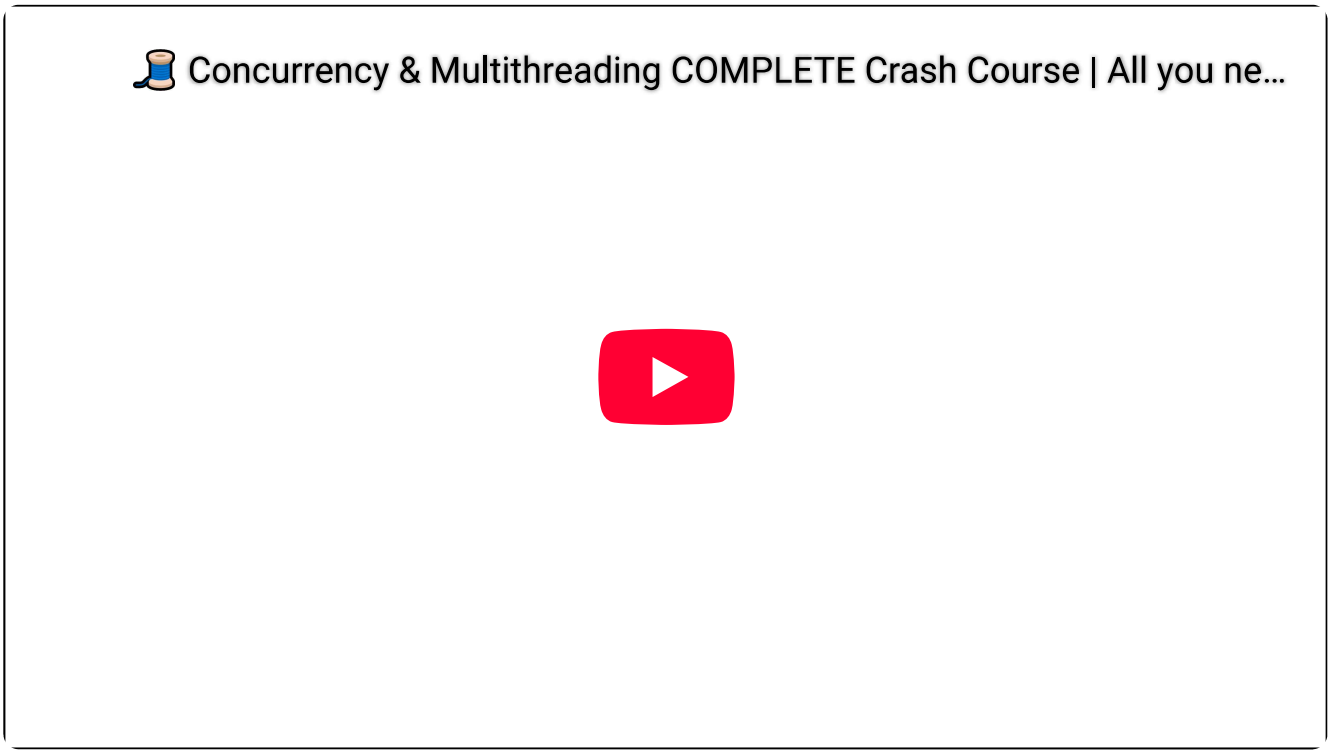
Search articles...

Sign In

Thread pool and Thread lifecycle



Topic Tags:

system design lld

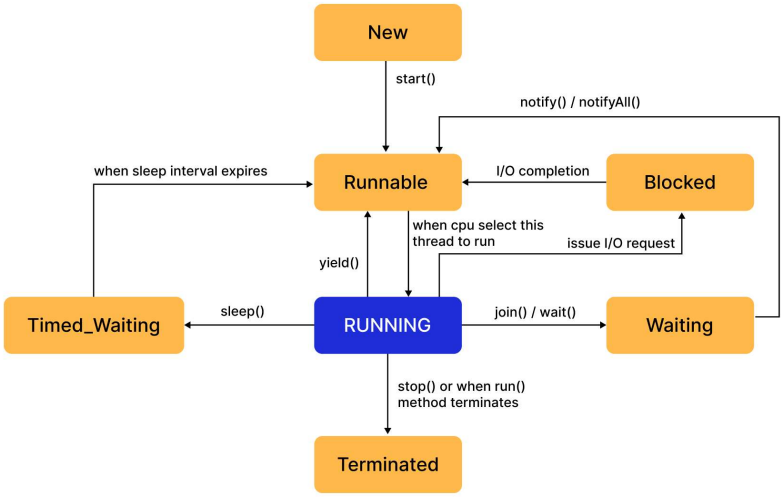
Github Codes Link: https://github.com/aryan-0077/CWA-LowLevelDesignCode

Java Thread Pools and Thread Lifecycle

Thread pools and understanding the thread lifecycle are crucial concepts for effective concurrent programming. They enable developers to create scalable applications that efficiently utilize system resources while maintaining control over thread creation and management.

