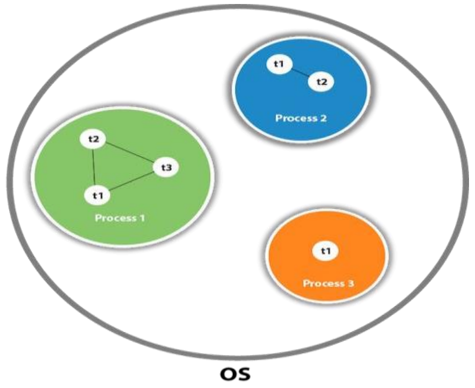
Basics of Multithreading

Agenda

- Basics of MultiThreading
- Threads in Java
- Volatile Keyword
- Synchronization in Threads
- Producer-Consumer Problem
- Blocking Queues
- Deadlock
- Reentrant Locks
- Semaphores
- Countdown Latches
- Thread Pool
- Callable and Future
- Interrupting a Thread

Basic of Multithreading

- Multithreading refers to a process of executing two or more threads simultaneously.
- A thread is a lightweight sub-process, the smallest unit of processing.
- Threads are independent from each-other.
- Threads use shared resources for maximum utilization.



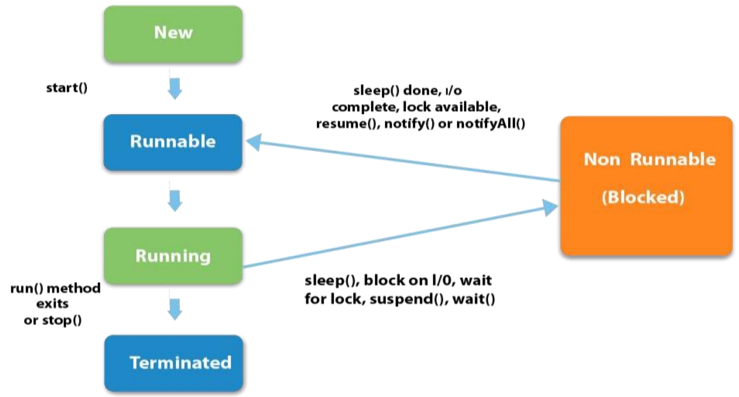
Types of Thread

- User Thread: These are High priority threads. Any thread created by user program are called user threads.
- Daemon Thread: These are Low priority threads. The role of Daemon thread is to provide services to the User Thread.

Threads in Java

There are two ways to create a thread in Java.

- By Extending Thread Class
- By Implementing Runnable Interface



New
start()
Runnable
sleep() done, I/o complete, lock available, resume(), notify() or notifyAll()
Non Runnable (Blocked)
Running
sleep(), block on I/0, wait for lock, suspend(), wait()
run() method exits or stop()
Terminated

States of Thread

- New
- Runnable
- Running
- Non-Runnable (Blocked)
- Terminated

Thread in Java (Continued..)

Constructor

- Thread(Runnable task)
- Thread(Runnable task, String name)
- Thread(ThreadGroup threadGroup, Runnable task, String name)

Static Method

- void sleep(long mills): To sleep the current thread for specified amount of time.
- void yield(): To pause current thread and allow other threads to execute temporarily.
- Thread currentThread(): Returns a reference to the currently executing thread object.
- boolean interrupted(): Return if the current thread has been interrupted.
- Integer activeCount(): Returns number of active threads.
- boolean holdLock(Object obj): Returns if the current thread holds the lock on the specified object.

Non Static Method

- void start(): To start execution of the thread.
- void run(): To specify action of the thread.
- Thread.State getState(): Returns the current state of the thread.

- void join() / join(long millis): Makes the calling thread wait till the complete execution of the thread on which join is called or timeout if specified.
- void setPriority(int priority): To set priority of the thread.
- int getPriority(): Returns priority of the thread.
- void setName(String name): To set name of the thread.
- String getName(): Returns name of the thread.
- long getId(): Returns id of the thread.
- ThreadGroup getThreadGroup(): Returns the thread group of the thread.
- boolean isAlive(): Returns if the thread is alive.
- void suspend(): To suspend the execution of the thread.
- void resume(): To resume the execution of the thread.
- void stop(): To stop the execution of the thread.
- void destroy(): To destroy the thread group and it’s sub group.
- void setDaemon(boolean on): To make a thread daemon thread.
- boolean isDaemon(): Returns if the thread is daemon thread.
- void interrupt(): To interrupt the thread.
- boolean isinterrupted(): Returns if the thread is interrupted.
- void checkAccess(): Executes if current thread can modify the thread else throw security exception.

