Concurrency and Asynchrony

مصطفی نصرتی پور

دانشگاه بهار

**همزمانی و ناهمزمانی** مفاهیمی اساسی در برنامه‌نویسی هستند که به ویژه در زبان‌هایی مانند جاوا که برای ساخت برنامه‌های پیچیده و چندنخی استفاده می‌شوند، اهمیت ویژه‌ای دارند. در این مقاله، به بررسی مفاهیم همزمانی و ناهمزمانی در جاوا، کاربردها و چالش‌های آن می‌پردازیم.

**همزمانی (Concurrency) چیست؟**

همزمانی به توانایی یک سیستم برای اجرای چندین کار به طور همزمان اشاره دارد. این به معنای اجرای واقعی چندین کار در یک لحظه نیست، بلکه به این معنی است که چندین کار به گونه‌ای برنامه‌ریزی شده‌اند که گویی به طور همزمان در حال اجرا هستند. در جاوا، این کار معمولاً با استفاده از نخ‌ها (Threads) انجام می‌شود.

**ناهمزمانی (Asynchrony) چیست؟**

ناهمزمانی به این معنا است که یک کار بدون انتظار برای تکمیل کار دیگری اجرا می‌شود. این به این معنی است که یک برنامه می‌تواند به انجام کارهای دیگر بپردازد در حالی که یک عملیات طولانی مدت در پس‌زمینه در حال اجرا است. ناهمزمانی اغلب با همزمانی همراه می‌شود و به برنامه‌ها اجازه می‌دهد تا پاسخگوتر و کارآمدتر باشند.

### چرا همزمانی و ناهمزمانی در جاوا مهم هستند؟

* **پاسخگویی (Responsiveness):** برنامه‌های همزمان و ناهمزمان می‌توانند به ورودی‌های کاربر به سرعت پاسخ دهند و از مسدود شدن رابط کاربری جلوگیری کنند.
* **کارایی (Performance):** با استفاده از چندین نخ، برنامه‌ها می‌توانند از منابع پردازنده به طور موثرتر استفاده کنند و زمان اجرای برنامه را کاهش دهند.
* **پیچیدگی (Complexity):** مدیریت همزمانی و ناهمزمانی می‌تواند پیچیده باشد و نیاز به درک عمیق از مفاهیم همگام‌سازی و مشکلات مرتبط با آن دارد.

### ابزارها و مفاهیم کلیدی در جاوا برای همزمانی و ناهمزمانی

* **نخ‌ها (Threads):** واحدهای اساسی اجرای در جاوا هستند. هر نخ یک جریان کنترل جداگانه است که می‌تواند کد جاوا را به طور مستقل اجرا کند.
* **همگام‌سازی (Synchronization):** مکانیزمی برای کنترل دسترسی چندین نخ به منابع مشترک است. از ابزارهایی مانند قفل‌ها (Locks)، مانیتورها (Monitors) و متغیرهای شرطی (Condition Variables) برای همگام‌سازی استفاده می‌شود.
* **Concurrent Collections:** مجموعه‌هایی که برای استفاده در محیط‌های چندنخی طراحی شده‌اند و به طور خودکار همگام‌سازی را مدیریت می‌کنند.
* **ExecutorService:** یک رابط برای اجرای وظایف به صورت ناهمزمان است.
* **Future:** یک رابط برای نشان دادن نتیجه یک عملیات ناهمزمان است.
* **CompletableFuture:** یک کلاس قدرتمند برای ترکیب و مدیریت عملیات ناهمزمان است.

**چالش‌های همزمانی و ناهمزمانی**

* **شرایط مسابقه (Race Conditions):** زمانی رخ می‌دهد که دو یا چند نخ به طور همزمان به یک منبع مشترک دسترسی پیدا کنند و نتیجه نهایی به ترتیب دسترسی آن‌ها بستگی داشته باشد.
* **قفل‌شدگی (Deadlocks):** زمانی رخ می‌دهد که دو یا چند نخ به گونه‌ای منتظر یکدیگر باشند که هیچ‌کدام نتوانند پیشرفت کنند.
* **مشکلات حافظه (Memory Consistency Errors):** زمانی رخ می‌دهد که چندین نخ به حافظه مشترک دسترسی دارند و تغییرات یک نخ به طور صحیح توسط نخ‌های دیگر دیده نشود.

مثال ساده از همزمانی در جاوا

public class MyThread extends Thread {

public void run() {

for (int i = 0; i < 5; i++) {

System.out.println("Hello from " + Thread.currentThread().getName());

}

}

public static void main(String[] args) {

MyThread t1 = new MyThread(); MyThread t2 = new MyThread();

t1.start();

t2.start();

}

}

در این مثال، دو نخ به طور همزمان اجرا می‌شوند و هر کدام پیامی را چاپ می‌کنند.