Thread Lifecycle in Java

State Diagram :



Java

```java
Thread thread = new Thread(() -> System.out.println("Hello from thread"));
// Thread is in NEW state here
```

2. RUNNABLE 🏃

- Thread is ready for execution and waiting for CPU allocation
- Once start() is called, thread moves to this state
- Includes ready-to-run state

Java

```java
Thread thread = new Thread(() -> System.out.println("Hello from thread"));
thread.start(); // Thread moves to RUNNABLE state
```

3. RUNNING ⚙️

- The thread is currently executing its task on the CPU.
- The CPU scheduler has allocated processing time to this thread.

Java

```java
// When the CPU scheduler picks a RUNNABLE thread, it enters the RUNNING sta
// The code within the thread's run() method is being executed here.
public class RunningExample extends Thread {
    @Override
    public void run() {
        System.out.println("Thread is now RUNNING.");
        // ... thread's task execution ...
    }


    public static void main(String[] args) {
        RunningExample thread = new RunningExample();
        thread.start(); // Moves to RUNNABLE, then eventually RUNNING
    }
}
```

4. BLOCKED 🚧

- Thread is temporarily inactive while waiting to acquire a lock
- Typically occurs when trying to enter a synchronized block/method already locked by another thread

Java

```java
synchronized(lockObject) {
    // If another thread holds lockObject's monitor,
    // this thread will be BLOCKED until lock is available
}
```

5. WAITING ⏳

- Thread is waiting indefinitely for another thread to perform a specific action
- Entered via methods like Object.wait(), Thread.join(), or LockSupport.park()
- No timeout specified

Java

```
synchronized(lockObject) {
    try {
        lockObject.wait(); // Thread enters WAITING state
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
}
```

Simulation: How Another Thread Wakes the waiting Thread :

Scenario: Restaurant Order Processing 🍔

- **Thread A (Waiter)** takes the customer's order and **waits** for the chef to prepare the food.
- **Thread B (Chef)** prepares the food and **notifies the waiter** when it's ready.

Code Implementation :

- **Thread A (Waiter - Enters WAITING State)**

Java

```
class WaiterThread extends Thread {
    private final Object lock;

    public WaiterThread(Object lock) {
        this.lock = lock;
    }

    @Override
    public void run() {
        synchronized (lock) {
            try {
                System.out.println("Waiter: Waiting for the food to be rea
                lock.wait(); // Waiter enters WAITING state
                System.out.println("Waiter: Food is ready! Delivering to t
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
```

```
        }
    }
}
```

- **Thread B (Chef - Notifies the Waiter) :**

Java

```java
class ChefThread extends Thread {
    private final Object lock;

    public ChefThread(Object lock) {
        this.lock = lock;
    }

    @Override
    public void run() {
        try {
            Thread.sleep(2000); // Simulate food preparation time
            synchronized (lock) {
                System.out.println("Chef: Food is ready! Notifying the wai
                lock.notify(); // Wake up the waiting waiter thread
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}
```

- **Main Execution :**

Java

```java
public class RestaurantSimulation {
    public static void main(String[] args) {
        Object lock = new Object();
        Thread waiter = new WaiterThread(lock);
        Thread chef = new ChefThread(lock);
```

```
        waiter.start();
        chef.start();
    }
}
```

Output :

```
Waiter: Waiting for the food to be ready... ⏳
Chef: Food is ready! Notifying the waiter. 🔔
Waiter: Food is ready! Delivering to the customer. 🍽️
```

A thread enters the **WAITING** state when it is **indefinitely waiting** for another thread to perform a specific action before it can proceed.

🛠 Entered via methods like:

- Object.wait()
- Thread.join()
- LockSupport.park()

❌ **No timeout is specified**, meaning the thread will remain **stuck indefinitely** unless another thread **wakes it up using the notify() or notifyAll() method**.

