Welcome

Java Concurrency Environment

Copyright Notice

Java Concurrency Environment

Prakash Badhe

prakash.badhe@vishwasoft.in

Process and Threads with Java

- Thread Class
- Runnable interface
- ThreadGroup
- Synchronized methods and synchronized blocks
- The volatile variable
- Inter-Thread communication

Java5 advanced concurrency

- New high-level concurrency features introduced with version 5.0 of the Java platform.
- **Concurrency Infrastructure** : Most features are implemented in the new java.util.concurrent package.
- **Concurrency Collections** : The new concurrent data structures in the Java Collections Framework.

Executors

- Executors define a high-level API for launching and managing threads.
- Executor implementations provided by java.util.concurrent provide thread pool management suitable for large-scale applications.
- For large-scale applications, separate thread management and creation from the rest of the applications for better usage and performance.
- The objects that encapsulate these functions at high level are *executors.*

Executor Interfaces

- Three executor interfaces:
- Executor: a simple interface that supports launching new tasks.
- ExecutorService: a subinterface of Executor, which adds features that help manage the lifecycle, both of the individual tasks and of the executor itself.
- ScheduledExecutorService : a subinterface of ExecutorService, supports future and/or periodic execution of tasks.
- The variables that refer to executor objects are declared as one of these three interface types, not with an executor class type.

Executor Interface

- The Executor interface provides a single method, execute, designed to be a drop-in replacement for a common thread-creation idiom.
- If runnableObj is a Runnable object,
  and executor is an Executor object , then replace
- (new Thread(runnableObj)).start();
- with
- **executor.execute(runnableObj);**

ExecutorService

- The ExecutorService interface supplements execute with a similar, but more versatile submit method.
- Like execute, *submit* accepts Runnable objects, but also accepts Callable objects, which allow the task to return a value.
- The submit method returns a *Future* object, which is used to retrieve the Callable return value and to manage the status of both Callable and Runnable tasks.
- ExecutorService also provides methods for submitting large collections of Callable objects.
- ExecutorService provides a number of methods for managing the shutdown of the executor. To support immediate shutdown, tasks should
handle interrupts correctly.

ScheduledExecutorService

- The ScheduledExecutorService interface supplements the methods of its parent ExecutorService with schedule, which executes a Runnable or Callable task after a specified delay.
- In addition, it defines scheduleAtFixedRate and scheduleWit hFixedDelay, which executes specified tasks repeatedly, at defined intervals.

Thread Pool

- Most of the executor implementations in java.util.concurrent use *thread pools*, which consist of *worker threads*.
- This kind of thread exists separately from the Runnable and Callable tasks it executes and is often used to execute multiple tasks.

Thread Pool Management

- One common type of thread pool is the *fixed thread pool*.
- This type of pool always has a specified number of threads running; if a thread is somehow terminated while it is still in use, it is automatically replaced with a new thread.
- Tasks are submitted to the pool via an internal queue, which holds extra tasks whenever there are more active tasks than threads.
- Very similar to DataSource connections managed by the DataSource Pool Manager along with the database driver.

Create a thread pool

- Executors. newFixedThreadPool(): Create a fixed thread pool.
- newCachedThreadPool() method creates an executor with an expandable thread pool. This executor is suitable for applications that launch many short-lived tasks.
- The newSingleThreadExecutor method creates an executor that executes a single task at a time.

Fork/Join Framework

- The fork/join framework is an implementation of the *ExecutorService* interface to take advantage of multiple processors.
- It is designed for work that can be broken into smaller pieces recursively.
- The goal is to use all the available processing power to enhance the performance of the application.

Fork/Join usage

- The fork/join framework distributes tasks to worker threads in a thread pool.
- The fork/join framework is distinct since it uses a *work-stealing* algorithm.
- Worker threads that run out of things to do can steal tasks from other threads that are still busy.
- The *ForkJoinPool* class, an extension of the AbstractExecutorService class,implements the core work-stealing algorithm and can execute *ForkJoinTask* processes.

Callable interface

- The existing implementations Thread and Runnable don't return any value with run() methods.
- The Callable defines a task that returns a result and may throw an exception.
- Implementers define a single method with no arguments called call.
- The Callable interface is similar to Runnable, both are designed for classes whose instances are potentially executed by another thread.
- A Runnable, however, does not return a result and cannot throw a checked exception.
- The Executors class contains utility methods to convert from other common forms to Callable classes.

Callable Implementations

- A class can implement the Callable with call method.
- Since this is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference, to generate dynamic implementations.

