

1주차

1. 용어 정리

1. Thread (스레드):

- 자바의 Thread 클래스는 하나의 독립적인 실행 경로를 의미합니다. 여러 스레드를 사용하여 동시에 여러 작업을 처리할 수 있습니다. 각 스레드는 JVM이 관리하며, OS에 의해 실제로 스케줄링됩니다.

2. Runnable (러너블):

- Runnable 인터페이스는 `run()` 메서드 하나만 포함되어 있으며, 스레드에서 실행될 수 있는 코드를 작성하기 위한 인터페이스입니다. Runnable은 상태를 반환하지 않기 때문에, 단순히 실행되는 작업을 정의할 때 사용됩니다.

3. Callable (콜러블):

- Callable은 `call()` 메서드를 구현하여 결과를 반환할 수 있는 작업을 정의하는 인터페이스입니다. Callable을 사용하면 스레드 실행 후 결과를 반환하거나 예외 처리를 할 수 있습니다. Runnable과 달리 Callable은 제네릭을 사용하여 리턴 타입을 지정할 수 있습니다.

4. ExecutorService (익스큐터 서비스):

- 스레드를 직접 관리하지 않고 작업을 제출하여 스레드를 실행하는 프레임워크입니다. ExecutorService는 스레드 풀(thread pool)을 관리하여 필요할 때 스레드를 재사용하거나 할당하여 성능을 향상시킬 수 있습니다.

5. Async (비동기):

- 자바의 비동기 처리로, `@Async`와 같은 애노테이션을 사용해 메서드가 비동기적으로 실행될 수 있도록 합니다. ExecutorService나 `CompletableFuture`와 함께 비동기 로직을 구현하는데 자주 사용됩니다.

6. CompletableFuture (컴플리터블퓨처) :

- 자바 8에서 도입된 `CompletableFuture`는 비동기 작업의 결과를 나타내는 객체입니다. 결과가 완료되거나, 예외가 발생하거나, 결과가 없는 작업이라도, 콜백을 사용해 후속 작업을 정의할 수 있습니다. `CompletableFuture`는 복잡한 비동기 작업을 관리할 때 유용합니다.

7. ThreadLocal (스레드 로컬):

- `ThreadLocal`은 각 스레드에 독립적인 데이터를 저장할 수 있는 클래스입니다. 이를 통해 여러 스레드가 같은 변수명으로 데이터를 보관하더라도 값의 충돌을 방지할 수 있으며, 스레드마다 고유한 데이터를 저장할 때 사용합니다.
-

동기화와 동시성 제어

8. Atomic (CAS):

- Atomic 클래스는 원자적 연산(atomic operations)을 제공하여 동시성 문제를 해결할 수 있습니다. 예를 들어 AtomicInteger는 여러 스레드가 공유 데이터를 변경할 때 발생하는 문제를 피하도록 설계되었습니다. 이 클래스는 CAS(Compare-And-Swap) 알고리즘을 사용하여 동기화 성능을 향상시킵니다.

9. Synchronized (동기화):

- synchronized 키워드는 메서드나 블록에 동기화 잠금을 설정하여 여러 스레드가 동시에 접근하는 것을 막습니다. 이를 통해 스레드가 안전하게 공유 리소스에 접근하도록 보장할 수 있습니다. 단점은 성능 저하가 발생할 수 있다는 점입니다.

10. volatile (변동):

- volatile 키워드는 변수 값이 스레드 간에 즉각적으로 업데이트되도록 보장합니다. 일반 변수는 캐시될 수 있지만 volatile 변수는 항상 메인 메모리에서 읽고 쓰여, 최신 상태를 다른 스레드가 볼 수 있습니다.

병렬 처리를 위한 고급 도구

11. ForkJoinPool (포크조인 풀):

- ForkJoinPool은 큰 작업을 작은 단위로 나누고(포크) 결과를 합치는(조인) 병렬 처리 프레임워크입니다. 자바의 ForkJoinPool은 특정 작업을 작은 조각으로 나누어 병렬로 처리한 후 결과를 결합하는 작업에 적합합니다.