Volatile Keyword

Problem

When a thread is reading/writing value from/on common resource repeatedly, CPU caches the object to optimize the process. So variable being read/written by one thread will not be visible to other threads.

Solution

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

Synchronization in Threads

Problem

- Data Consistency
- Thread Interference

Solution

Synchronization in Java is the capability to control the access of multiple threads to any shared resource. Synchronization is used to prevent thread interference and consistency problem. Types of Synchronization:

- Mutual Exclusion
- Inter-Thread Communication

Mutual Exclusion

Mutual Exclusion helps keep threads from interfering with one another while sharing data. It can be achieved by using the following three ways:

- By Using Synchronized Method
- By Using Synchronized Block
- By Using Static Synchronization

Synchronization is built around an internal entity known as the lock. Every object has a lock associated with it. A thread needing consistent access to an object's fields has to acquire the object's lock before accessing them and then release the lock when it's done.

Inter-Thread Communication

- Threads can communicate with each other through wait(), notify() and notifyAll().
- These are final methods defined in the Object class and can be called only from within a synchronized context.
- The wait() method halts the thread to acquire lock on the object.
- The notify() method wakes up a single thread that is waiting to acquire lock on the object. ● The notifyAll() method wakes up all threads that are waiting to acquire lock on the object.
- The wait() method causes the current thread to wait until another thread invokes the notify() or notifyAll() methods for that object.

Producer-Consumer Problem

Problem

In the producer-consumer problem, there is a Producer that is producing something and there is a Consumer that is consuming the products produced by the Producer. The producers and consumers share the same memory buffer that is of fixed-size. The job of the Producer is to generate the data, put it into the buffer and repeat. Whereas the job of the Consumer is to consume the data from the buffer. To optimize the process these two conditions must be satisfied:

- The producer should produce data only when the buffer is not full.
- The consumer should consume data only when the buffer is not empty.

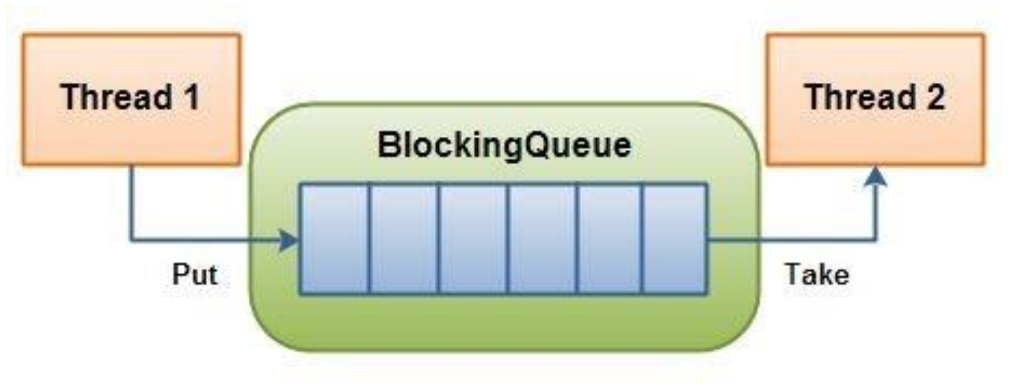
Solution

- The Producer will produce till the time buffer is not full, once the buffer is full we will make the producer thread to be in wait state.
- The consumer will consume the data from the buffer till the time buffer is not empty, once the buffer is empty we will make the producer thread to be in wait state.
- Consumer thread notifies when consuming data from the buffer so that Producer thread can start producing new items.
- And similarly Producer thread notifies when producing into the buffer so that Consumer thread can start consuming new items.

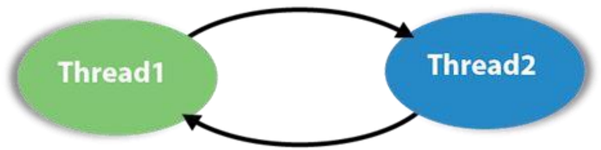
Blocking Queue

A blocking queue is a queue that blocks

- When thread (Consumer) try to remove an item from it if the queue is empty.
- When thread (Producer) try to put an item to it if the queue is already full.



