소개

예전에는 컴퓨터 시스템이 운영체제 없이 동작했습니다. 한 번에 하나의 프로그램만 실행할 수 있었고 그 프로그램은 모든 자원에 대한 직접적인 제어권을 가졌습니다.

운영체제가 없는 시스템

이러한 구조의 **단점**은

- 시스템 자원을 공유할 수 없게 했습니다.
- 여러 작업을 동시에 실행하려는 시도도 불가능하게 만들었습니다(모든 자원을 프로그램 하나가 다 써버리니..)

위의 단점들을 보완하기 위해 **운영체제**가 도입되게 되었습니다.

운영체제가 도입되면서, 하나의 시스템에서 여러 프로그램을 **동시**에 실행할 수 있게 되었습니다.

프로세스

각자의 자원을 가진 프로그램들은 **프로세스(process)**라는 단위로 분리되었고, 각 프로세스는 메모리, 파일 핸들, 보안 자격 등의 자원을 독립적으로 가졌습니다.

이러한 프로세스들은, 소켓, 시그널, 공유 메모리, 세마포어, 파일 등을 통해 서로 **통신**할 수 있었습니다.

운영 체제가 동시성을 지원하게 된 이유

운영체제가 동시성을 지원하게 된 주된 이유는 다음과 같습니다.

- **자원 활용률:** 많은 프로그램들은 I/O 작업 등 외부 자원을 기다리느라 멈추게 됩니다. cpu는 아무것도 하지 않고 대기만 하게 되는데, 이 시간에 다른 프로그램에게 CPU를 사용하도록 하면 자원을 훨씬 효율적으로 활용할 수 있습니다.
- **공정성 :** 여러 사용자나 프로그램이 자원을 사용할 수 있도록 하는 것이 좋습니다. 한 프로그램이 너무 오래 CPU를 독점하지 못하도록 시간 분할(time slicing)방식으로 자원을 할당합니다.(라운드 로빈 방식)

- **편의성 :** 때론 여러 작업을 전부 처리하는 프로그램 하나를 작성하는 것보다 각기 일을 하나씩 처리하고 필요할 때 프로그램 간에 조율하는 프로그램을 여러 개 작성하는 것이 더 쉽고 바람직합니다.

1.2 스레드의 이점

적절하게 사용된 스레드는 다음과 같은 **장점**을 가집니다.

- 개발과 유지보수 비용을 줄이고 복잡한 애플리케이션의 성능을 향상시킬 수 있습니다.
- 복잡하게 꼬인 코드를 순차적인 코드로 바꿀 수 있습니다.
- GUI 애플리케이션에서 인터페이스의 반응성을 높이는데 유용합니다.
- 서버 애플리케이션에서는 자원 활용도와 처리량을 높이는데 유용합니다.
- JVM자체의 구현을 단순화하는데도 도움을 줍니다.(즉 JVM 내부도 스레드를 통해 설계되었다)
  EX: 가비지 컬렉터는 보통 하나 이상의 전용 스레드를 사용합니다.

1.2.1 멀티 프로세서 활용

최근에는 컴퓨터에 CPU의 코어 수가 점점 늘어나고 있지만 결국 스케줄링의 기본 단위는 여전히 **스레드**입니다.

- 프로그램이 단 하나의 스레드만 가진다면, 그 프로그램은 오직 하나의 **프로세서(CPU)**에서만 실행될 수 있습니다.
  즉!! 프로그램이 싱글스레드로 사용된다면 코어가 100개여도 단 하나만 사용하는 시스템 리소스 99퍼의 리소스를 버리는 짓을 할 수 있습니다.
- 잘 설계된 멀티스레드 프로그램은 자원을 효과적으로 사용해 처리량을 향상 시킬 수 있습니다.
- 단일 프로세서 시스템에서도 처리량 향상에 기여할 수 있습니다. 단일 스레드 프로그램이 동기식 I/O를 기다릴 때 CPU는 유휴 상태가 되지만, 멀티스레드 프로그램에서는 다른 스레드가 계속 실행될 수 있기 때문!

1.2.2 단순한 모델링

한 종류의 작업만을 순차적으로 처리하는 프로그램은 여러 가지 종류의 작업을 동시에 처리하는 프로그램보다 아래와 같은 **이점**을 가집니다.

- 순차적인 흐름의 착각을 만들어 개발자가 코드를 더 간단하게 작성, 테스트 유지 할 수 있게 해줍니다.
- 스케줄링, 비동기 I/O, 리소스 대기 등의 복잡한 문제를 도메인 로직을 분리시킬 수 있습니다.

(위는 동기적인 프로그램의 장점임)

각 작업에 대해 하나의 스레드를 할당하거나, 시뮬레이션 내 요소마다 스레드를 할당하면 순차적인 흐름의 착각을 제공할 수 있습니다.

(즉 순차적인 것 처럼 보이게 코드를 짜기 쉽게 해준다.. 당연히 비동기와 멀티 프로세스의 이점을 가져가면서!)