12. BlockingDeque (블로킹 덱):

- BlockingDeque는 이중 연결 리스트 형태의 큐로, 스레드가 안전하게 작업을 추가하거나 삭제할 수 있도록 설계된 구조입니다. 작업 큐가 비었거나 꽉 찼을 때 스레드를 블로킹시키는 기능이 있어, 생산자-소비자 패턴에 유용하게 사용됩니다.

2. JVM에서 Thread의 동작방식

JVM에서 스레드의 동작 방식은 다음과 같은 순서로 진행됩니다.

1. 스레드 생성:

- 사용자가 새로운 스레드를 생성하면 JVM은 OS에게 새로운 스레드를 요청합니다.
- OS는 각 스레드에 별도의 스택과 레지스터 등을 할당합니다. 이를 통해 각 스레드가 독립적인 실행 경로를 가질 수 있게 됩니다.

2. 스케줄링:

- JVM은 OS의 스케줄러를 통해 스레드를 관리합니다. OS 스케줄러는 각 스레드가 CPU 자원을 얼마나, 언제 사용할지 결정합니다.

- 스케줄링 방식은 OS에 따라 다르지만, 보통 우선순위, 상태, 그리고 스레드의 작업 시간 등을 기준으로 CPU 할당이 이루어집니다.

3. 컨텍스트 스위칭:

- 여러 스레드가 동시에 CPU를 사용할 수 없으므로, 각 스레드는 컨텍스트 스위칭(Context Switching)이라는 과정을 통해 CPU 자원을 순차적으로 사용하게 됩니다.
- 이때 각 스레드의 레지스터, 프로그램 카운터, 스택 정보 등이 저장되고 복원되며, 이는 멀티스레딩의 비용이 됩니다.

4. 종료:

- 스레드가 모든 작업을 마치면 종료 상태로 진입합니다. JVM은 자동으로 사용이 끝난 스레드의 자원을 정리합니다.

JVM에서 스레드는 OS가 제공하는 스레드를 활용하여 동작합니다. JVM은 OS의 스레드를 추상화하여, 자바 및 코틀린과 같은 JVM 기반 언어에서 스레드를 쉽게 사용할 수 있게 합니다. 여기서 JVM이 스레드를 관리하는 방식과 주요 동작 흐름을 간단히 설명한 후, 코틀린 예제 코드를 통해 스레드를 다루는 방법을 보여드리겠습니다.

JVM의 스레드 관리 방식

JVM에서 스레드의 동작 방식은 다음과 같은 순서로 진행됩니다.

1. 스레드 생성:

- 사용자가 새로운 스레드를 생성하면 JVM은 OS에게 새로운 스레드를 요청합니다.
- OS는 각 스레드에 별도의 스택과 레지스터 등을 할당합니다. 이를 통해 각 스레드가 독립적인 실행 경로를 가질 수 있게 됩니다.

2. 스케줄링:

- JVM은 OS의 스케줄러를 통해 스레드를 관리합니다. OS 스케줄러는 각 스레드가 CPU 자원을 얼마나, 언제 사용할지 결정합니다.
- 스케줄링 방식은 OS에 따라 다르지만, 보통 우선순위, 상태, 그리고 스레드의 작업 시간 등을 기준으로 CPU 할당이 이루어집니다.

3. 컨텍스트 스위칭:

- 여러 스레드가 동시에 CPU를 사용할 수 없으므로, 각 스레드는 컨텍스트 스위칭(Context Switching)이라는 과정을 통해 CPU 자원을 순차적으로 사용하게 됩니다.
- 이때 각 스레드의 레지스터, 프로그램 카운터, 스택 정보 등이 저장되고 복원되며, 이는 멀티스레딩의 비용이 됩니다.

4. 종료:

- 스레드가 모든 작업을 마치면 종료 상태로 진입합니다. JVM은 자동으로 사용이 끝난 스레드의 자원을 정리합니다.