Deadlock



Problem Solution

A situation, where a thread (T1) is waiting for an object’s lock, that is acquired by another thread (T2) and the thread (T2) is waiting for an object’s lock that is acquired by thread (T1). Now both the threads cannot proceed further as both need resources held by the other thread, is called Deadlock.

- Avoid Nested Locks
- Avoid Unnecessary Locks
- Using Thread Join with Timeout

Reentrant Locks

A Reentrant lock is a mutual exclusion mechanism that allows threads to re-enter into a lock on a resource (multiple times) without a deadlock situation. A thread acquires the lock by calling lock method and increases the hold count by one every time. Similarly, the hold count decreases when unlock is requested using unlock method. Therefore, a resource is locked until the counter returns to zero.

Condition

Condition object is used for wait and notify in reentrant lock. The thread will go into wait state by calling await method. And waiting thread will go into runnable again when some other thread calls signal or signalAll method.

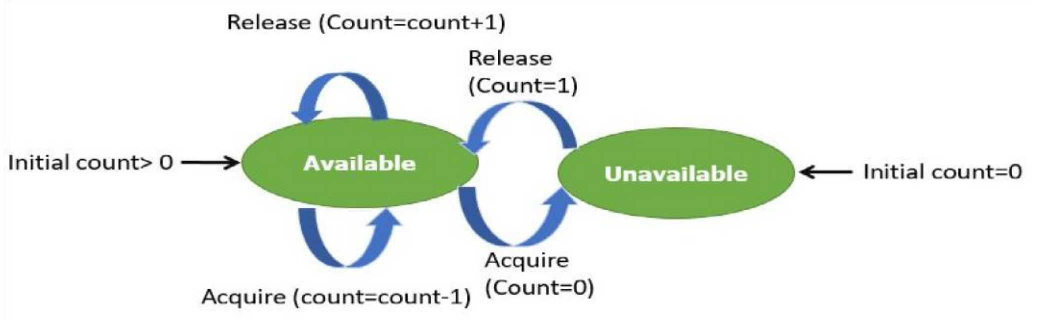
TryLock

If lock is acquired by some thread and another thread is trying to acquire the lock then it will wait till the time it is not released by first thread. So it can still cause deadlock situation. To avoid such scenario we can use tryLock method. On calling tryLock method, thread will attempt to acquire lock and return true if acquired else it will return false.

Semaphores

Problem

Currently using synchronized keyword/reentrant locks, only one thread can access the shared resource at once. If we want limited number of threads (More than 1) to access the shared resource simultaneously then we cannot do that.



Solution

A semaphore controls access to a shared resource through the use of a counter. If the counter is greater than zero, then access is allowed. If it is zero, then access is denied. While initialising the Semaphore we can provide allowed capacity. If a thread wants to access a shared resource, it will call acquire method which will decrement the counter by 1 and on releasing, the thread will call release method which will increment the counter by 1.

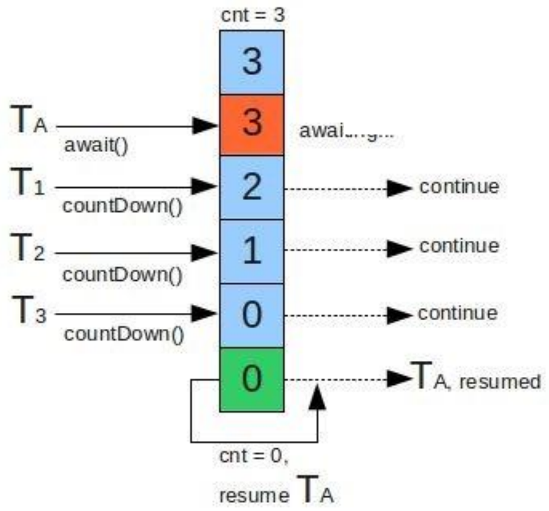
Countdown Latches

Problem

Currently if we want to wait for the thread to complete its task, we can use join method. So the thread calling join (T1) will wait for the thread (T2) on which join is called to finish its task before proceeding further. But consider a scenario where we want thread (T1) to wait until a fixed number of threads have finished its task and then proceed further.

Solution

CountDown Latch is used to make sure that a thread (T1) waits for other threads (Thread Group T) before it starts. When we create a CountDown Latch, we specify the number of threads it should wait for. All such threads (Thread Group T) are required to do count down by calling countDown method once they are completed or at a state when T1 can start its task. As soon as count reaches zero, the waiting thread (T1) starts running.