1.2.3 단순한 비동기 이벤트 처리

서버 애플리케이션에서는 외부 클라이언트로부터의 **소켓 연결**을 다수 동시에 수락해야 할 수 있다. 이때 각 클라이언트 연결마다 별도의 스레드를 할당하면 구현이 훨씬 쉬워집니다. (스레드마다 클라이언트(사용자)를 분배 한다는 이야기!)

1.2.4 더 빨리 반응하는 사용자 인터페이스

스레드는 **UI 애플리케이션에서 특히 유용합니다**. 긴 작업을 별도 스레드에서 실행해서 UI가 나타나는 시간을 줄일 수 있습니다.

반응성 향상? → 사용자의 증가 → 부자

1.3 스레드 사용의 위험성

스레드를 사용하기 위해서는 높은 수준의 이해와 주의가 필요합니다.

일반적인 개발자도 스레드 안정성에 대해 반드시 인지하고 있어야 합니다.

1.3.1 안전성 위해 요소

충분한 동기화가 없으면, 여러 스레드에서 **실행되는 연산들의 실행 순서가 예측할 수 없고 종종 매우 이상하게 작동**하기 때문입니다.(동시성 문제를 이야기 하는 듯.. 경쟁상태(race condition이나), 공유 참조 문제)

예를 들어 `UnsafeSequence`클래스는 하나의 스레드 환경에서는 문제 없이 작동하지만, 여러 스레드가 동시에 접근하면 중복된 값을 반환하는 문제가 발생합니다.

> UnsafeSequence 같은 클래스, 실무에서 아직도 쓰이냐?

Java의 Unsafe 클래스

- **Oracle Blogs에 따르면** Unsafe 클래스는 CPU 및 하드웨어 기능에 직접 액세스할 수 있도록 해줍니다.
- 이 클래스를 사용하면 객체를 생성하지만 생성자를 실행하지 않을 수도 있습니다.
- 또한, 익명 클래스를 생성하고, 오프-힙 메모리를 수동으로 관리할 수도 있습니다.
- Unsafe 클래스는 많은 주요 프레임워크에서 사용되고 있지만, 액세스하기 쉽다는 비판을 받기도 했습니다.

장점도 있지만 사용하면 안된다. ㅎㅎ 안전성을 포기하게 됨

1.3.2 활동성 문제

안전성은 “나쁜 일이 절대 일어나지 않는 것”

활동성은 “원하는 일은 결국에는 일어나야 한다는 것”

활동성의 실패란 → 더 이상 앞으로 나아갈 수 없는 상태에 빠지는것

ex 데드락

- A 스레드가 B 스레드가 소유한 리소스를 기다리는데 B스레드는 절대 그것을 반환하지 않는 상태

1.3.3 성능 위험

성능(performance)도 중요한 동시성 문제입니다.

성능 문제는 처리 시간, 반응 속도, 처리량(throughput), 자원 소모, 확장성 등 광범위한 주제를 포괄합니다.

그리고 멀티스레드 프로그램은 싱글스레드에서의 성능 문제에 **더해,**

스레드 자체로 인해 생기는 새로운 비용도 안고 갑니다:

- **컨텍스트 스위칭**: 실행 중인 스레드를 중단하고 다른 스레드로 전환할 때 CPU가 낭비된다
- **캐시 손실**: 스레드 전환 시 CPU 캐시 무효화, 메모리 접근 비용 증가
- **스케줄링 오버헤드**: 실행보다는 스레드를 **관리**하는 데 시간이 쓰인다
- **동기화 비용**: 락(lock), 메모리 버스 경쟁, 컴파일러 최적화 제한 등

1.4 스레드는 어디에나!

개발자가 직접적으로 스레드를 만들지 않더라도 **프레임워크가 자동으로 스레드를 생성할 수 있습니다.**

그렇게 자동으로 스레드를 생성할 수 있고 그 **스레드 또한 스레드 안전성을 가져야합니다.**

대부분의 자바 애플리케이션이 멀티스레드이며,이러한 프레임워크들은

공유 상태에 대한 적절한 접근 제어 책임을 개발자로부터 대신해주지 않습니다.

스프링 빈의 동시성 문제:

위의 `OrderService` 빈은 `orderCount`라는 상태를 유지하는 필드를 가지고 있습니다. 여러 스레드가 동시에 `processOrder()` 메서드를 호출하면 `orderCount` 값이 정확하지 않을 수 있습니다 스프링 빈은 싱글턴으로 여러 인스턴스에서 돌려쓰기 때문 ㅎㅎ

```
//ApplicationContext에서 싱글턴으로 관리하는 Bean은 싱글턴이다.
//멀티 스레드에서 동시에 자원에 접근하게 된다면...!
@Component
public class OrderService {
```

```
    private int orderCount = 0;

    public void processOrder() {
        orderCount++;
        // 주문 처리 로직
    }

    public int getOrderCount() {
        return orderCount;
    }
}
```
