Java Threads Introductions

April 18, 2024

1 Java Threads

In Java, a thread is a lightweight subprocess or the smallest unit of processing. It is a separate path of execution. Threads are independent, and if there occurs an exception in one thread, it doesn't affect other threads. It shares a common memory area.

Java is a multi-threaded programming language which means we can develop multi-threaded program using Java. A multi-threaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution.

Here are some key points about Java threads:

1. **Multitasking**: Threads allow concurrent execution of two or more parts of a program for maximum utilization of CPU.
2. **Thread Class and Runnable Interface**: Threads can be created by extending the `Thread` class and overriding its `run()` method, or by implementing the `Runnable` interface and implementing the `run()` method.
3. **Thread Lifecycle**: A thread goes through various stages in its lifecycle, such as New, Runnable, Running, Non-Runnable (Blocked), and Terminated.
4. **Thread Priority**: Each thread has a priority which helps the operating system determine the order in which threads are scheduled for running.
5. **Synchronization**: Java provides a way of creating threads and synchronizing their task by using synchronized blocks to avoid thread interference and consistency problems.
6. **Inter-thread Communication**: Inter-thread communication allows synchronized threads to communicate with each other.

Remember, threads can be very useful for implementing tasks that can run in parallel, but they also require careful programming to avoid issues with shared data, such as race conditions or deadlocks.

2 What are the way to create threads?

In Java, there are two main ways to create threads:

1. By extending the `Thread` class
2. By implementing the `Runnable` interface

Here's a basic example of each:

Extending the Thread class:

[ ]:
```
class MyThread extends Thread {
  public void run(){
     System.out.println("Thread created by extending Thread class.");
  }
}

public class Main {
  public static void main(String args[]) {
     MyThread t = new MyThread();
     t.start();
  }
}
```

Implementing the Runnable interface:

[ ]:
```
class MyRunnable implements Runnable {
  public void run(){
     System.out.println("Thread created by implementing Runnable interface.");
  }
}

public class Main {
  public static void main(String args[]) {
     Thread t = new Thread(new MyRunnable());
     t.start();
  }
}
```

In both cases, the `run()` method is where you define the code that the thread will execute. To start the thread, you call the `start()` method, which in turn calls the `run()` method.

3 How can I control the execution of threads in Java?

In Java, you can control the execution of threads using several methods provided by the `Thread` class. Here are some of them:

1. **start():** This method is used to start the execution of a thread. The JVM calls the `run()` method on the thread.

[ ]:
```
Thread t = new Thread(new MyRunnable());
t.start();
```

2. **sleep(long millis):** This static method causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.

[ ]:
```
try {
    Thread.sleep(2000); // Pause for 2 seconds
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

3. **join():** This method allows one thread to wait until another thread completes its execution. If `t` is a Thread object whose thread is currently executing, then `t.join()` will make sure that `t` is terminated before the next instruction is executed by the program.

[ ]:
```
Thread t1 = new Thread(new MyRunnable());
t1.start();
try {
    t1.join(); // Wait for t1 to finish
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

4. **yield():** It's a static method that causes the currently executing thread object to temporarily pause and allow other threads to execute.

[ ]:
```
Thread.yield();
```

5. **setPriority(int newPriority):** Changes the priority of this thread. The possible range is from `Thread.MIN_PRIORITY` (a constant of 1) to `Thread.MAX_PRIORITY` (a constant of 10).

[ ]:
```
t.setPriority(Thread.MAX_PRIORITY);
```

6. **interrupt():** Interrupts this thread. In case the thread is waiting or sleeping, it wakes it up and throws `InterruptedException`.

[ ]:
```
t.interrupt();
```

Remember, the exact scheduling of threads can depend on the Java Virtual Machine (JVM) and the operating system's policies, so you can't always predict the exact order in which threads will execute.

4 Concurrency Problems?

Concurrency problems arise when multiple threads or processes access shared data and they try to change it at the same time. This can lead to unexpected results if the threads are not properly synchronized. Here are some common concurrency problems:

1. **Race Condition:** A race condition occurs when two or more threads can access shared data and they try to change it at the same time. As a result, the values of variables may be unpredictable and vary depending on the timings of context switches of the processes.

2. **Deadlock:** Deadlock is a situation where two or more threads are blocked forever, waiting for each other. This usually happens when multiple threads need the same locks but obtain them in different order.

3. **Starvation:** Starvation describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress. This happens when shared resources are

made unavailable for long periods by “greedy” threads.

4. **LiveLock:** A thread often acts in response to the action of another thread. If the other thread’s action is also a response to the action of another thread, then livelock may result. As with deadlock, livelocked threads are unable to make further progress.
5. **Priority Inversion:** A scenario where a lower priority thread holds a lock needed by a higher priority thread. The higher priority thread is blocked, effectively inverting the priority of the two threads.

To avoid these problems, synchronization techniques such as locks, semaphores, monitors, or condition variables can be used. These techniques can help ensure that only one thread at a time uses a shared resource.

5 What are some common synchronization techniques used to prevent concurrency problems in Java?

Java provides several mechanisms for synchronizing access to shared resources to prevent concurrency problems:

1. **Synchronized Blocks:** The simplest way to synchronize access in Java is by using synchronized blocks. A synchronized block ensures that only one thread at a time can execute a particular section of code.

[ ]:
```
synchronized(object) {
    // Code here is protected by the lock on the object
}
```

2. **Synchronized Methods:** You can also use the `synchronized` keyword to mark an entire method as synchronized. This works the same way as a synchronized block that covers an entire method body.