**ابزارهای همزمانی در جاوا: Concurrent Utilities**

**Concurrent Utilities** مجموعه ای از کلاس‌ها و رابط‌های از پیش تعریف شده در جاوا هستند که برای مدیریت همزمانی و ناهمزمانی در برنامه‌های چندنخی طراحی شده‌اند. این ابزارها به توسعه‌دهندگان کمک می‌کنند تا به طور ایمن و کارآمد با چندین نخ کار کنند و از بروز مشکلات رایجی مانند شرایط مسابقه، قفل‌شدگی و خطاهای حافظه جلوگیری کنند.

**چرا به Concurrent Utilities نیاز داریم؟**

* **سادگی:** این ابزارها سطح بالاتری از انتزاع را فراهم می‌کنند و به توسعه‌دهندگان اجازه می‌دهند بدون نگرانی در مورد جزئیات پیاده‌سازی همزمانی، کدهای خود را بنویسند.
* **کارایی:** بسیاری از این ابزارها بهینه شده‌اند و عملکرد بالایی دارند.
* **ایمنی:** استفاده از این ابزارها به طور قابل توجهی احتمال بروز خطاهای همزمانی را کاهش می‌دهد.

**برخی از مهم‌ترین Concurrent Utilities در جاوا**

* **ExecutorService:**
  + یک رابط برای اجرای وظایف به صورت ناهمزمان است.
  + امکان ایجاد ThreadPool و مدیریت اجرای وظایف را فراهم می‌کند.

**مثال:**

ExecutorService executor = Executors.newFixedThreadPool(5);

executor.submit(() -> {

// کد وظیفه

});

**Future**  :

* یک رابط برای نشان دادن نتیجه یک عملیات ناهمزمان است.
* امکان بررسی وضعیت یک وظیفه، دریافت نتیجه و لغو آن را فراهم می‌کند.

**CompletableFuture**  :

* یک کلاس قدرتمند برای ترکیب و مدیریت عملیات ناهمزمان است.
* از مفهوم زنجیره‌ای کردن عملیات پشتیبانی می‌کند و امکان اجرای عملیات به صورت موازی یا سریال را فراهم می‌کند.

**Concurrent Collections**  :

* مجموعه‌هایی مانند ConcurrentHashMap، CopyOnWriteArrayList و ConcurrentLinkedQueue که به طور خاص برای استفاده در محیط‌های چندنخی طراحی شده‌اند و به طور خودکار همگام‌سازی را مدیریت می‌کنند.

**Atomic Variables :**

* متغیرهایی مانند AtomicInteger، AtomicLong و AtomicReference که عملیات خواندن و نوشتن بر روی آن‌ها اتمی است و از بروز مشکلات همزمانی جلوگیری می‌کند.

**Locks**  :

* مکانیزمی برای کنترل دسترسی به منابع مشترک است.
* انواع مختلفی از قفل‌ها مانند ReentrantLock و ReadWriteLock وجود دارد.

**Semaphores**  :

* برای محدود کردن تعداد نخ‌هایی که می‌توانند به یک منبع دسترسی داشته باشند استفاده می‌شود.

**CountDownLatch**  :

* برای هماهنگ کردن چندین نخ استفاده می‌شود.
* زمانی که همه نخ‌ها به یک نقطه مشخص رسیدند، یک نخ دیگر می‌تواند آزاد شود.

**CyclicBarrier**  :

مشابه CountDownLatch است، اما با این تفاوت که پس از رسیدن همه نخ‌ها به یک نقطه مشخص، همه نخ‌ها دوباره شروع به کار می‌کنند.

مثال عملی: استفاده از ExecutorService و Future

import java.util.concurrent.\*;

public class ExecutorExample {

public static void main(String[] args) throws ExecutionException, InterruptedException

{

ExecutorService executor = Executors.newSingleThreadExecutor();

Future<String>

result = executor.submit(() -> {

عملیات طولانی مدت//

Thread.sleep(2000);

return "Hello from another thread";

});

System.out.println("Doing something else while task is running...");

String value = result.get();

System.out.println("Result: " + value); executor.shutdown();

}

}

در این مثال، یک وظیفه به صورت ناهمزمان اجرا می‌شود و در همین حین برنامه می‌تواند به انجام کارهای دیگر بپردازد. پس از پایان یافتن وظیفه، نتیجه آن دریافت می‌شود.