**What Is A Thread?**

A **thread** is a lightweight unit of execution within a process. It has its stack of memory and can run independently of other threads in the same process. Threads share the same process resources, such as the heap and the code section. The Java Virtual Machine allows an application to have multiple threads of execution running concurrently.

Every thread has a **priority**. Threads with higher priority are executed in preference to threads with lower priority. Each thread may or may not also be marked as a **daemon**. When code running in some thread creates a new Thread object, the new thread has its priority initially set equal to the priority of the creating thread and is a **daemon** thread if and only if the creating thread is a daemon. When a Java Virtual Machine starts up, there is usually a single non-daemon thread (which typically calls the method named **main** of some designated class).

**Thread States And LifeCycle**

A thread can be in one of the following states:

**NEW:** A thread that has not yet started is in this state.

**RUNNABLE:** A thread executing in the Java virtual machine is in this state.

**BLOCKED:**A thread that is blocked waiting for a monitor lock is in this state.

**WAITING:** A thread that is waiting indefinitely for another thread to perform a particular action is in this state.

**TIMED\_WAITING:** A thread that is waiting for another thread to act for up to a specified waiting time is in this state.

**TERMINATED:** A thread that has exited is in this state.

**Below is the thread lifecycle:**

A diagram of a running process

Description automatically generated

Here’s an explanation of the thread lifecycle in Java:

1. **New (or Created):** A thread is created by instantiating the **Thread** class. At this point, it is in the new state.

Thread myThread = new Thread();

**2. Runnable (or Ready):** After the thread is created, it moves to the runnable state when the start() method is called. The start() method internally calls the run() method, and the thread becomes ready for execution.

myThread.start(); // Moves the thread to the runnable state

**3. Running:** Once the scheduler selects the thread for execution, it enters the running state. The run() method contains the code that will be executed when the thread is running.

public void run() {  
 // Code to be executed when the thread is running  
}

**4. Blocked (or Waiting):**A running thread may enter the blocked state if it encounters an operation that makes it wait, such as calling sleep(), wait(), or performing I/O operations.

// Example: Thread sleeps for 1 second  
try {  
 Thread.sleep(1000); // Thread enters blocked state for 1 second  
} catch (InterruptedException e) {  
 e.printStackTrace();  
}

**5. Terminated (or Dead):**The thread enters the terminated state when the **run() method completes** its execution or when an uncaught exception occurs.

// Example: Completing the run method  
public void run() {  
 // Code to be executed when the thread is running  
 // ...  
 // The thread terminates when this method completes  
}

It’s important to note that the**start()** method should be used to initiate the execution of a thread. The actual code to be executed should be placed in the **run()** method. Calling **run() directly won’t create a new thread**; it will execute the run() method in the context of the current thread. Using start() is essential for multithreading, as it signals the system to create a new thread and invoke the run() method in that new thread.

**Creating Threads**

There are two main ways to create threads in Java:

* **Extending the Thread class:** We can **extend** the **Thread** class and override the **run()** method to define the thread’s behavior. To extend the Thread class, we must override the run() method. The run() method contains the code that the thread will execute. Once we have created a thread, we can start it by calling the **start()** method. The **start()** method causes the thread to begin executing the run() method.

class MyThread extends Thread {  
 public void run() {  
 // Code to be executed in the new thread  
 }  
}  
MyThread myThread = new MyThread();  
myThread.start(); // Start the new thread

* **Implementing the Runnable interface:** We can implement the Runnable interface and pass an instance of the class to the Thread constructor.

class MyRunnable implements Runnable {  
 public void run() {  
 // Code to be executed in the new thread  
 }  
}  
Thread myThread = new Thread(new MyRunnable());  
myThread.start(); // Start the new thread

In this example, **MyRunnable** implements the **Runnable** interface, and we override the run method. Then, we create Thread instances, pass an instance of MyRunnable to their constructors, and start the threads.

