complete.

An atomic action is one that can be executed in the knowledge that no interference can occur from another thread when it happens. Java only guarantees that reading from and writing to a memory word is atomic. A memory word is 32 bits on 32-bit machines and 64 bits on 64-bit machines. This means that if two integer variables attempt to write to the same address simultaneously, the writes will occur in some order. The memory word will have one of the values that were written to it, not a mixture of the two. Similarly, if a read and a write on the same memory word in memory is executed simultaneously, the write will be successful and the read will deliver either the value that was in the word before the write, or the value just written.

The reason why reads and writes of long and double are not atomic is because it requires two reads, the first word (32 low order bits), followed by the second word (32 high order bits) to complete a read. A similar situation occurs in the case of a write instruction.

In java we ensure atomic actions by ensuring that all threads are synchronized on the shared reference variable(s). For example each thread would create a synchronized block.

ThreadPools

A thread pool is a class to help in achieving concurrency. A thread pool maintains multiple threads waiting for tasks to be allocated for concurrent execution by the supervising program. By maintaining a pool of threads, the model increases performance and avoids latency in execution due to frequent creation and destruction of threads for short-lived tasks.

The Components of a Threadpool are:

* Worker threads that execute the callback functions
* Waiter threads that wait on multiple wait handles
* A work queue
* A default thread pool for each process
* A worker factory that manages the worker threads

Semaphore

A semaphore is a thread synchronizing data structure invented by Dijkstra to solve, amongst other problems, the mutual exclusion problem.

This control manages a single integer value that is initialized to a non-negative value and has two public methods acquire() and release().

When a thread invokes the acquire() operation on a semaphore it checks to see if the value is greater than 0. If so, it decrements the value and allows the thread to continue. If the value is 0, the thread is put to sleep.

Checking the value, changing it, and possibly going to sleep is all done as a single, indivisible, atomic action. It is guaranteed that once a semaphore operation has started, no other process can access the semaphore until the operation has completed or blocked.

The release() operation increments the value of the semaphore. If one or more processes were sleeping on that semaphore, unable to complete an earlier acquire() operation, one of them is chosen by the system (e.g., at random), and is allowed to complete its acquire().

It is important to note the meaning of the value, n, of a semaphore: n positive means that n threads can decrement its value without going to sleep; n negative means that there are n threads sleeping on the semaphore; n zero means that there are no threads waiting but the next thread to invoke acquire will sleep.

Note that acquire() throws an InterruptedException and, hence, must be used in a try…catch block.

Serialization

Serialization is the process of converting a data structure or object state into a binary format that can be stored on file or sent over a network. A serialized object is an object represented as a sequence of bytes that includes all data as well as information about its type and the types of data stored in the object (its methods are not included).

It is a binary stream of bytes that represent the state of a complete object. This object may be a single instance of a simple class or one that instantiates a collection of items.

E.G.:

class Password implements java.io.Serializable

Parallel Streams

A parallel stream is a stream with a single source and a single consumer where aggregrate operations are executed in parallel. The source data set is divided and distributed among the different streams and passed through all intermediate operations to be consumed by a single terminal consumer.

Provides parallel data processing

Uses ForkJoinPool.commonPool to provide parallel processing

Does not guarantee order preservation

Consumers, suppliers and intermediate operations can all be executed in parallel

The **disadvantages** of Parallel Streams is that we don’t know how many streams are actually used and we have no guarantees about the order in which the data elements will be consumed by forEach.

Consumers are used in all contexts where an object needs to be consumed, i.e. taken as an input and some operation is to be performed on the object without returning any result.

Suppliers can be used in all contexts where there is no input but an output is expected.

Aggregate/intermediate operations process elements from a stream, not directly from a collection