Race Condition

- A *race condition* is a special condition that may occur inside a critical section.
- A *critical section* is a section of code that is executed by multiple threads and where the sequence of execution for the threads makes a difference in the result of the concurrent execution of the critical section.
- This the critical section is said to contain a race condition.

Prevent Race condition

- To prevent race conditions from occurring must make sure that the critical section is executed as an atomic instruction.
- That means that once a single thread is executing it, no other threads can execute it until the first thread has left the critical section.

Synchronization of code

- Race conditions can be avoided by proper thread synchronization in critical sections.
- Thread synchronization can be achieved using a **synchronized block of Java code**. Thread synchronization can also be achieved using other synchronization constructs like **locks** or atomic variables like *java.util.concurrent.atomic.AtomicInteger.*

Lock

- A lock is a tool for controlling access to a shared resource by multiple threads.
- A lock is a thread synchronization mechanism like synchronized blocks except locks can be more sophisticated than Java's synchronized blocks.
- Locks (advanced synchronization mechanisms) are created using synchronized blocks, so it is not like we can get totally rid of
  the synchronized keyword.

Lock interface

- From Java 5 the package java.util.concurrent.locks contains several lock implementations.
- 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.

Lock methods

- Void lock()
- Void lockInterruptibly()
- Condition newCondition(): new Condition instance that is bound to this Lock instance.
- boolean tryLock()
- boolean tryLock(long time, TimeUnit unit)
- Public void unlock()

Lock implementations

- **ReentrantLock**
- **ReentrantReadWriteLock.ReadLock**
- **ReentrantReadWriteLock.WriteLock**
- The locking operations work just like synchronized but are more optimized.

Locks vs. Synchronized Blocks

- A synchronized block makes no guarantees about the sequence in which threads waiting to entering it are granted access.
- No parameters passed to the entry of a synchronized block. Thus, having a timeout trying to get access to a synchronized block is not possible.
- The synchronized block must be fully contained within a single method. A Lock can have it's calls to lock() and unlock() in separate methods.

Reentrant Lock

- Synchronized blocks in Java are reentrant.
- This means, that if a Java thread enters a synchronized block of code, and thereby take the lock on the monitor object the block is synchronized on, the thread can enter other Java code blocks synchronized on the same monitor object.

ReadWriteLock

- It allows multiple threads to read a certain resource, but only one to write it, at a time.
- Multiple threads can read from a shared resource without causing concurrency errors.
- The concurrency errors first occur when reads and writes to a shared resource occur concurrently, or if multiple writes take place concurrently.

Locking Rules

- **Read Lock :** If no threads have locked
  the ReadWriteLock for writing,
  and no thread have requested a write lock (but not yet obtained it).
  Thus, multiple threads can lock the lock for reading.
- **Write Lock** If no threads are reading or writing.
  Thus, only one thread at a time can lock the lock for writing.
- ReadWriteLock Implementations**:**
  ReentrantReadWriteLock

Sync vs. Async

- **Synchronous** means things are getting done in sequential order. This means if one process is going on, other one won't start unless already running process is finished.
- **Asynchronous** means things won't wait for process to finish and can start running new process or proceed with further execution.
- *Synchronous methods blocks the calling threads in case the method takes long time to return.*
- *In Asynchronous methods the calling threads never gets blocked, since the tasks are running parallel.*