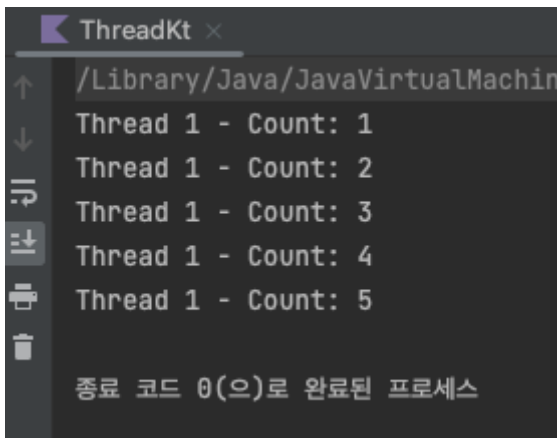
코틀린에서 스레드 생성 및 관리 예시

아래 예제에서는 코틀린에서 `Thread`, `Runnable`, 그리고 `ExecutorService`를 사용하여 스레드를 다루는 방법을 보여줍니다. 이 예제는 여러 스레드가 각각 독립적으로 숫자를 출력하는 작업을 수행하는 코드입니다.

1. 기본 스레드 생성 및 실행

```
fun main() {  
    // 새로운 스레드를 생성하고 시작  
    val thread = Thread {  
        for (i in 1..5) {  
            println("Thread 1 - Count: $i")  
            Thread.sleep(500) // 0.5초 대기  
        }  
    }  
    thread.start()  
}
```

이 코드는 하나의 스레드를 생성하여 숫자를 출력하는 작업을 수행합니다. `Thread` 생성자에 `Runnable` 랩다를 전달하여 실행할 코드를 지정합니다.



2. `Runnable` 인터페이스 사용

```
class PrintTask : Runnable {  
    override fun run() {  
        for (i in 1..5) {  
            println("Runnable Thread - Count: $i")  
            Thread.sleep(500)  
        }  
    }  
}  
  
fun main() {  
    val runnableThread = Thread(PrintTask())  
}
```

```
runnableThread.start()
}
```

여기서는 Runnable 인터페이스를 구현한 PrintTask 클래스를 사용해 스레드를 생성합니다. 이 방식은 스레드 로직을 다른 클래스에서 재사용할 수 있는 장점이 있습니다.

```
/Library/Java/JavaVirtualMachines
Runnable Thread - Count: 1
Runnable Thread - Count: 2
Runnable Thread - Count: 3
Runnable Thread - Count: 4
Runnable Thread - Count: 5

종료 코드 0(으)로 완료된 프로세스
```

3. ExecutorService 를 사용한 스레드 풀 관리

스레드 풀이란 여러 스레드를 미리 생성하여 필요할 때마다 재사용하는 방식입니다. ExecutorService 를 사용하면 스레드 풀을 관리할 수 있습니다.

```
import java.util.concurrent.Executors

fun main() {
    // 스레드 풀 생성 (3개의 스레드)
    val executorService = Executors.newFixedThreadPool(3)

    for (i in 1..5) {
        val task = Runnable {
            println("ExecutorService Thread - Task $i")
            Thread.sleep(500)
        }
        executorService.execute(task)
    }

    // 모든 작업이 끝나면 스레드 풀 종료
    executorService.shutdown()
}
```

이 예제에서는 newFixedThreadPool(3) 로 3개의 스레드를 가지는 스레드 풀을 생성했습니다. 작업이 요청되면 스레드가 할당되어 작업을 수행하고, 작업이 끝난 스레드는 재사용됩니다.

```
/Library/Java/JavaVirtualMachines/jdk-
ExecutorService Thread - Task 1
ExecutorService Thread - Task 2
ExecutorService Thread - Task 3
ExecutorService Thread - Task 4
ExecutorService Thread - Task 5

종료 코드 0(으)로 완료된 프로세스
```

4. ThreadLocal 을 사용한 스레드별 데이터 저장

ThreadLocal 을 사용하면 각 스레드가 고유한 값을 저장할 수 있습니다. 아래 코드는 각 스레드가 독립적으로 값을 유지하도록 합니다.

```
val threadLocalData = ThreadLocal<Int>()

fun main() {
    val thread1 = Thread {
        threadLocalData.set(100)
        println("Thread 1 - Value: ${threadLocalData.get()}")
    }

    val thread2 = Thread {
        threadLocalData.set(200)
        println("Thread 2 - Value: ${threadLocalData.get()}")
    }

    thread1.start()
    thread2.start()
}
```