Using the **Runnable** interface is often preferred because it allows for better flexibility. ***We can use the same Runnable object to create multiple threads, and it separates the task from the thread, promoting a cleaner design.***

**Thread Synchronization**

Thread synchronization is the process of coordinating the execution of multiple threads to ensure that they access shared resources or perform specific tasks in a mutually exclusive or coordinated manner. The goal is to avoid race conditions, data inconsistencies, and other concurrency issues that may arise when multiple threads execute concurrently. In Java, synchronization can be achieved using various mechanisms:

1. **Synchronized Methods:** Using the synchronized keyword with a method ensures that only one thread can execute the synchronized method on the same object at a time. This provides intrinsic lock-based synchronization.

public synchronized void synchronizedMethod() {  
 // Code to be executed in a mutually exclusive manner  
}

**2. Synchronized Blocks:**Synchronized blocks allow more granular control over the region of code that needs to be synchronized. It’s often used to avoid unnecessary contention for the lock.

public void someMethod() {  
 // Non-critical section code  
  
 synchronized (lockObject) {  
 // Critical section code  
 }  
  
 // Non-critical section code  
}

**3. Volatile Keyword:** The volatile keyword ensures that a variable’s value is always read and written directly from and to the main memory. While it doesn’t provide full synchronization, it is useful for certain scenarios, such as flagging a variable to be accessed by multiple threads.

private volatile boolean flag = false;  
  
public void setFlag() {  
 flag = true;  
}  
  
public boolean checkFlag() {  
 return flag;  
}

**4. Wait and Notify:**The wait() and notify() (or notifyAll()) methods, along with synchronized blocks, can be used for more advanced thread communication and coordination.

synchronized (sharedObject) {  
 while (conditionNotMet) {  
 sharedObject.wait();  
 }  
 // Perform actions when the condition is met  
}

Proper synchronization is essential for writing correct and thread-safe concurrent programs. Careful consideration and understanding of the synchronization mechanisms, as well as the potential for deadlocks and contention, are crucial for effective multithreading in Java.

**Understanding the Producer-Consumer Problem:**

The Producer-Consumer problem is a synchronization problem where two threads, the producer and the consumer, share a common, fixed-size buffer. The producer’s role is to produce data and add it to the buffer, while the consumer’s role is to consume the data from the buffer. The challenge lies in ensuring that the producer doesn’t produce data when the buffer is full and that the consumer doesn’t consume when the buffer is empty.

Now, let’s delve into the Java implementation of this scenario using the best and most efficient approach.

**Implementation using BlockingQueue:**

In Java, the BlockingQueue interface provides a convenient solution for implementing the Producer-Consumer problem efficiently. We'll use the ArrayBlockingQueue class, a bounded blocking queue backed by an array.

class Buffer {

private Queue<Integer> queue;

private int capacity;

public Buffer(int capacity) {

this.queue = new LinkedList<>();

this.capacity = capacity;

}

public synchronized void produce(int item) throws InterruptedException {

*// Wait while the buffer is full*

while (queue.size() == capacity) {

wait();

}

queue.add(item);

System.out.println("Produced: " + item);

*// Notify waiting consumers that an item is available*

notify();

}

public synchronized int consume() throws InterruptedException {

*// Wait while the buffer is empty*

while (queue.isEmpty()) {

wait();

}

int item = queue.remove();

System.out.println("Consumed: " + item);

*// Notify waiting producers that space is available*

notify();

return item;

}

}

class Producer implements Runnable {

private Buffer buffer;

public Producer(Buffer buffer) {

this.buffer = buffer;

}

@Override

public void run() {

for (int i = 0; i < 10; i++) {

try {

buffer.produce(i);

*// Simulate varying production times*

Thread.sleep(100);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

class Consumer implements Runnable {

private Buffer buffer;

public Consumer(Buffer buffer) {

this.buffer = buffer;

}

@Override

public void run() {

for (int i = 0; i < 10; i++) {

try {

buffer.consume();

*// Simulate varying consumption times*

Thread.sleep(200);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

class ProducerConsumerExample {

public static void main(String[] args) {

Buffer buffer = new Buffer(5);

Thread producerThread = new Thread(new Producer(buffer));

Thread consumerThread = new Thread(new Consumer(buffer));

producerThread.start();

consumerThread.start();

}

}

**DeadLock and LiveLock**

**Deadlock:**

* Occurs when threads are waiting for each other to release resources, forming a circular dependency.
* All involved threads are blocked and unable to proceed.
* In our producer-consumer example, we avoid deadlock by using a single lock (the intrinsic lock of the Buffer object) and carefully managing when threads wait and notify.

**Livelock:**

* Occurs when threads are actively running but unable to make progress.
* Unlike deadlock, threads are not blocked, but they're stuck in a loop of responses to each other.
* In a producer-consumer scenario, a livelock could occur if the producer and consumer continuously yield to each other without actually producing or consuming.