Multithreading in Java

Ali Saab

Contents

1 Processes and Threads

In the context of computer programming, a process and a thread are both units of execution, but they have some fundamental differences.

Process

A process can be thought of as an instance of a program running on a computer. It is an independent entity that has its own memory space, program code, and system resources. Each process is isolated from other processes, meaning they cannot directly access each other's memory or resources. Processes are managed by the operating system and can be created, scheduled, and terminated independently. Examples of processes include running multiple instances of a program or running different programs simultaneously.

Thread

A thread is a unit of execution within a process. Unlike a process, a thread does not have its own memory space or system resources. Threads within the same process share the same memory and resources, allowing them to communicate and cooperate with each other more easily. Threads are scheduled by the operating system's thread scheduler, and they can be created, executed, and terminated independently. Multiple threads can run concurrently within a process, enabling parallel execution of tasks. Threads are lighter-weight compared to processes and have lower overhead.

Difference between a Process and a Thread

- Processes are independent entities, while threads are subsets of processes.
- Each process has its own memory space, while threads share the same memory space within a process.
- Processes do not directly share memory or resources with each other, while threads within the same process can access shared memory and resources.
- Processes are more isolated and have higher overhead, whereas threads have lower overhead and are more lightweight.
- Processes are managed by the operating system, while threads are managed by the process they belong to.

2 Main Thread

In Java, the main thread refers to the thread that executes the `main` method, which is the entry point of a Java program. When a Java program starts, the Java Virtual Machine (JVM) automatically creates the main thread and begins executing the code inside the `main` method.

The main thread is responsible for executing the program sequentially, starting from the `main` method and progressing through subsequent statements and method calls. It is the primary thread of execution in a Java program.

The main thread can perform various tasks, such as initializing variables, invoking other methods, creating additional threads, and coordinating the overall flow of the program. It continues its execution until the `main` method completes or until the program is terminated explicitly.

It's worth noting that the main thread is just one of potentially many threads that can exist in a Java program. Additional threads can be created explicitly using the `Thread` class or higher-level concurrency utilities. These threads can run concurrently with the main thread, allowing for parallel execution and multitasking within the program.

In a Java program, the main thread can end before other threads in certain scenarios. By default, the main thread executes the code inside the `main` method sequentially, and once it reaches the end of the `main` method, it can terminate, regardless of whether other threads are still running.

If there are additional threads created and started within the `main` method, they can continue their execution even after the main thread has ended. These threads can run independently and perform their tasks concurrently with the main thread.

However, if the main thread needs to wait for the completion of other threads before terminating, it can utilize various mechanisms to achieve synchronization and coordination. For example, it can use the `join()` method on the other threads, which blocks the main thread until the specified thread completes its execution. By using this approach, the main thread can ensure that all other threads have finished their tasks before it terminates.

3 Runnable Interface

The `Runnable` interface in Java provides a way to create a thread by implementing the `run()` method. This interface is commonly used to create concurrent programs in Java. Here's an example of how to implement the `Runnable` interface:

```java
class MyRunnable implements Runnable {
  public void run() {
    // Code to be executed concurrently
  }
}
```

To use the `MyRunnable` class, you can create an instance of it and pass it to a `Thread` object. The `Thread` class will then execute the `run()` method in a separate thread:

```java
MyRunnable myRunnable = new MyRunnable();
Thread thread = new Thread(myRunnable);
thread.start();
```

By implementing the `Runnable` interface, you can separate the code that needs to run concurrently from the code that controls the thread execution. This approach provides better separation of concerns and improves the overall structure of multithreaded programs in Java.

