Advanced Java Programming Course

Java Atomic Operations and Classes

Faculty of Information Technologies
Industrial University of Ho Chi Minh City

Introduction

- Atomic operations are those that can be performed atomically without interference from other threads.
- These operations are typically used when dealing with shared data that can be accessed and modified by multiple threads concurrently.
- An atomic operation is one that effectively happens all at once or it doesn't happen at all
- Any side effects of an atomic operation aren't visible until the operation completes

Key Concepts Related to Java Atomic Operations

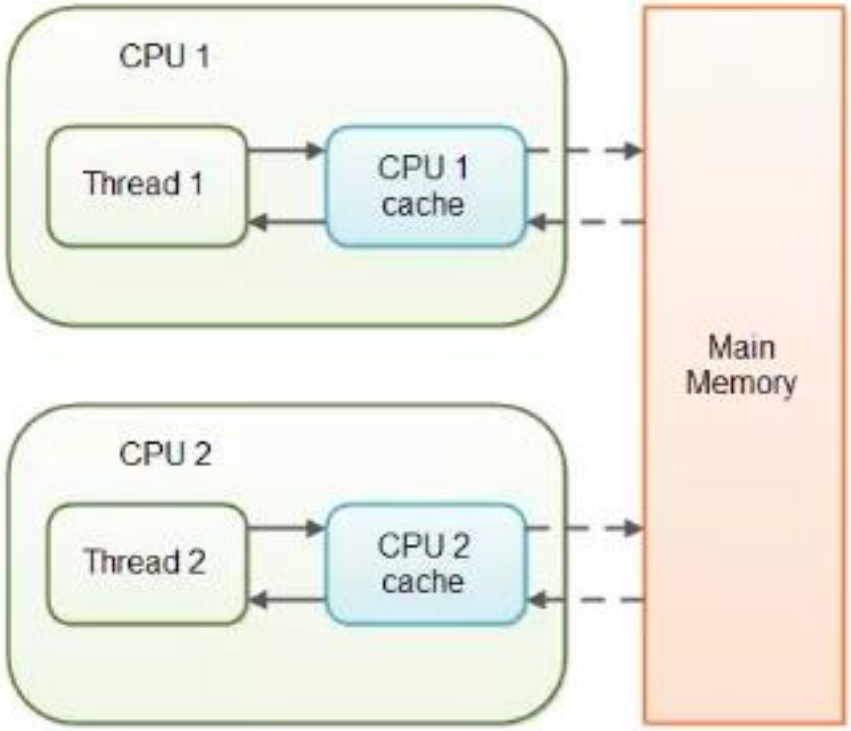
- Three key concepts are associated with atomic operations in Java
  - **Atomicity**: An atomic operation is an operation that is performed as a single unit of work without the possibility of interference from other operations.
  - **Visibility**: An atomic operation has full visibility of the effects of the operation by other threads.
  - **Ordering**: An atomic operation can be ordered concerning another atomic operation.

Java Atomic Operations & Variables

- Java supports several types of atomicity
  - Volatile variables
  - Low-level atomic operations in the Java Unsafe class
  - Atomic classes

Java Atomic Operations & Variables

- Java supports several types of atomicity
  - **Volatile variables:** Ensure a variable is read from & written to main memory & not cached



Java Atomic Operations & Variables

- Java supports several types of atomicity
  - Low-level atomic operations in the Java Unsafe class. It's designed for use only by the Java Class Library, not by normal app programs

Java Atomic Operations & Variables

- Java supports several types of atomicity
  - Atomic classes. A small toolkit of classes that support lock-free thread-safe programming on single variables.
  - https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/package-summary.html

Atomic class example

```
public class CounterApp {

    public static void main(String[] args) {
        Counter counter = new Counter();
        Runnable task = () -> {
                counter.increment();
        };

        ExecutorService executorService = java.util.concurrent.Executors.newFixedThreadPool(10);
        for (int i = 0; i < 1000; i++) {
            executorService.submit(task);
        }
        for (int i = 0; i < 1000; i++) {
            executorService.submit(task);
        }
        executorService.shutdown();
        while (!executorService.isTerminated()) {
        }
        System.out.println("Count = " + counter.getCount());
    }
}

class Counter {
    private AtomicInteger count = new AtomicInteger(0);

    public int increment() {
        return count.incrementAndGet();
    }

    public int getCount() {
        return count.get();
    }
}
```