Future

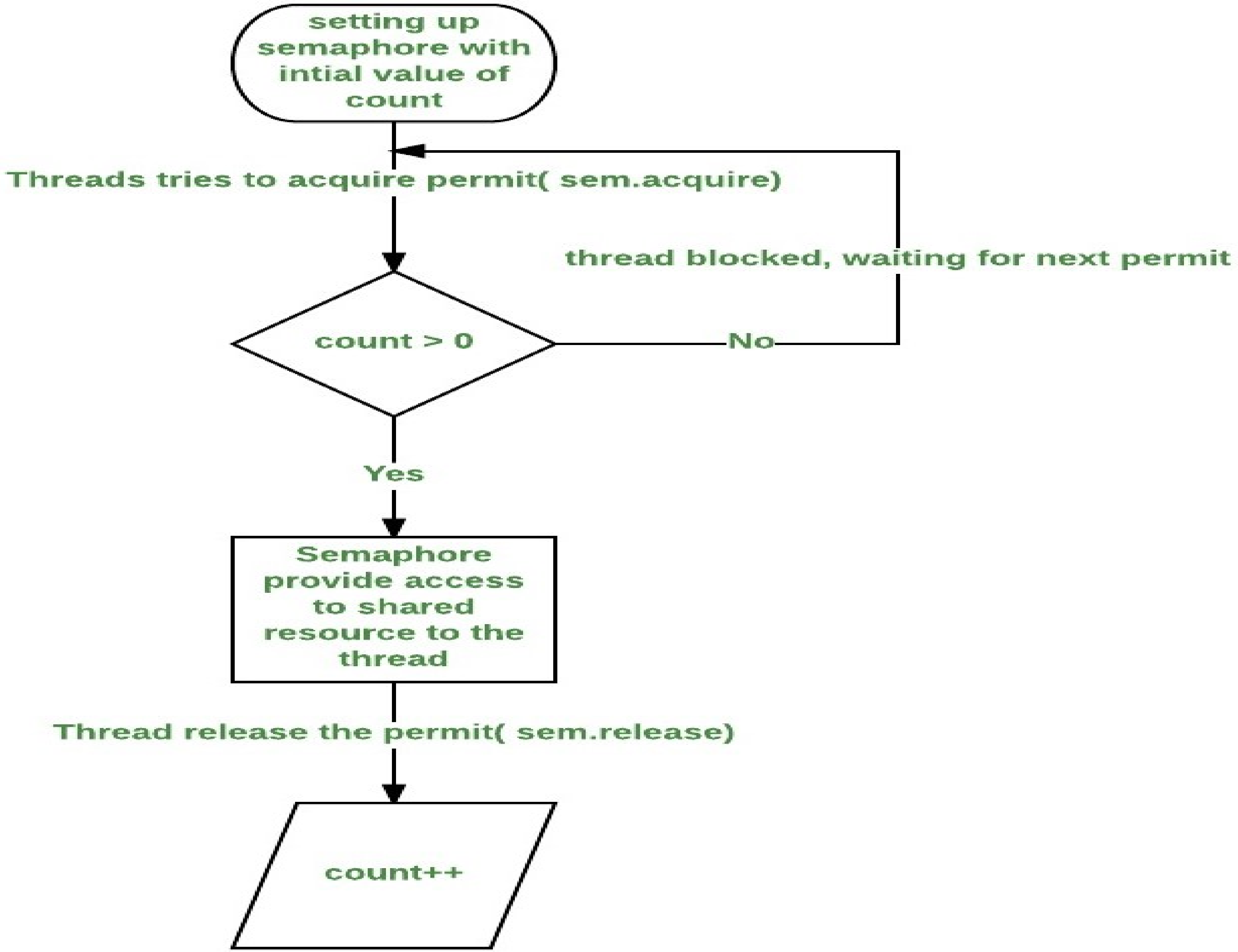
- Future is an abstraction for asynchronous computation.
- It represents the result of the computation, which might be available at some point: either a computed value or an exception.
- Most of the methods of the ExecutorService use Future as a return type.
- It exposes methods to examine the current state of the future or block until the result is available.

CountDownLatch

- Allows one or more threads to wait until a set of operations being performed in other threads completes.
- The *CountDownLatch* is initialized with a count.
- Threads may call await() to wait for the count to reach 0.
- Other threads (or the same thread) may call countDown() to reduce the count.
- Not reusable once the count has reached 0.
- Used to trigger an unknown set of threads once some number of actions has occurred.

Semaphore

- 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.
- What the counter is counting are permits that allow access to the shared resource.
- To use a semaphore, the thread that wants access to the shared resource tries to acquire a permit first.


setting up semaphore with intial value of count
Threads tries to acquire permit(sem.acquire)
thread blocked, waiting for next permit
count > 0
No
Yes
Semaphore provide access to shared resource to the thread
Thread release the permit(sem.release)
count++

Semaphore Usage

- Conceptually, a semaphore maintains a set of permits.
- Each acquire() blocks until a permit is available, and then takes it.
- Each release() adds a permit, potentially releasing a blocking acquirer.
- However, no actual permit objects are used.
- The Semaphore just keeps a count of the number available and acts accordingly.
- Semaphores are often used to restrict the number of threads than can access some (physical or logical) resource.

Mutex

- In a multithreaded application, two or more threads may need to access a shared resource at the same time, resulting in unexpected behavior.
- Examples of such shared resources are data-structures, input-output devices, files, and network connections.

CompletableFuture

- CompletableFuture is an abstraction for async computation.
- Unlike plain Future, where the only possibility to get the result is to block, register callbacks to create a pipeline of tasks to be executed when either the result or an exception is available.
- Either during creation (via CompletableFuture#supplyAsync/runAsync) or during adding callbacks (*async family's methods), an executor, where the computation should happen, can be specified (if it is not specified, it is the standard global ForkJoinPool#commonPool).