Example:

```java
// The task class for printing numbers from 1 to n for a given n
class PrintNum implements Runnable {
  private int lastNum;

  /** Construct a task for printing 1, 2, ... i */
  public PrintNum(int n) {
    lastNum = n;
  }

  @Override /** Tell the thread how to run */
  public void run() {
    for (int i = 1; i <= lastNum; i++) {
      System.out.print(" " + i);
    }
  }
}

// The task for printing a specified character in specified times
class PrintChar implements Runnable {
  private char charToPrint;
  private int times;

  /** Construct a task with specified character and number of times to print the character */
  public PrintChar(char c, int t) {
    charToPrint = c;
    times = t;
  }

  @Override /** Override the run() method to tell the system what task to perform */
  public void run() {
    for (int i = 0; i < times; i++) {
      System.out.print(charToPrint);
    }
  }
}

public class TaskThreadDemo {
  public static void main(String[] args) {
    // Create tasks
    Runnable printA = new PrintChar('a', 100);
    Runnable printB = new PrintChar('b', 100);
    Runnable print100 = new PrintNum(100);

    // Create threads
    Thread thread1 = new Thread(printA);
    Thread thread2 = new Thread(printB);
    Thread thread3 = new Thread(print100);

    // Start threads
    thread1.start();
    thread2.start();
    thread3.start();
  }
}
```

Output: ababababababababa 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72b 73 74 75 76 77 78 79 80bba 81a 82 83 84 85 86 87b 88ab 89bba 90 91 92 93 94 95aba 96 97 98 99 100babaababababababababababababababababababababababababababababab

4 Thread Class

The ‘Thread‘ class in Java provides the functionality to create and control threads. Here are some important methods of the ‘Thread‘ class:

1. `Thread(Runnable)`: The constructor of the ‘Thread‘ class that takes a ‘Runnable‘ object as a parameter. It creates a new thread and associates it with the provided ‘Runnable‘ object. The thread will execute the ‘run()‘ method of the ‘Runnable‘ object when started.

   Example:

```java
Runnable myRunnable = new MyRunnable();
Thread thread = new Thread(myRunnable);
thread.start();
```

2. `start()`: This method starts the execution of the thread. It calls the ‘run()‘ method of the associated ‘Runnable‘ object in a separate thread. The code inside the ‘run()‘ method will be executed concurrently.

   Example:

```java
Thread thread = new Thread(myRunnable);
thread.start();
```

3. `join()`: This method waits for the thread to complete its execution. It blocks the calling thread until the target thread terminates.

   Example:

```java
Thread thread = new Thread(myRunnable);
thread.start();

// Wait for the thread to complete
try {
  thread.join();
} catch (InterruptedException e) {
  // Handle interrupted exception
}
```

4. `isAlive()`: This method checks whether the thread is still active or not. It returns ‘true‘ if the thread is running or has been started but not yet completed, and ‘false‘ otherwise.

   Example:

```java
Thread thread = new Thread(myRunnable);
thread.start();

// Check if the thread is still active
if (thread.isAlive()) {
  System.out.println("Thread is still running.");
} else {
  System.out.println("Thread has completed.");
}
```

5. `setPriority(int)`: This method sets the priority of the thread. The valid priority values range from 1 (lowest) to 10 (highest). The default priority is 5. The thread scheduler uses the priority value to determine the order in which threads should be executed, but it's not guaranteed to have precise behavior across different platforms.

   Example:

```java
Thread thread = new Thread(myRunnable);
thread.setPriority(Thread.MAX_PRIORITY); // Set maximum priority
thread.start();
```

6. `sleep(long)`: This method causes the currently executing thread to sleep for the specified number of milliseconds. It temporarily suspends the execution of the thread without releasing any locks or resources.

   Example:

```java
// Sleep for 1 second (1000 milliseconds)
try {
  Thread.sleep(1000);
} catch (InterruptedException e) {
  // Handle interrupted exception
}
```

7. `yield()`: This method causes the currently executing thread to temporarily pause and allow other threads of the same priority to execute. It's a hint to the scheduler that the current thread is willing to yield its current use of the CPU.

   Example:

```java
// Allow other threads of the same priority to execute
Thread.yield();
```

8. `interrupt()`: This method interrupts the execution of the thread by throwing an 'InterruptedException'. It can be used to gracefully stop the thread execution or to communicate a cancellation request to the thread.