6. TIMED_WAITING ⏱️

- Thread is waiting for a specified period of time
- Entered via methods like Thread.sleep(timeout), Object.wait(timeout), etc.
- Will automatically return to RUNNABLE after timeout expires or notification

Java

```java
try {
    Thread.sleep(1000); // Thread enters TIMED_WAITING state for 1 second
} catch (InterruptedException e) {
    Thread.currentThread().interrupt();
}
```

To Understand about how sleep() and wait() methods work, refer to the first article **Thread - Thread Class and Runnable Interface** of Concurrency Module.

7. TERMINATED 🏁

- Thread has completed its execution or was stopped
- The run() method has exited, either normally or due to an exception
- Thread object still exists but cannot be restarted

```
// After thread's run() method completes
// Thread is in TERMINATED state
```

Thread Pools in Java 🏊

Thread pools are a managed collection of reusable threads designed to execute tasks concurrently. They offer significant advantages in resource management, performance, and application stability. 🛠️

Example :

Java

```java
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

class WorkerThread implements Runnable {
    private final int taskId;

    public WorkerThread(int taskId) {
        this.taskId = taskId;
    }

    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + " is process
        try {
            Thread.sleep(2000); // Simulate task execution time
        } catch (InterruptedException e) {
            System.out.println("Task interrupted: " + e.getMessage());
        }
```

```
        System.out.println(Thread.currentThread().getName() + " finished t
    }
}


public class ThreadPoolExample {
    public static void main(String[] args) {
        // Create a fixed thread pool with 3 threads
        ExecutorService executorService = Executors.newFixedThreadPool(3);

        // Submit 5 tasks to the thread pool
        for (int i = 1; i <= 5; i++) {
            executorService.submit(new WorkerThread(i));
        }

        // Shutdown the executor service
        executorService.shutdown();
    }
}
```

Output :

```
pool-1-thread-1 is processing task: 1
pool-1-thread-2 is processing task: 2
pool-1-thread-3 is processing task: 3
pool-1-thread-1 finished task: 1
pool-1-thread-1 is processing task: 4
pool-1-thread-2 finished task: 2
pool-1-thread-2 is processing task: 5
pool-1-thread-3 finished task: 3
pool-1-thread-1 finished task: 4
pool-1-thread-2 finished task: 5
```

Explanation :

1. **Thread Pool Creation** 🚀

- Executors.newFixedThreadPool(3) creates a pool with 3 reusable threads.

2.Task Submission 📌

- Five tasks are submitted. Since only 3 threads exist, the first 3 tasks start immediately.
- As tasks complete, the available threads pick up the remaining tasks.

2.Efficient Thread Usage 🛠️

- Threads are **reused**, avoiding the overhead of creating new threads for each task.
- The execution order may vary based on CPU scheduling.

Benefits of Thread Pools 🌟

- **Resource Management**: Limit the number of threads to prevent system overload. 🛑
- **Performance Improvement**: Reuse existing threads instead of creating new ones. ⚡
- **Predictability**: Control thread creation and scheduling for better application behavior. 📊
- **Task Management**: Queuing, scheduling, and monitoring tasks becomes streamlined. 📋

Combining Thread Lifecycle and Pools 🔄👩‍💻

Understanding how thread lifecycle relates to thread pools helps create more efficient applications:

1. **Pool Creation**: When a thread pool is created, it may pre-create some threads (core threads) in the NEW state and immediately start them to RUNNABLE.

2.Task Execution: When a task is submitted:

- An idle thread in the pool executes the task
- The thread's state changes according to task operations (RUNNABLE, RUNNING, BLOCKED, WAITING, etc.)
- After task completion, the thread returns to the pool (RUNNABLE state waiting for next task)

3.Pool Shutdown: During shutdown, threads complete their current tasks and are eventually terminated.

Example :

Java

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

class Task implements Runnable {
```

```
    private final int taskId;

    public Task(int taskId) {
        this.taskId = taskId;
    }

    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + " - STARTING
        try {
            // Simulating different thread states
            Thread.sleep(2000); // Simulates RUNNABLE -> TIMED_WAITING (Sl

           synchronized (this) {
                System.out.println(Thread.currentThread().getName() + " -
                // The thread is now RUNNING and enters a synchronized blo
                this.wait(1000); // Simulates WAITING state for 1 second
                // The thread leaves the RUNNING state and enters the WAIT
            }
                        // After wait() (either by timeout or notify), th
            // When the scheduler picks it, it re-enters the RUNNING state
            System.out.println(Thread.currentThread().getName() + " - Task
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            // If interrupted while RUNNING, it might transition to TERMIN
            // If interrupted while in TIMED_WAITING or WAITING, it will t
        }
        // After the try-catch block, if the pool is still active, the thr
        // waiting for a new task. If the pool is shutting down, it will e
    }
}

public class ThreadPoolLifecycleDemo {
    public static void main(String[] args) {
        // Step 1: Create a Thread Pool with 3 core threads
        ExecutorService executor = Executors.newFixedThreadPool(3);
        System.out.println("Thread Pool Created 🏫");
        // Step 2: Submit 5 tasks to the pool
        for (int i = 1; i <= 5; i++) {
            executor.execute(new Task(i)); // Threads pick tasks and move
        }
```