Thread Pool

Problem

- Whenever we have to create thread we have to write a common code everywhere.
- Once the task assigned to the thread is done, it is destroyed and we cannot re-use it.
- If we have to create multiple threads, it is repetitive work which we have to do every time. ● Thread handling has to be done by the developer.

Solution

Thread pool is a concept where we create multiple threads. These threads are waiting for tasks to get assigned. Whenever any task is assigned to the thread pool, if threads are available then any thread from the pool picks up the task and starts working on it or else waits. Once the thread has completed the task assigned, it will be available in the pool again for next task.

Creating Thread Pool

- Executor Interface: It has single method for submitting tasks for execution.
- ExecutorService Interface (Extends Executor): It has methods to control the progress of the tasks assigned and termination of the pool along with submitting tasks.

Thread Pool in Java

Type of Thread Pool

- Thread Pool: Task assigned is executed by one of the thread.
- Fork Join Pool: Task assigned is executed by dividing amongst the threads.
- Scheduled Thread Pool: Task assigned are executed in the future Schedule.

Configuring Thread Pool

- Core Pool Size: Initial Size of Thread Pool
- Maximum Pool Size: Maximum number of threads in a Pool ● Keep Alive Time: Amount of time thread will wait till it is destroyed.
- Unit: Time Unit for Keep Alive Time.
- Work Queue: Data structure used to store waiting tasks.
- Thread Factory: Factory used to create new thread.
- Rejected Execution Handler: Code to execute when thread pool rejecting any task.

Executors

It is a utility class using which we can create thread pool without specifying all the configurations. It has different methods that create thread pools of different types using commonly used configurations.

Methods

- void shutdown(): Shutdown Thread Pool for any new tasks.
- List<Runnable> shutdownNow(): Attempt to stop all the actively executing tasks, halts the execution of all the waiting/sleeping tasks. And returns a list of tasks that are waiting for execution.
- boolean isShutdown(): Returns if the thread pool is shutdown or not.
- boolean isTerminated(): Returns if all the tasks are completed for thread pool post shutdown.
- boolean awaitTermination(long timeout, TimeUnit unit): Blocks until all tasks have been completed its execution post shutdown or time-out whichever is earlier.
- void execute(Runnable task): Submits runnable task to execute in thread pool.
- Future<?> submit(Runnable task): Submits runnable task to execute in thread pool and return future to manage the task.
- Future<T> submit(Runnable task, T result): Submits runnable task to execute in thread pool and return the value passed as an argument through future.
- Future<T> submit(Callable<T> task): Submits callable task to execute in thread pool and return the result through future.
- List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks) / invokeAll(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit): Submits collection of callable tasks to execute in the thread pool within a timeout if specified.
- T invokeAny(Collection<? extends Callable<T>> tasks) / invokeAny(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit): Submits collection of callable tasks to execute any one of the tasks and returns the output from the task that is completed successfully (Not throwing exception) within a timeout if specified. Rest of the tasks get cancelled.

Callable And Future

Problem

Currently we cannot return any value from the thread run method once the execution of run method is complete.

Solution

In Thread pool, we can create tasks by implementing callable interface instead of runnable interface when we want to return any value. In case of callable, we need to implement call method which can return result of the type mentioned while creating task. Returned result can be accessed through future. (Cannot be used for simple Thread creation)

Future

Future interface represents a future result of an asynchronous computation. Whenever we are submitting any task to Thread pool it returns a Future. Using Future we can check the result as well as manage execution of task.

Method

cancel(boolean mayInterruptIfRunning): To cancel the non completed task (if mayInterruptIfRunning is true then cancel always otherwise cancel only if not in progress). isCancelled() : To check if the task is cancelled.

isDone(): To check if the task is completed (Finished due to completion/Exception or Cancelled) get() / get(long timeout, TimeUnit unit): To get the result for the task (Wait till the time the task is completed) or timeout is specified.

Interrupting a Thread

If any thread is in sleeping or waiting state, we can interrupt the execution of that thread by calling interrupt method on the thread. It sets interrupted flag of the thread which is checked while waking up the thread from waiting or sleeping state. If we check the flag using interrupted or isInterrupted method while execution then we can interrupt the running thread as well. Similarly using shutdownNow() we can interrupt tasks in the thread pool.

Thank You