이 코드에서 ThreadLocal 을 사용하여 각 스레드가 고유한 값을 가지게 하였으며, 다른 스레드와 값이 혼동되지 않도록 합니다.

```
/Library/Java/JavaVirtualMachines
Thread 1 - Value: 100
Thread 2 - Value: 200

종료 코드 0(으)로 완료된 프로세스
```

그렇다면 질문

```
val threadLocalData = ThreadLocal<Int>()

fun main() {

    var testNum = 0
    val thread1 = Thread {
        threadLocalData.set(100)
        testNum = 100
        println("Thread 1 - Value: ${threadLocalData.get()}")
        println("1st$testNum")
    }

    val thread2 = Thread {
        testNum = 200
        threadLocalData.set(200)
        println("Thread 2 - Value: ${threadLocalData.get()}")
        println("2nd$testNum")
    }

    thread1.start()
    thread2.start()
    println("3rd$testNum")
}
```

이렇게 코드가 진행이되면 어떻게 출력이 될까?

정답은



```
/Library/Java/JavaVirtualMac
3rd0
Thread 1 - Value: 100
1st100
Thread 2 - Value: 200
2nd100

종료 코드 0(으)로 완료된 프로세스
```

이지만!!! 스케줄러에 따라서 다르게 출력될 수 있다

이 코드에서 각 스레드(thread1 과 thread2)가 testNum 과 threadLocalData 에 서로 다른 값을 설정 하지만, 그 설정이 메인 스레드에서의 println("3rd\$testNum") 출력에 영향을 주지 않는 이유와 출력

순서가 예상과 다르게 나타나는 이유를 설명드리겠습니다.

출력 순서와 `testNum` 값의 불확정성

코드에서 `println("3rd$testNum")` 는 메인 스레드에서 실행됩니다. 스레드들이 비동기적으로 동작하기 때문에, `thread1` 과 `thread2` 가 시작된 후 메인 스레드가 바로 `println("3rd$testNum")` 을 실행할 가능성이 높습니다. 이로 인해 예상한 순서와는 다르게 출력될 수 있습니다.

코드 분석

- `thread1` 과 `thread2` 는 각각 다른 값으로 `testNum` 을 설정하지만, 두 스레드 간의 실행 순서는 비결정적입니다. 이로 인해 `testNum` 의 최종 값은 `thread1` 과 `thread2` 의 실행 순서에 따라 달라질 수 있습니다.
- `ThreadLocal` 변수인 `threadLocalData` 는 각 스레드가 고유하게 접근할 수 있는 변수를 생성하기 때문에, `threadLocalData.set()` 에 의해 설정된 값은 각 스레드 내부에서만 사용되고 다른 스레드에는 영향을 미치지 않습니다.

예상 가능한 실행 결과 예시

실제 결과는 JVM 스케줄러에 따라 다를 수 있지만, 가능한 출력 예시는 다음과 같습니다:

```
3rd0           // 메인 스레드가 가장 먼저 실행됨
Thread 2 - Value: 200
2nd200
Thread 1 - Value: 100
1st100
```

이유 설명

- `thread1` 과 `thread2` 는 메인 스레드와 비동기적으로 동작하므로, 메인 스레드는 두 스레드가 값을 설정하기 전 초기 값인 `0` 을 `println("3rd$testNum")` 에서 출력할 수 있습니다.
- `threadLocalData` 는 각 스레드에 독립적으로 값이 설정되므로, `Thread 1` 과 `Thread 2` 에서 각각 `100` 과 `200` 으로 설정된 값이 출력되며, 이는 서로에게 영향을 주지 않습니다.
- `testNum` 은 메인 스레드와 공유되는 변수이므로, 두 스레드가 값을 바꾸면서 출력할 때 최종 값이 `200` 일지, `100` 일지는 실행 순서에 따라 달라질 수 있습니다.