```
        // Step 3: Initiate shutdown after all tasks are submitted
        executor.shutdown();
        System.out.println("Thread Pool Shutdown Initiated 🚦");

        try {
            // Wait for all threads to terminate
            if (!executor.awaitTermination(10, TimeUnit.SECONDS)) {
                executor.shutdownNow();
                System.out.println("Forcing Shutdown! 🚧");
               // If shutdownNow is called, threads currently in RUNNING s
            }
        } catch (InterruptedException e) {
            executor.shutdownNow();
        }
        System.out.println("All Threads Terminated ✅");

// Once shutdown is complete, all threads that were processing tasks (RUNN
// will have completed their work or been interrupted and will eventually
    }
}
```

Output :

```
Thread Pool Created 🏊
Thread-0 - STARTING Task 1
Thread-1 - STARTING Task 2
Thread-2 - STARTING Task 3
Thread-0 - WAITING on Task 1
Thread-1 - WAITING on Task 2
Thread-2 - WAITING on Task 3
Thread-0 - Task 1 COMPLETED
Thread-0 - STARTING Task 4
Thread-1 - Task 2 COMPLETED
Thread-1 - STARTING Task 5
Thread-0 - WAITING on Task 4
Thread-1 - WAITING on Task 5
Thread-2 - Task 3 COMPLETED
Thread Pool Shutdown Initiated 🚦
Thread-0 - Task 4 COMPLETED
Thread-1 - Task 5 COMPLETED
All Threads Terminated ✅
```

Thread Lifecycle Management

1. **Handle Interrupted Exception properly** to allow clean thread termination. ⚠️

Example : A worker thread checking for updates in a loop should **exit gracefully** when interrupted instead of ignoring the exception.

Java

```java
class WorkerThread implements Runnable {
    @Override
    public void run() {
        try {
            while (!Thread.currentThread().isInterrupted()) {
                System.out.println("Checking for updates...");
                Thread.sleep(2000); // Simulating work
            }
        } catch (InterruptedException e) {
            System.out.println("Thread interrupted, shutting down graceful
        }
    }
}

public class ThreadInterruptionExample {
    public static void main(String[] args) throws InterruptedException {
        Thread thread = new Thread(new WorkerThread());
        thread.start();
        Thread.sleep(5000); // Let it run for some time
        thread.interrupt(); // Interrupt the thread
    }
}
```

2.Avoid thread leaks by ensuring threads don't get stuck in WAITING or BLOCKED states 🧹

Example : A thread waiting **indefinitely** for a signal can cause a leak. Use **timeouts** to prevent this while acquiring locks or waiting on conditions.

Java

```java
class SafeLock {
    private final Object lock = new Object();
```

```
    void waitForSignal() {
        synchronized (lock) {
            try {
                System.out.println(Thread.currentThread().getName() + " is
                lock.wait(3000); // Wait with a timeout to prevent leak
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }
}

public class ThreadLeakExample {
    public static void main(String[] args) {
        SafeLock safeLock = new SafeLock();
        new Thread(safeLock::waitForSignal, "WorkerThread").start();
    }
}
```

Thread Pool Usage 🛠️

1. **Choose the right pool type** for your specific workload characteristics. 🎯

Example Scenario:

- **CPU-intensive tasks** → Executors.newFixedThreadPool(n)
- **CPU-bound tasks** (like image processing, video encoding, or complex calculations) spend most of their time **using the CPU**, rather than waiting for external resources.
- **Too many threads** can lead to **excessive context switching**, slowing down performance.
- A **fixed number of threads** (equal to the number of CPU cores) ensures that CPU resources are **fully utilized without excessive overhead**.

'

Java

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class CPUIntensiveExample {
```

```
    private static final int NUM_CORES = Runtime.getRuntime().availablePro
    public static void main(String[] args) {
        ExecutorService fixedPool = Executors.newFixedThreadPool(NUM_CORES
        for (int i = 0; i < 10; i++) {
            fixedPool.execute(() -> {
                int result = performComputation();
                System.out.println(Thread.currentThread().getName() + " co
            });
        }
        fixedPool.shutdown();
    }

    private static int performComputation() {
        int sum = 0;
        for (int i = 0; i < 1_000_000; i++) {
            sum += Math.sqrt(i); // Simulating heavy computation
        }
        return sum;
    }
}
```

