Chapter 14: Multithreading and Concurrency

Introduction

Modern applications often require performing multiple tasks simultaneously—be it a web browser loading a page while rendering images, or a video game processing user input, rendering graphics, and playing music concurrently. Multithreading and concurrency are fundamental to achieving such multitasking. This chapter delves deep into the concepts of multithreading, concurrency, and synchronization in programming, especially in Java and other object-oriented languages. It lays the foundation for building efficient, scalable, and responsive systems.

14.1 Overview of Multithreading

What is a Thread?

A **thread** is the smallest unit of execution in a process. A process may have one or multiple threads that share the same memory space but execute independently.

Single-threaded vs. Multi-threaded Applications

- **Single-threaded**: Executes tasks sequentially. Less overhead, but poor responsiveness.
- **Multi-threaded**: Executes multiple tasks concurrently using different threads. Improves responsiveness and performance on multi-core processors.

14.2 Life Cycle of a Thread

The life cycle of a thread includes:

1. **New** – Thread object is created but not started.
2. **Runnable** – Thread is ready to run and waiting for CPU.
3. **Running** – Thread is currently executing.
4. **Blocked/Waiting** – Thread is waiting for a resource or signal.
5. **Terminated/Dead** – Thread has finished execution or been stopped.

14.3 Creating Threads in Java

Using the `Thread` Class:

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread is running");
```

```
    }
}
MyThread t1 = new MyThread();
t1.start();
```

Using the `Runnable` Interface:

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Runnable thread is running");
    }
}
Thread t2 = new Thread(new MyRunnable());
t2.start();
```

14.4 Thread Methods

Commonly used methods in the `Thread` class:

- `start()` – Starts the thread.
- `run()` – Contains the code executed by the thread.
- `sleep(ms)` – Pauses the thread for specified milliseconds.
- `join()` – Waits for the thread to finish.
- `yield()` – Suggests the thread scheduler to pause current thread and allow others to execute.
- `interrupt()` – Interrupts the thread.

14.5 Thread Priorities and Scheduling

- Threads can be assigned priorities using `setPriority()`.
- JVM schedules threads based on priority but exact behavior depends on the OS scheduler.
- Priorities: `MIN_PRIORITY` (1), `NORM_PRIORITY` (5), `MAX_PRIORITY` (10).

14.6 Concurrency and Parallelism

Concurrency:

Multiple tasks are in progress at the same time (context switching). It may run on a single CPU core.

Parallelism:

Tasks are literally executed at the same time using multiple CPU cores.

Concurrency ≠ Parallelism, but they can coexist.

14.7 Synchronization

When multiple threads access shared resources (variables, files, databases), data inconsistency may arise. **Synchronization** ensures that only one thread accesses a critical section at a time.

Synchronized Methods:

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

Synchronized Blocks:

```
synchronized(this) {
    // critical section
}
```

14.8 Inter-thread Communication

Threads can communicate using `wait()`, `notify()`, and `notifyAll()` methods from the `Object` class.

Example:

```
synchronized(obj) {
    obj.wait(); // thread waits
    obj.notify(); // wakes one waiting thread
}
```

This is used in producer-consumer problems, thread coordination, etc.

14.9 Deadlock and Its Avoidance

Deadlock:

Occurs when two or more threads are blocked forever, each waiting for the other to release a lock.

Example:

Thread A locks Resource 1 and waits for Resource 2; Thread B locks Resource 2 and waits for Resource 1.

Avoiding Deadlock:

- Lock ordering
- Timeout for lock acquisition
- Using try-lock mechanisms (e.g., `ReentrantLock.tryLock()`)

14.10 Thread-safe Collections and Concurrent Utilities

Java provides thread-safe collections and utilities in `java.util.concurrent` package.

Key Classes:

- `ConcurrentHashMap`
- `CopyOnWriteArrayList`
- `BlockingQueue`
- `ExecutorService`
- `Semaphore`, `CountDownLatch`, `CyclicBarrier`

These allow high-performance concurrent programming without manually handling synchronization in many cases.

14.11 Executors and Thread Pools

Executors:

Provide a flexible way to manage and reuse threads via thread pools.

```
ExecutorService executor = Executors.newFixedThreadPool(5);
executor.execute(new Task());
executor.shutdown();
```

Types:

- FixedThreadPool
- CachedThreadPool
- SingleThreadExecutor
- ScheduledThreadPool

Advantages:

- Better resource management
- Avoids thread exhaustion
- Reduces latency from frequent thread creation/destruction

14.12 Atomic Variables

`java.util.concurrent.atomic` Package:

Provides lock-free, thread-safe operations on single variables.

Examples:

- `AtomicInteger`
- `AtomicBoolean`
- `AtomicLong`

```
AtomicInteger counter = new AtomicInteger(0);
counter.incrementAndGet();
```

14.13 Fork/Join Framework (Advanced Topic)

Used for **divide-and-conquer** style parallelism.

Example:

Breaking a task into smaller subtasks recursively (e.g., merge sort), running them in parallel using `ForkJoinPool`.

```
ForkJoinPool pool = new ForkJoinPool();
pool.invoke(new RecursiveTaskImpl());
```

14.14 Best Practices for Multithreading

- Minimize synchronization to reduce contention.
- Prefer concurrent utilities over manual synchronization.
- Avoid sharing mutable state.
- Always shut down executor services.
- Use thread-safe data structures.
- Avoid unnecessary thread creation.
- Use thread pools for large-scale task execution.
- Use profiling tools to detect deadlocks and performance bottlenecks.

Summary

In this chapter, we explored the core principles and techniques of multithreading and concurrency—vital for modern, high-performance applications. From thread creation to inter-thread communication, synchronization, deadlocks, and advanced concurrency utilities, mastering these topics enables developers to write scalable, efficient, and responsive programs.

With growing demand for concurrent applications across domains—from backend systems to mobile apps and AI—understanding concurrency is not just useful, but essential for all serious programmers.