해결 방안: `join()` 을 통한 순서 제어

만약 출력 순서를 보장하고 싶다면 `join()` 을 사용하여 메인 스레드가 두 스레드의 작업이 완료될 때까지 기다리게 할 수 있습니다:


```

fun main() {
    var testNum = 0
    val thread1 = Thread {
        threadLocalData.set(100)
        testNum = 100
        println("Thread 1 - Value: ${threadLocalData.get()}")
        println("1st$testNum")
    }

    val thread2 = Thread {
        testNum = 200
        threadLocalData.set(200)
        println("Thread 2 - Value: ${threadLocalData.get()}")
        println("2nd$testNum")
    }

    thread1.start()
    thread2.start()

    thread1.join() // thread1이 끝날 때까지 대기
    thread2.join() // thread2가 끝날 때까지 대기

    println("3rd$testNum") // thread1과 thread2의 작업이 끝난 후 출력
}

```

이렇게 하면 `println("3rd$testNum")` 이 `thread1` 과 `thread2` 가 완료된 후 실행되어, 두 스레드의 작업 후 `testNum` 의 최종 값을 출력하게 됩니다.

3. 컨텍스트 스위칭 비용이란?

컨텍스트 스위칭 비용(**Context Switching Cost**)은 하나의 스레드 또는 프로세스가 실행 중인 상태에서 다른 스레드나 프로세스로 전환될 때 발생하는 시스템 자원의 소모를 의미합니다. 컨텍스트 스위칭은 CPU가 여러 작업을 동시에 수행할 때 중요한 역할을 하지만, 전환 작업 자체에도 시간이 필요하고 자원이 소모되기 때문에 일정한 비용이 발생합니다.

컨텍스트 스위칭 과정

1. **현재 상태 저장:** 현재 실행 중인 스레드나 프로세스의 레지스터 값, 프로그램 카운터(PC), 스택 포인터와 같은 정보가 저장됩니다. 이 정보를 **Context**라고 부릅니다.
2. **새로운 상태 로드:** 전환 대상 스레드 또는 프로세스의 Context가 로드됩니다. 이때 새로운 작업에 맞는 레지스터, 프로그램 카운터 등이 메모리에서 CPU로 복원됩니다.
3. **전환 실행:** 새로운 작업이 시작됩니다. 이제 CPU는 새 작업에 할당되며 이전 작업은 일시 중단된 상태가 됩니다.

컨텍스트 스위칭 비용 요소

컨텍스트 스위칭의 비용은 다음과 같은 이유로 발생합니다:

- **메모리 접근 비용:** Context를 저장하고 로드하는 과정에서 메모리 접근이 이루어지므로, 캐시 메모리 미스(cache miss)가 발생할 가능성이 높습니다. 이는 CPU의 처리 속도를 저하시킵니다.
- **캐시 무효화:** 스레드가 전환되면 CPU 캐시에 있던 이전 스레드 관련 데이터는 무효화됩니다. 따라서 새 스레드에 맞는 데이터를 다시 로드해야 하고, 캐시 히트를 줄여 성능 저하가 발생할 수 있습니다.
- **커널 모드 전환 비용:** 스레드나 프로세스 간 전환 시에는 커널 모드로 전환되는 경우가 많습니다. 이는 사용자 모드에서 커널 모드로, 그리고 다시 커널 모드에서 사용자 모드로 전환하는 과정에서 오버헤드가 발생합니다.
- **스케줄링 알고리즘:** CPU 스케줄러가 어떤 스레드를 실행할지 결정하는데도 시간이 필요합니다. 스케줄링 알고리즘에 따라 이 비용은 다르게 나타납니다.

컨텍스트 스위칭 비용을 줄이는 방법

컨텍스트 스위칭 비용을 줄이기 위해 다양한 최적화 기법이 사용됩니다:

- **스레드 풀 사용:** 새로운 스레드를 생성하지 않고 미리 생성된 스레드를 재사용함으로써 컨텍스트 스위칭을 줄일 수 있습니다.
- **비동기 처리:** 스레드 기반 처리 대신, 이벤트 기반이나 비동기 처리 방식을 사용하여 컨텍스트 스위칭을 최소화할 수 있습니다.
- **스케줄링 최적화:** CPU 스케줄러를 효율적으로 설계하여, 동일한 작업에 할당된 스레드 간의 전환을 줄이는 방식으로 비용을 절감할 수 있습니다.