[ ]:
```
public synchronized void myMethod() {
    // Code here is protected by the lock on this object
}
```

3. **Locks:** The `java.util.concurrent.locks` package provides a more flexible locking mechanism with `ReentrantLock`. This allows more sophisticated behaviors such as try-lock and timed lock.

[ ]:
```
Lock lock = new ReentrantLock();
lock.lock();
try {
    // Access shared resources
} finally {
    lock.unlock();
}
```

4. **Semaphores:** A semaphore is a thread synchronization construct that can be used to protect an arbitrary number of execution instances or resources. It’s more flexible than a lock as it

can control access to more than one thread at a time.

[ ]:
```java
Semaphore semaphore = new Semaphore(3); // Allow 3 concurrent threads
semaphore.acquire();
try {
    // Access shared resources
} finally {
    semaphore.release();
}
```

5. **CountDownLatch:** A synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes.

[ ]:
```java
CountDownLatch latch = new CountDownLatch(3); // Wait for 3 threads to complete
latch.countDown(); // In each thread, after completion
latch.await(); // Wait for all threads to complete
```

6. **CyclicBarrier:** A synchronization aid that allows a set of threads to all wait for each other to reach a common barrier point.

[ ]:
```java
CyclicBarrier barrier = new CyclicBarrier(3); // Wait for 3 threads to reach a␣
  ↪common point
barrier.await(); // In each thread, when it reaches the barrier point
```

Remember, synchronization can lead to thread contention, which occurs when two or more threads try to access the same resource simultaneously and cause the Java runtime to execute one or more threads more slowly, or even halt their execution. So, it's important to use these tools judiciously to balance between concurrency and thread contention.

6 What is the difference between synchronized keyword and ReentrantLock in Java?

Both `synchronized` keyword and `ReentrantLock` are used to provide synchronization in Java, but there are several differences between them:

1. **Flexibility:** `ReentrantLock` is more flexible than `synchronized`. With `ReentrantLock`, you can make lock fairness, which means the lock would be granted to the longest-waiting thread. Also, `ReentrantLock` provides a method `tryLock()`, which tries to acquire the lock and returns a boolean based on whether the lock was acquired.

2. **Condition Variables:** `ReentrantLock` provides a `Condition` interface, which provides methods like `await()`, `signal()`, and `signalAll()` similar to `wait()`, `notify()`, and `notifyAll()` of `synchronized`. But `Condition` supports multiple wait-notify pools per lock, which is not possible with `synchronized`.

3. **Lock Interruption:** `ReentrantLock` provides a mechanism to interrupt the thread waiting for the lock, which is not possible with `synchronized`.

4. **Manual Unlock:** With `ReentrantLock`, you have to manually release the lock using the `unlock()` method. If you forget, the lock will remain held. With `synchronized`, the lock is

automatically released when control leaves the synchronized block or method.

Here is an example of using `ReentrantLock`:

```
[ ]: ReentrantLock lock = new ReentrantLock();
     lock.lock();
     try {
         // access shared resources
     } finally {
         lock.unlock(); // make sure to unlock in finally block to avoid deadlock
     }
```

And here is an example of using `synchronized`:

```
[ ]: synchronized(this) {
         // access shared resources
     } // lock is automatically released here
```

In general, `synchronized` is easier to use and sufficient for most cases. `ReentrantLock` should be used when you need more control over the locking mechanism.

7 What are some best practices for handling concurrency in Java?

Handling concurrency correctly is crucial in multi-threaded programming. Here are some best practices for handling concurrency in Java:

1. **Minimize the scope of synchronization:** Synchronization can lead to thread contention, which can degrade performance. Therefore, you should only synchronize the code that absolutely needs to be synchronized.

2. **Use higher-level concurrency utilities:** Java provides higher-level concurrency utilities like `ExecutorService`, `Semaphore`, `CyclicBarrier`, `CountDownLatch` etc. in the `java.util.concurrent` package. These utilities can simplify the management of threads and improve readability and maintainability of your code.

3. **Avoid using `Thread.stop()`, `Thread.suspend()`, and `Thread.resume()`:** These methods are deprecated because they are inherently unsafe. Use other mechanisms like interruption or a volatile boolean field to stop a thread.

4. **Prefer `Atomic` classes over `synchronized`:** The `java.util.concurrent.atomic` package provides classes that support lock-free and thread-safe programming on single variables. These classes can be more scalable than using `synchronized` in some cases.

5. **Avoid using `volatile` for non-boolean variables:** The `volatile` keyword guarantees visibility and ordering of reads and writes to a variable, but it doesn't provide atomicity for compound operations.

6. **Always release resources in `finally` block:** If you acquire a lock, always release it in a `finally` block to ensure that the lock is released even if an exception is thrown.

7. **Be aware of the `java.lang.ThreadLocal` class:** `ThreadLocal` can be used to maintain thread confinement, which is a powerful way to achieve thread safety.

8. **Test your code:** Concurrency bugs can be difficult to detect and reproduce. Use tools like FindBugs, PMD, or Checkstyle to detect potential concurrency bugs. Also, write multi-threaded unit tests and stress tests to ensure your code works correctly under concurrent conditions.

9. **Keep up-to-date with the latest in Java concurrency:** Java's concurrency utilities have evolved significantly over the years. Make sure you're familiar with the latest tools and best practices.

8 Thank You!