   Example:

```java
Thread thread = new Thread(myRunnable);
thread.start();

// Interrupt the thread
thread.interrupt();
```

9. `isInterrupted()`: This method checks whether the thread has been interrupted. It returns `true` if the thread has been interrupted, and `false` otherwise.

   Example:

```java
  Thread thread = new Thread(myRunnable);
  thread.start();

  // Check if the thread has been interrupted
  if (thread.isInterrupted()) {
    System.out.println("Thread has been interrupted.");
  } else {
    System.out.println("Thread has not been interrupted.");
  }
```

10. `stop()`: This method is deprecated and should not be used. It forcefully terminates the execution of the thread, causing it to immediately stop. Using this method can lead to unpredictable behavior and potential resource leaks. It is recommended to use other mechanisms, such as cooperative thread termination using shared variables or interruption.

    Example:

```java
Thread thread = new Thread(myRunnable);
thread.start();

// Stop the thread forcefully (deprecated, not recommended)
thread.stop();
```

11. `suspend()`: This method is deprecated and should not be used. It temporarily suspends the execution of the thread. However, using this method can lead to thread deadlock or other synchronization issues. It is recommended to use other thread synchronization mechanisms, such as locks or condition variables, for pausing and resuming thread execution.

    Example:

```java
Thread thread = new Thread(myRunnable);
thread.start();

// Suspend the thread (deprecated, not recommended)
thread.suspend();
```

12. `resume()`: This method is deprecated and should not be used. It resumes the execution of a thread that has been suspended using the `suspend()` method. However, using this method can lead to thread deadlock or other synchronization issues. It is recommended to use other thread synchronization mechanisms, such as locks or condition variables, for pausing and resuming thread execution.

    Example:

```java
Thread thread = new Thread(myRunnable);
thread.start();

// Suspend the thread (deprecated, not recommended)
thread.suspend();

// Resume the thread (deprecated, not recommended)
thread.resume();
```

5 Thread Priority

Each thread in Java is assigned a default priority of `Thread.NORM_PRIORITY`, which has a value of 5. The thread priority is used by the thread scheduler to determine the order in which threads should be executed. The higher the priority value, the more likely a thread will be scheduled for execution. However, thread priority is not guaranteed to have precise behavior across different platforms.

You can adjust the priority of a thread using the `setPriority(int priority)` method of the `Thread` class. The valid priority values range from 1 (lowest) to 10 (highest). Java provides some constants for commonly used priority values:

- `Thread.MIN_PRIORITY`: The minimum thread priority value, which is 0.
- `Thread.MAX_PRIORITY`: The maximum thread priority value, which is 10.
- `Thread.NORM_PRIORITY`: The default thread priority value, which is 5.

Example:

```java
Thread thread1 = new Thread(myRunnable);
thread1.setPriority(Thread.MIN_PRIORITY); // Set minimum priority

Thread thread2 = new Thread(myRunnable);
thread2.setPriority(Thread.MAX_PRIORITY); // Set maximum priority

Thread thread3 = new Thread(myRunnable);
thread3.setPriority(Thread.NORM_PRIORITY); // Set default priority
```

Remember that the actual behavior of thread scheduling based on priority may vary depending on the underlying operating system and JVM implementation.

6 ExecutorService

The ‘ExecutorService‘ is a higher-level concurrency utility in Java that provides a way to manage and control the execution of tasks in a multithreaded environment. It provides a thread pool that manages the execution of tasks asynchronously, allowing for efficient utilization of threads.

To create an ‘ExecutorService‘ instance, you can use the ‘Executors‘ class, which provides various factory methods for creating different types of thread pools. Two commonly used methods are ‘newFixedThreadPool()‘ and ‘newCachedThreadPool()‘.

6.1 newFixedThreadPool()

The ‘newFixedThreadPool(int nThreads)‘ method creates an ‘ExecutorService‘ with a fixed-size thread pool. It maintains a pool of a specified number of threads that remain active even when they are idle. If all threads are busy, additional tasks are queued until a thread becomes available. The number of threads in the pool remains constant unless explicitly changed.