Short-lived tasks → Executors.newCachedThreadPool()

- **I/O-bound tasks** (like web scraping, database queries, file I/O) spend most of their time **waiting**.
- **Threads are created dynamically** as needed, avoiding delays due to waiting.
- If a thread is **inactive**, it is **reused** instead of creating a new one, **reducing overhead**.

Java

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class IOBoundExample {
  public static void main(String[] args) {
    ExecutorService cachedPool =
        Executors.newCachedThreadPool(); // Best for I/O-bound short tasks
    for (int i = 0; i < 10; i++) {
```

```
        cachedPool.execute(() -> {
          simulateWebRequest();
          System.out.println(
              Thread.currentThread().getName() + " completed I/O task.");
        });
      }
      cachedPool.shutdown();
    }

    private static void simulateWebRequest() {
      try {
        System.out.println(
            Thread.currentThread().getName() + " is waiting for response...");
        TimeUnit.MILLISECONDS.sleep(500); // Simulating network delay
      } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
      }
    }
}
```

Why newCachedThreadPool() is often better for I/O-bound tasks than newFixedThreadPool() ? :

- **Efficient Resource Utilization**: I/O-bound tasks spend a significant amount of time waiting for I/O operations to complete. In a newFixedThreadPool(), the fixed number of threads might be blocked waiting, even if there's other work that could be done. newCachedThreadPool() spins up new threads when needed, allowing more tasks to proceed concurrently while others are waiting on I/O. This maximizes the utilization of available CPU resources during those waiting periods.

- **Responsiveness:** Because new threads can be created quickly to handle incoming tasks, a cached thread pool can be more responsive to bursts of I/O-bound operations. A fixed thread pool might have a backlog of tasks waiting for a thread to become free.

- **Automatic Thread Management:** The cached thread pool automatically manages thread creation and termination. Idle threads are reclaimed after a period of inactivity (typically 60 seconds), reducing resource consumption when the system is less busy. With a fixed thread pool, the threads exist for the lifetime of the executor, even if they are often idle.

Why newFixedThreadPool() might be less ideal for I/O-bound tasks in many scenarios ? :

- **Potential Underutilization:** If the number of threads in the fixed pool is too small, many tasks might be waiting for a thread to become available while other threads are blocked on I/O. This can lead to underutilization of the system's potential concurrency.

- **Overhead with Too Many Threads:** If you try to compensate for I/O wait times by making the fixed thread pool very large, you might introduce excessive context switching overhead, which can negatively impact performance.

In Summary, newCachedThreadPool() adapts better to the fluctuating nature of I/O-bound tasks, efficiently utilizing resources by creating threads as needed and reclaiming them when idle. This dynamic behavior often leads to better throughput and responsiveness compared to the static size of a newFixedThreadPool() for such workloads. However, it's important to be mindful of the potential for unbounded thread creation under extreme load, although this is less of a concern for typical I/O-bound scenarios where waiting is the dominant factor.

🚀 When to Use Which Pool?

Workload Type	Best Thread Pool	Why?
CPU-bound (Heavy Computation)	Executors.newFixedThreadPool(n)	Prevents too many threads from overloading the CPU.
I/O-bound (Short-Lived)	Executors.newCachedThreadPool()	Creates threads as needed, preventing delays.
Mixed Workload	Custom pool using ThreadPoolExecutor	Fine-tune thread count, queue size, and timeouts.

1. **Set appropriate queue sizes** to balance memory usage and throughput. 📦

Example Scenario:

- **Too large a queue** → Delayed execution.
- **Too small a queue** → Frequent task rejections.

Java

```java
ExecutorService executor = new ThreadPoolExecutor(
        4, 8, 30L, TimeUnit.SECONDS,
        new LinkedBlockingQueue<>(10)); // Balanced queue size
```

2.Name your threads for easier debugging and monitoring. 🏷️
Example :

Java

