Kevin Chen

2/17/17

Managing Threads in Java Notes

**Java Threads**

What is a Thread?

* Every Java program has a default main thread (the main method)
* A thread is an independent path of code execution.
* Many threads can run concurrently in a Java program
* Threads can be used to perform time-intensive tasks
* Runnables are objects that encapsulate code sequences, so each thread executes a runnable object.
* Threads can initiate an asynchronous task (can run concurrently instead of sequentially)
* The JVM gives each thread its own private JVM stack (own private area of memory), preventing threads from interfering with each other. The stack holds local variables and keeps track of next instructions and calls methods.
* Supports threads through java.lang.Thread class and java.lang.Runnable interface
* Threads are either daemon or non-daemon. Daemon threads still runs after the program finishes, but they don’t stop the JVM from ending when the program stops running. By default, threads are non-daemon
* Example of daemon thread: Java garbage collection. Create a daemon thread by calling setDaemon(true). Main thread is a non-daemon thread. The program ends when non-daemon threads have died.
* Thread lifecycle:
  + Start with New Thread(). After calling start(), it enters Runnable, and then the code in the run() method is executed (it is now Running its task).
  + A thread may go to the Waiting state by calling wait(). The thread transitions back to the runnable state only when another thread notifies (notify()) the waiting thread to continue executing.
  + A thread may go the timed waiting state (the sleep state) for a specified interval of time by calling sleep().
  + Thread is in the Dead state when terminated.
  + Sleep vs. Wait: sleep() is static method (performs the operation on the currently running thread). currentThread() returns the current thread. wait() method is called upon an object.

How to create a thread

* Two main ways to create a thread.
  + First way is to define a new class as a subclass of the Thread class. Must override the run() method. Then an instance of this subclass can be allocated and started.
  + Second way is to implement the Runnable interface. Can do this by creating a new class that implements Runnable OR (create a Runnable object and use a Thread constructor). Still must override the run() method.
* Examples: (HelloRunnable implements Runnable, HelloThread implements Thread)
  + Using runnable: (new Thread(new HelloRunnable())).start();
  + Using thread: (new HelloThread()).start();
  + Using anonymous classes: Runnable r = new Runnable() { /\*override run() method \*/}; Thread t = new Thread(r, “Name-of-thread”); t1.start();
  + Using lambda expressions: Runnable r = () -> /\*method body for the overridden run() method\*/ Thread t = new Thread(r); t.start();
* If you don’t provide a name for the thread, the default name is Thread-*n*, where *n* is the next number (starting from zero) that hasn’t been used yet.
  + Get the name of the thread using the method getName() on the thread.

Thread States

* Each object has a state, which includes the name, alive/dead, execution state, priority, and daemon/non-daemon status
* Thread states: NEW (created but not started), RUNNABLE (thread executing in JVM), BLOCKED (waiting for a monitor to be unlocked), WAITING (waiting to be notified to continue), TIMED\_WAITING (waiting with a time limit), TERMINATED (a thread that has completed execution).
* (Need to throw InterruptedException when we use a static method for a thread class)
* Return whether a thread is alive using the isAlive() method on the thread. Can get the thread’s state with getState() and get the thread’s priority with getPriority().
* Thread.sleep(int *milliseconds*) puts the main thread to sleep for the specified time
* (Default priority is 5)

Advanced thread tasks

* Thread class has static and non-static methods for thread management. Static methods affect the current thread; non-static methods operate on the calling thread.
* Static methods
  + activeCount(): returns estimated number of active threads
  + currentThread(): returns reference to current thread
  + enumerate(Thread[] tarr): list of active threads. (tarr is an out parameter)
  + sleep(long millis) – sleep or stop execution of the currently running thread
* Non-static methods
  + join() – causes the current thread to wait for the thread to die. If you pass in a time value, that’s used to specify the max wait time for the thread to die.
  + interrupt() – stops a thread

**Synchronization**

Problems with Threads

* Thread interaction can cause problems, making the application thread-unsafe.
* Problems include race conditions, data races, and cached variable problems.
* Threads can increase complexity. Reliance on synchronization can lead to performance issues, which affects an application’s scalability.
* Race condition: correctness of a computation depends on the relative timing, or the interleaving of multiple threads by the scheduler.
  + This wouldn’t occur for local variables because each thread has its own copy of the local variables
* Data race: two or more threads access the same memory location concurrently
  + Can occur when one thread starts, get paused by the scheduler, another thread sneaks in and changes the value, and then resume the first thread.

Synchronization Explained

* Helps resolve race conditions, data races, and cached variable problems.
* Prevents two threads from simultaneously accessing a critical section.
* Ensures threads can safely update shared variables
* Can be applied to methods or code blocks by labelling them with the “synchronized” keyword.
* Synchronized code is known as critical sections
* JVM supports it via monitors, which enter and exit JVM instructions.
  + Every Java object is associated with a monitor, which is a mutual exclusion construct that prevents multiple threads from concurrently executing in a critical section.
  + Before a thread can enter a critical section, it’s required to lock the monitor.
  + If the monitor is already locked, the thread is locked until it is unlocked.
  + When a thread locks a monitor, the variables stored in main memory are copied in the thread’s working memory (aka local memory or cache memory). This ensures that the thread will work with the most recent values of these variables.
  + When done, the thread writes values back to main memory

Synchronized Methods

* One way to use synchronization is to use a synchronized method.
* Add the “synchronized” keyword to the method header, which prevents two threads from accessing same critical code. Example: public static synchronized int getID() {}
* Also possible to add a lock associated with an object by adding the “synchronized” keyword to one of its member functions.