Example:

```java
ExecutorService executor = Executors.newFixedThreadPool(5);

// Submit tasks for execution
executor.execute(task1);
executor.execute(task2);
executor.execute(task3);

// Shutdown the executor when no more tasks need to be submitted
executor.shutdown();
```

In this example, we create an ‘ExecutorService‘ with a fixed-size thread pool of 5 threads. We submit three tasks for execution, and once all tasks are completed, we shut down the executor using the ‘shutdown()‘ method.

6.2 newCachedThreadPool()

The ‘newCachedThreadPool()‘ method creates an ‘ExecutorService‘ with a dynamically-sized thread pool. It does not have a fixed number of threads. Instead, it creates new threads as needed and reuses idle threads if available. If a thread remains idle for a certain period, it may be terminated to free up resources. This makes it suitable for applications with a varying workload.

Example:

```java
ExecutorService executor = Executors.newCachedThreadPool();

// Submit tasks for execution
executor.execute(task1);
executor.execute(task2);
executor.execute(task3);

// Shutdown the executor when no more tasks need to be submitted
executor.shutdown();
```

In this example, we create an ‘ExecutorService‘ with a dynamically-sized thread pool. We submit three tasks for execution, and once all tasks are completed, we shut down the executor using the ‘shutdown()‘ method.

The choice between ‘newFixedThreadPool()‘ and ‘newCachedThreadPool()‘ depends on the specific requirements of your application. If you have a fixed number of tasks or want to limit resource usage, ‘newFixedThreadPool()‘ is suitable. If your workload is dynamic and can benefit from automatically adjusting the number of threads, ‘newCachedThreadPool()‘ is a good choice.

7 Race Condition and `synchronized` Keyword

In concurrent programming, a race condition occurs when multiple threads access shared data concurrently, and the final result of the computation depends on the timing and interleaving of their execution. Race conditions can lead to unpredictable and incorrect results in multithreaded programs.

To prevent race conditions and ensure thread safety, Java provides the ‘synchronized‘ keyword, which allows for the synchronization of methods or blocks of code. When a method or block is marked as ‘synchronized‘, only one thread can execute it at a time, preventing concurrent access to shared resources.

7.1 Synchronized Methods

A synchronized method is a method that is declared with the ‘synchronized‘ keyword. When a thread invokes a synchronized method, it acquires the intrinsic lock (also known as the monitor lock) associated with the object on which the method is called. Other threads that try to invoke synchronized methods on the same object have to wait until the lock is released.

Example:

```java
public class Counter {
  private int count;

  public synchronized void increment() {
    count++;
  }
}
```

In this example, the ‘increment()‘ method is marked as synchronized. Only one thread can execute this method at a time, ensuring that the ‘count‘ variable is updated atomically.

7.2 Synchronized Blocks

In addition to synchronized methods, Java also allows the synchronization of specific blocks of code using the `synchronized` keyword. With synchronized blocks, you can control the granularity of synchronization by specifying the object on which the block should be synchronized.

ReentrantLock is a synchronization mechanism in Java that provides exclusive access to a shared resource by multiple threads.Consider the following example:

```java
public class ReentrantLockExample {
    private ReentrantLock lock = new ReentrantLock();

    public void sharedResource() {
        lock.lock(); // Acquire the lock
        try {
            // Critical section: Access the shared resource
            // ...
        } finally {
            lock.unlock(); // Release the lock
        }
    }
}
```