```java
ExecutorService executor = Executors.newFixedThreadPool(2, runnable -> {
    Thread thread = new Thread(runnable);
    thread.setName("CustomThread-" + thread.getId());
    return thread;
});
```

Interview Questions 🎯

1. What happens to a thread in a thread pool after it finishes executing a task?
Answer: After task completion, the thread doesn't terminate but returns to the pool, ready to execute another task. This reuse eliminates the overhead of constantly creating and destroying threads. ♻️

2. How does a ThreadPoolExecutor's queue size affect its behavior?
Answer: The queue stores tasks when all core threads are busy. A larger queue can handle more pending tasks but consumes more memory. If the queue reaches capacity, the pool creates additional threads up to maxPoolSize. If maxPoolSize is reached and the queue is full, the rejection policy is applied. 📊

Java

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

public class ThreadPoolQueueExample {
    public static void main(String[] args) {
        // ThreadPoolExecutor with 2 core threads, max 4 threads, and queu
        ThreadPoolExecutor executor = new ThreadPoolExecutor(
                2, 4, 10, TimeUnit.SECONDS, new ArrayBlockingQueue<>(2),
                new ThreadPoolExecutor.AbortPolicy() // Reject tasks if qu
        );

        for (int i = 1; i <= 10; i++) {
            final int taskId = i;
            executor.execute(() -> {
                System.out.println(Thread.currentThread().getName() + " is
```

```
                try {
                    Thread.sleep(2000); // Simulating task execution
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            });
        }
        executor.shutdown();
    }
}
```

Behavior Based on Queue Size :

Small Queue (Size = 2) :

- The **first 2 tasks** are assigned to **core threads** (Thread-1, Thread-2).
- The **next 2 tasks** wait in the **queue**.
- When the queue fills up, **new threads (up to maxPoolSize = 4) are created**.
- If all **4 threads are busy** and **queue is full**, new tasks are **rejected** (handled by AbortPolicy).

✅ **Outcome**: Faster execution due to additional threads, but at the cost of **higher CPU load**.

Large Queue (Size = 6) :

- The **first 2 tasks** are executed by **core threads**.
- The **next 6 tasks** are **queued** instead of creating new threads.
- Additional threads **are not created until the queue is full**.

✅ **Outcome**: Less CPU usage, but **tasks may take longer to start.**

3. What is the difference between shutdown() and shutdownNow()?
Answer: shutdown() initiates a graceful shutdown, allowing queued tasks to complete but not accepting new tasks. shutdownNow() attempts to stop all executing tasks immediately and returns a list of tasks that were awaiting execution. 🛑

Java

```
import java.util.List;
import java.util.concurrent.ExecutorService;
```

```java
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;


public class ShutdownExample {
    public static void main(String[] args) throws InterruptedException {
        // Example 1: Using shutdown()
        System.out.println("EXAMPLE 1: shutdown()");
        ExecutorService executor1 = Executors.newFixedThreadPool(2);

        for (int i = 1; i <= 5; i++) {
            final int taskId = i;
            executor1.submit(() -> {
                try {
                    System.out.println("Task " + taskId + " started");
                    // Simulate work
                    TimeUnit.SECONDS.sleep(2);
                    System.out.println("Task " + taskId + " completed");
                    return "Result of Task " + taskId;
                } catch (InterruptedException e) {
                    System.out.println("Task " + taskId + " was interrupte
                    return null;
                }
            });
        }

        // Allow some tasks to start
        TimeUnit.SECONDS.sleep(1);

        // Initiate graceful shutdown
        System.out.println("nCalling shutdown()...");
        executor1.shutdown();

        System.out.println("Is shutdown: " + executor1.isShutdown());
        System.out.println("Is terminated: " + executor1.isTerminated());
        System.out.println("Can submit new tasks? " + !executor1.isShutdow

        // Wait for tasks to complete
        boolean tasksCompleted = executor1.awaitTermination(10, TimeUnit.S
        System.out.println("All tasks completed: " + tasksCompleted);
        System.out.println("Is terminated now: " + executor1.isTerminated(
```

```
44
45          // Example 2: Using shutdownNow()
46          System.out.println("nnEXAMPLE 2: shutdownNow()");
47          ExecutorService executor2 = Executors.newFixedThreadPool(2);
48
49          for (int i = 1; i <= 5; i++) {
50              final int taskId = i;
51              executor2.submit(() -> {
52                  try {
53                      System.out.println("Task " + taskId + " started");
54                      // Simulate work
55                      TimeUnit.SECONDS.sleep(5);
56                      System.out.println("Task " + taskId + " completed");
57                      return "Result of Task " + taskId;
58                  } catch (InterruptedException e) {
59                      System.out.println("Task " + taskId + " was interrupte
60                      return null;
61                  }
62              });
63          }
64
65          // Allow some tasks to start
66          TimeUnit.SECONDS.sleep(1);
67
68          // Immediate shutdown - return list of waiting tasks
69          System.out.println("nCalling shutdownNow()...");
70          List<Runnable> pendingTasks = executor2.shutdownNow();
71
72          System.out.println("Is shutdown: " + executor2.isShutdown());
73          System.out.println("Number of pending tasks that never started: "
74
75          // Wait for executing tasks to respond to interruption
76          executor2.awaitTermination(5, TimeUnit.SECONDS);
77          System.out.println("Is terminated now: " + executor2.isTerminated(
78      }
79  }
```