Synchronized Block

* Another way to use synchronization is to use synchronized block.
* A synchronized block is identified by a special header, which identifies the object for the lock. Syntax: synchronized(object) { /\* statements \*/}
  + In the above example, the lock is associated with the object in the parenthesis. The curly brackets identify the critical section of code.
  + Any statements in the synchronized block will be synchronized.

Wait and Notify API

* Java has an API that supports communication between threads. This allows a thread to wait until another thread does/gives something the first thread needs and notifies the first thread to resume.
* The java.lang.object class provides a wait and notify API with 3 different wait method. wait() causes the thread to wait until another thread invokes the notify or notify all method for this object. wait(long mills) will continue if the time passed in has elapsed, or it has been notified. wait(long mills, long nano) allows more precision for the time.
* Notify methods: notify() wakes up a single arbitrary thread that’s waiting on the object’s monitor. notifyAll() wakes up all threads waiting on the object’s monitor.
* This API uses the object’s condition queue, which stores the threads that are in a wait state for that object. The waiting threads are known as the wait set.
  + The condition queue is tightly bound to an object’s lock, so all of the methods must be called from within a synchronized context. (The current thread must be the owner of the object’s monitor).
  + Otherwise, it will invoke the IllegalMonitorStateException.
* Deadlock situation: when 2 or more threads are blocked forever waiting for each other. Example: Thread 1 locks Resource 2 and will unlock it once it has access to Resource 1. Thread 2 locks Resource 1 and will unlock it once it has access to Resource 2. No progress can be made for either threads.

**Concurrency**

Concurrency Utilities

* Offer a powerful and extensive framework for threads.
  + Includes a high-performance thread pool, a framework for asynchronous tasks, and a host of collection classes optimized for concurrent access
* Advantages: reduced programming effort, increased performance, increased reliability, improved maintainability.
* Include:
  + Task scheduling framework: e.g. executors, which standardizes the invocation, or the calling of, the scheduling, execution, and control of asynchronous tasks according to a set of execution policies
  + Fork/join framework: run a large number of tasks using a pool of worker threads
  + Concurrent collections: newQueue, BlockingQueue, and BlockingDeque interfaces. High-performance concurrent implementations of map, list, and queue
  + Atomic variables: atomically manipulate variables/references, providing high-performance, atomic arithmetic and compareAndSet methods.
  + Synchronizers: general-purposes synchronization classes that facilitate coordination between threads
  + Locks: while lock is built into the Java language through the synchronize keyword, there are lot of limitations to built-in monitor locks. The locks package provides a high-performance lock implementation, and supports specifying a timeout when attempting to acquire a lock
  + Nanosecond granularity timing: system.nanotime() method enables access to a nanosecond granularity time source for making relative time measurements.

Executors

* Defines a high-level API to manage threads. Provides thread pool management. Separates thread creation and management from main. Uses worker threads to minimize overhead.
* Three types of executor interfaces: simplifies the creation of threads.
  + Executor – supports launching new tasks
  + ExecutorService – adds features to manage lifecycle
  + ScheduledExecutorService – supports future tasks
* Creating an executor with n working threads
  + ExecutorService executor = Executors.newFixedThreadPool(int numOfThreads);
* Using an executor
  + *nameofexecutorobject*.execute(Runnable runnable);
* If there needs to be more threads than are available in the executor, then any working threads that become available will run what’s needed after finishing.
* End all your tasks at the end
  + *nameofexecutorobject*.shutdown();
* *nameofexecutorobject*.isTerminated() returns a boolean that says if all the threads in the executor are terminated.

Synchronizers

* Java provides the synchronized keyword, but it can be difficult to correctly write synchronized code, so there are high level synchronizes.
* Examples include: latches, barriers, semaphores, and exchangers.
* One type of latch is a CountDownLatch, which causes one or more threads to wait at a gate until another thread opens the gate, after which the other threads can continue.
  + It contains a count value and logic to decrement the count until it reaches zero.
* CountDownLatch Methods
  + void await() forces the calling thread to wait until the latch has counted down to zero.
  + boolean await(long timeout, TimeUnit u) forces the calling thread to wait until the latch has counted down to zero, or the specified time out value in unit TimeUnits has expired.
  + void countdown() decrements the count
  + long getCount() returns the current count.
  + String toString() identifies the latch and the state.

Locking framework

* Improves on intrinsic synchronization with lock polling, timed waits, etc.
* Includes these types of locks: Lock, ReentrantLock, Condition, ReadWriteLock, and ReentrantReadWriteLock.
* Lock interface methods
  + void lock() acquires the lock. When the lock isn’t available, the calling thread is forced to wait until it becomes available.
  + void lockinterruptibly() acquires the lock unless the calling thread is interrupted. When the lock isn't available the calling thread is forced to wait until it becomes available or the thread is interrupted.
  + condition newCondition() returns a new condition instance that’s bound to this lock instance
  + boolean tryLock() acquires a lock when it’s available at the time this method is invoked. Returns whether lock is acquired.
  + boolean tryLock(long time, TimeUnit u) acquires the lock when it's available within the specified wait time, measured in the TimeUnit units.
  + void unlock releases the lock.
* This framework is more flexible.
  + Allows chain locking.
* Basic locking program structure:

ReentrantLock lock1 = new ReentrantLock();

lock1.lock

try {

//Perform critical section that requires the lock

} catch (Exception e) {}

finally { lock1.unlock();} // ALWAYS release the lock.

Next Steps

* New concurrency utilities are being introduced in future releases of Java. Thus, pay attention to future releases of Java and any additional utilities for managing multithreaded applications.