Another Big Example:

```java
import java.util.concurrent.*;
import java.util.concurrent.locks.*;

public class AccountWithSyncUsingLock {
  private static Account account = new Account();

  public static void main(String[] args) {
    ExecutorService executor = Executors.newCachedThreadPool();

    // Create and launch 100 threads
    for (int i = 0; i < 100; i++) {
      executor.execute(new AddAPennyTask());
    }
    executor.shutdown();

    // Wait until all tasks are finished
    while (!executor.isTerminated()) {
    }

    System.out.println("What is the balance? " + account.getBalance());
  }

  // A thread for adding a penny to the account
  public static class AddAPennyTask implements Runnable {
    public void run() {
      account.deposit(1);
    }
  }

  // An inner class for the account
  public static class Account {
    private static Lock lock = new ReentrantLock();
    private int balance = 0;

    public int getBalance() {
      return balance;
    }

    public void deposit(int amount) {
      lock.lock(); // Acquire the lock
      try {
        int newBalance = balance + amount;
        // This delay is deliberately added to magnify the
        // data-corruption problem and make it easy to see.
        Thread.sleep(5);
        balance = newBalance;
      } catch (InterruptedException ex) {
      } finally {
        lock.unlock(); // Release the lock
      }
    }
  }
}
```

In this example, we have a scenario where multiple threads are concurrently depositing one penny into an account. The ‘AccountWithSyncUsingLock‘ class encapsulates the functionality.

The ‘Account‘ class contains a ‘deposit‘ method that is responsible for adding the specified amount to the account balance. The method is synchronized using a lock provided by the ‘Lock‘ interface, specifically the ‘ReentrantLock‘ implementation. By acquiring the lock before modifying the balance, we ensure that only one thread can execute the critical section of code at a time. This prevents race conditions and data corruption.

The ‘AddAPennyTask‘ class represents a thread that executes the ‘deposit‘ method to add one penny to the account.

The ‘ExecutorService‘ is used to create and manage a thread pool, where 100 instances of ‘AddAPennyTask‘ are submitted for execution. The threads concurrently execute the ‘deposit‘ method, but due to the synchronized block and the lock, only one thread can modify the account balance at any given time.

Once all tasks are completed, the main thread retrieves the final account balance using the ‘getBalance‘ method and outputs the result.

By utilizing synchronized blocks and locks, we ensure thread safety and prevent race conditions when multiple threads access shared data concurrently.

8 Thread Synchronization with wait(), notify(), and notifyAll()

In Java, threads can communicate and coordinate their actions using the methods wait(), notify(), and notifyAll() provided by the Object class. These methods enable thread synchronization and inter-thread communication.

8.1 Understanding wait(), notify(), and notifyAll()

The wait() method causes the current thread to release the lock it holds and enter a suspended state until another thread notifies it to resume. This method is typically used in conjunction with the notify() or notifyAll() methods.

The notify() method wakes up a single thread that is waiting on the same object. If multiple threads are waiting, the thread awakened is arbitrary and not specified. On the other hand, the notifyAll() method wakes up all threads that are waiting on the same object, allowing them to compete for the lock and resume execution.

To use these methods effectively, the calling thread must have acquired the lock on the object using the synchronized keyword. Otherwise, an IllegalMonitorStateException will be thrown.

8.2 Example Usage

Consider the following example:

```java
public class WaitNotifyExample {
private static final Object lock = new Object();
private static boolean isReady = false;

public static void main(String[] args) {
    Thread waitingThread = new Thread(new WaitingTask());
    Thread notifyingThread = new Thread(new NotifyingTask());

    waitingThread.start();
    notifyingThread.start();
}

public static class WaitingTask implements Runnable {
    @Override
    public void run() {
        synchronized (lock) {
            while (!isReady) {
                try {
                    System.out.println("Waiting for notification...");
                    lock.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            System.out.println("Received notification. Resuming execution.");
        }
    }
}

public static class NotifyingTask implements Runnable {
    @Override
    public void run() {
        synchronized (lock) {
            try {
                Thread.sleep(2000); // Simulate some work
                System.out.println("Performing work and notifying waiting thread.");
                isReady = true;
                lock.notify();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
}
```

In this example, we have two threads: a waiting thread and a notifying thread. The waiting thread waits for a notification from the notifying thread before resuming its execution.

The waiting thread enters a synchronized block and checks the condition variable isReady. If the condition is false, it enters a waiting state using the wait() method. This releases the lock held by the thread and allows other threads to execute. The waiting thread remains suspended until it is notified by the notifying thread.