Output :

```
EXAMPLE 1: shutdown()
Task 1 started
Task 2 started

Calling shutdown()...
Is shutdown: true
Is terminated: false
Can submit new tasks? false
Task 1 completed
Task 2 completed
Task 3 started
Task 4 started
Task 3 completed
Task 4 completed
Task 5 started
Task 5 completed
All tasks completed: true
Is terminated now: true

EXAMPLE 2: shutdownNow()
Task 1 started
Task 2 started

Calling shutdownNow()...
Is shutdown: true
Number of pending tasks that never started: 3
Task 1 was interrupted!
Task 2 was interrupted!
Is terminated now: true
```

4. Can a thread in TIMED_WAITING state move directly to TERMINATED state?
Answer: Yes, if the thread is interrupted during TIMED_WAITING, it can throw an InterruptedException and complete its run method, transitioning to TERMINATED state. 🔄

Java

```java
class MyThread extends Thread {
    public void run() {
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            System.out.println("Thread interrupted!");
        }
```

```
    }
}

public class Main {
    public static void main(String[] args) {
        MyThread t = new MyThread();
        t.start();
        t.interrupt(); // Interrupting the sleeping thread
    }
}
```

5. What is thread starvation and how can thread pools help prevent it?

Answer: Thread starvation occurs when threads are unable to gain regular access to shared resources and make progress. Thread pools help prevent this by controlling the number of active threads and implementing fair scheduling policies. ⚖️