The notifying thread also enters a synchronized block, performs some work (simulated by a Thread.sleep() call), updates the condition variable isReady, and then calls notify() to wake up the waiting thread.

When the waiting thread is notified, it reacquires the lock and resumes execution from where it left off. In this case, it prints a message indicating that it has received the notification and resumes further processing.

8.3 Choosing the Right Method

When deciding between notify() and notifyAll(), consider the following:

If only one thread needs to be awakened, and it is known which thread should be awakened, use notify(). If multiple threads are waiting and all need to be awakened, use notifyAll(). Using notifyAll() when only one thread is waiting can cause unnecessary wake-ups and potential performance issues.

8.4 Summary

The wait(), notify(), and notifyAll() methods provide a powerful mechanism for thread synchronization and coordination. By utilizing these methods along with locks and synchronized blocks, you can design more efficient and robust multi-threaded applications.

Remember to always call these methods within a synchronized block, using the same lock object, to ensure proper thread synchronization and avoid IllegalMonitorStateException errors.

9 Condition Interface

The `Condition` interface is part of the `java.util.concurrent.locks` package and provides a way to manage thread synchronization and coordination. It is used in conjunction with locks, typically the `ReentrantLock` class, to enable threads to wait for certain conditions to be satisfied before proceeding.

The `Condition` interface defines three fundamental methods:

- `await()`: Causes the current thread to wait until the condition is signaled by another thread or interrupted. The thread releases the associated lock and suspends execution until it is signaled or interrupted. When the thread is awakened, it re-acquires the lock before returning from the `await()` call.
- `signal()`: Wakes up one waiting thread that is waiting on this condition. If multiple threads are waiting, it is not specified which thread will be signaled. The awakened thread must re-acquire the lock before it can proceed.
- `signalAll()`: Wakes up all waiting threads that are waiting on this condition. Each thread must re-acquire the lock before it can proceed.

Here's an example to illustrate the usage of `Condition` in a producer-consumer scenario:

```java
import java.util.LinkedList;
import java.util.Queue;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;

class ProducerConsumer {
    private Queue<Integer> buffer;
    private int maxSize;
    private ReentrantLock lock;
    private Condition bufferNotFull;
    private Condition bufferNotEmpty;

    public ProducerConsumer(int maxSize) {
        this.maxSize = maxSize;
        buffer = new LinkedList<>();
        lock = new ReentrantLock();
        bufferNotFull = lock.newCondition();
        bufferNotEmpty = lock.newCondition();
    }

    public void produce() throws InterruptedException {
        lock.lock();
        try {
            while (buffer.size() == maxSize) {
                bufferNotFull.await();  // Wait until the buffer is not full
            }
            int item = generateItem();
            buffer.add(item);
            System.out.println("Produced: " + item);
            bufferNotEmpty.signal();  // Signal that the buffer is not empty
        } finally {
            lock.unlock();
        }
    }

    public void consume() throws InterruptedException {
        lock.lock();
        try {
            while (buffer.isEmpty()) {
                bufferNotEmpty.await();  // Wait until the buffer is not empty
            }
            int item = buffer.remove();
            System.out.println("Consumed: " + item);
            bufferNotFull.signal();  // Signal that the buffer is not full
        } finally {
            lock.unlock();
        }
    }

    private int generateItem() {
        // Generate and return a random item
        return (int) (Math.random() * 100);
    }
}

}
```

In this example, the `ProducerConsumer` class represents a shared buffer between a producer and a consumer. The `bufferNotFull` condition is used by the producer to wait if the buffer is full, and the `bufferNotEmpty` condition is used by the consumer to wait if the buffer is empty. The producer and consumer methods, as well as other methods, would be implemented accordingly to utilize these conditions.

By using the `await()`, `signal()`, and `signalAll()` methods of the `Condition` interface along with appropriate locking mechanisms, you can effectively coordinate threads and manage their execution based on certain conditions.