Java

```
import java.util.concurrent.*;
import java.util.concurrent.atomic.AtomicInteger;


 - Thread starvation occurs when threads are unable to gain regular access
 - to shared resources, causing some threads to make little or no progress
 */
public class ThreadStarvationExample {

    // Counter to track task completion by priority
    private static AtomicInteger[] completedTasks = new AtomicInteger[3];

    // Initialize counters
    static {
        for (int i = 0; i < completedTasks.length; i++) {
            completedTasks[i] = new AtomicInteger(0);
        }
    }

    public static void main(String[] args) throws InterruptedException {
        System.out.println("--- Example 1: Without Thread Pool (Potential
        withoutThreadPool();
```

```
23
24          // Reset counters
25          for (AtomicInteger counter : completedTasks) {
26              counter.set(0);
27          }
28
29          System.out.println("n--- Example 2: With Thread Pool (Fair Schedul
30          withThreadPool();
31      }
32
33      /**
34       - Example 1: Without Thread Pool
35       - In this approach, we directly create many threads with different pr
36       - High-priority threads can potentially monopolize CPU, causing starv
37       */
38      private static void withoutThreadPool() throws InterruptedException {
39          // Create a resource that will be shared across threads
40          final Object sharedResource = new Object();
41
42          // Create and start a large number of threads with different prior
43          for (int i = 0; i < 30; i++) {
44              Thread thread = new Thread(new PriorityTask(i % 3, sharedResou
45
46              // Set thread priority (highest priority will likely monopoliz
47              thread.setPriority(Thread.MIN_PRIORITY + (i % 3) * 2);
48              thread.start();
49          }
50
51          // Wait to see results
52          Thread.sleep(5000);
53
54          // Display results
55          System.out.println("Tasks completed by priority:");
56          System.out.println("Low priority: " + completedTasks[0].get());
57          System.out.println("Medium priority: " + completedTasks[1].get());
58          System.out.println("High priority: " + completedTasks[2].get());
59      }
60
61      /**
62       - Example 2: With Thread Pool
63       - Thread pools help prevent starvation by:
```

```
     - 1. Limiting the total number of active threads
     - 2. Using queue-based scheduling which can be fair
     - 3. Allowing task execution order to be controlled
     */
    private static void withThreadPool() throws InterruptedException {
        // Create a resource that will be shared across threads
        final Object sharedResource = new Object();

        // Create thread pool with FIFO scheduling using a fair lock
        ThreadPoolExecutor executor = new ThreadPoolExecutor(
            4,                          // Core pool size
            4,                          // Maximum pool size
            0, TimeUnit.MILLISECONDS,   // Keep-alive time
            new LinkedBlockingQueue<>(),// Work queue (FIFO)
            new ThreadPoolExecutor.CallerRunsPolicy()  // Rejection policy
        );

        // Set fair scheduling (helps prevent starvation)
        executor.setThreadFactory(r -> {
            Thread t = new Thread(r);
            t.setDaemon(true);
            return t;
        });

        // Submit tasks (same number as previous example)
        for (int i = 0; i < 30; i++) {
            executor.submit(new PriorityTask(i % 3, sharedResource));
        }

        // Wait to see results
        Thread.sleep(5000);

        // Display results
        System.out.println("Tasks completed by priority:");
        System.out.println("Low priority: " + completedTasks[0].get());
        System.out.println("Medium priority: " + completedTasks[1].get());
        System.out.println("High priority: " + completedTasks[2].get());

        // Shutdown executor
        executor.shutdown();
    }
```

```
105
106     /**
107      - A task that simulates work with different priorities
108      */
109     static class PriorityTask implements Runnable {
110         private final int priority; // 0=Low, 1=Medium, 2=High
111         private final Object sharedResource;
112
113         public PriorityTask(int priority, Object sharedResource) {
114             this.priority = priority;
115             this.sharedResource = sharedResource;
116         }
117
118         @Override
119         public void run() {
120             try {
121                 // Run in a loop to simulate ongoing work
122                 for (int i = 0; i < 10; i++) {
123                     // Simulate accessing a shared resource
124                     synchronized (sharedResource) {
125                         // Higher priority tasks do more work with the res
126                         // This can lead to starvation without proper sche
127                         Thread.sleep(20 + (10 * priority));
128
129                         // Increment counter for this priority level
130                         completedTasks[priority].incrementAndGet();
131                     }
132
133                     // Simulate some computation outside critical section
134                     Thread.sleep(10);
135                 }
136             } catch (InterruptedException e) {
137                 Thread.currentThread().interrupt();
138             }
139         }
140     }
141 }
```

Example Analysis :

Example 1: Without Thread Pool :

In this approach:

- We create 30 threads with varying priorities (low, medium, high)
- Each thread tries to access a shared resource in a synchronized block
- Higher priority threads are favored by the OS scheduler
- This can lead to lower priority threads being "starved" of CPU time

Example 2: With Thread Pool :

In this approach:

- We use a ThreadPoolExecutor with a fixed number of worker threads (4)
- Tasks are submitted to a queue and executed in order
- The thread pool provides fair scheduling
- This ensures all tasks get a fair chance at execution regardless of priority

Expected Output:

For the first example (without thread pool), you'll likely see an uneven distribution of completed tasks with high-priority tasks completing significantly more work:
--- Example 1: Without Thread Pool (Potential Starvation) ---

```
Tasks completed by priority:
Low priority: 24
Medium priority: 42
High priority: 89
```

For the second example (with thread pool), you'll see a much more balanced distribution:
--- Example 2: With Thread Pool (Fair Scheduling) ---

```
Tasks completed by priority:
Low priority: 96
Medium priority: 98
High priority: 101
```

Key Benefits of Thread Pools for Preventing Starvation :

1. **Controlled Concurrency**: By limiting the number of active threads, thread pools prevent resource oversaturation
2. **Fair Scheduling**: Tasks can be queued and executed in a fair order
3. **Work Queue Management**: Different queueing strategies can be employed based on requirements
4. **Resource Management**: Thread pools efficiently reuse threads instead of creating new ones

Conclusion 🎯

Thread pools and thread lifecycle management are fundamental concepts in Java concurrency. By effectively utilizing thread pools, you can create applications that efficiently manage system resources while maintaining control over thread creation and execution. Understanding the thread lifecycle allows you to properly monitor and manage thread states, preventing common concurrency issues like deadlocks, livelocks, and resource starvation. 🚀🧵

As you develop multi-threaded applications, remember that proper thread management is a balance between maximizing performance and ensuring system stability. The Java concurrency utilities provide robust tools for achieving this balance, making complex concurrent programming more accessible and reliable. 💪🔄