컨텍스트 스위칭은 멀티스레딩과 멀티태스킹 환경에서 필연적이지만, 너무 잦은 컨텍스트 스위칭은 시스템 성능을 저하시키므로 이 비용을 줄이는 것이 중요한 최적화 포인트가 됩니다.

4. 병렬 프로그램시 알아야할 인프라 리소스

병렬 프로그램을 개발할 때 성능을 극대화하기 위해 알아야 할 주요 인프라 리소스는 **CPU, 메모리, 스토리지, 네트워크** 등입니다. 각각의 리소스가 병렬 처리를 수행하는 데 어떤 영향을 미치는지, 그리고 병렬 프로그램 최적화를 위해 고려해야 할 사항들을 설명드리겠습니다.

1. CPU (Central Processing Unit)

CPU는 병렬 프로그램의 성능에 가장 큰 영향을 미칩니다. 병렬 처리를 위해 여러 개의 CPU 코어가 사용되며, 작업을 동시에 실행할 수 있는 스레드 수를 결정하는 중요한 리소스입니다.

- **코어 수:** 병렬 프로그램은 여러 코어에서 동시에 실행될 수 있으므로, 코어 수가 많을수록 더 많은 작업을 병렬로 처리할 수 있습니다.
- **클럭 속도:** 코어의 처리 속도가 빠를수록 개별 스레드가 작업을 빨리 완료할 수 있습니다.

- **하이퍼스레딩(Hyper-Threading):** 일부 CPU는 하나의 코어에서 두 개의 스레드를 실행할 수 있는 하이퍼스레딩 기능을 지원합니다. 이를 통해 병렬 처리 효율을 높일 수 있습니다.
- **캐시 메모리:** L1, L2, L3 캐시 메모리는 CPU와 메모리 간의 데이터 접근 속도를 높입니다. 병렬 작업에서 캐시 일관성 문제가 발생할 수 있으며, 데이터 로컬리티를 최대화하여 캐시 히트율을 높이는 것이 중요합니다.

2. 메모리 (RAM)

메모리는 데이터를 일시적으로 저장하여 CPU가 빠르게 접근할 수 있도록 지원합니다. 병렬 프로그램에서는 메모리 사용량과 접근 패턴이 성능에 큰 영향을 미칩니다.

- **메모리 대역폭:** 여러 스레드가 동시에 메모리에 접근하므로, 메모리 대역폭이 충분해야 메모리 병목현상을 방지할 수 있습니다. 대역폭이 낮으면 스레드가 메모리에 접근하는 데 많은 시간이 걸려 병렬 처리의 성능이 떨어질 수 있습니다.
- **메모리 용량:** 병렬 작업은 일반적으로 많은 데이터를 처리하므로, 메모리 용량이 충분해야 합니다. 메모리가 부족하면 디스크 스왑이 발생해 성능이 크게 저하됩니다.
- **NUMA 구조:** 다중 프로세서 시스템에서는 비균일 메모리 접근(NUMA) 구조가 사용될 수 있습니다. NUMA 환경에서는 각 프로세서가 독립적인 메모리 영역을 가지며, 병렬 프로세스가 "로컬 메모리"에 접근하는 것이 성능에 유리합니다. NUMA 최적화를 통해 데이터 접근 성능을 높일 수 있습니다.

3. 스토리지 (Disk I/O)

병렬 프로그램에서 많은 데이터를 읽고 쓰는 경우 스토리지 성능이 중요해집니다. 특히 대용량 데이터를 처리하는 병렬 프로그램은 빠른 데이터 입출력이 필요합니다.

- **디스크 종류:** HDD보다는 SSD가 I/O 성능이 뛰어나므로, 대규모 데이터에 대한 병렬 처리가 필요한 경우 SSD를 사용하는 것이 유리합니다.
- **I/O 대역폭:** 디스크의 읽기/쓰기 대역폭이 높아야 다수의 스레드가 동시에 데이터를 읽고 쓸 때 병목현상이 줄어듭니다.
- **RAID 및 병렬 파일 시스템:** RAID 구성을 통해 여러 디스크를 병렬로 사용하거나, 병렬 파일 시스템을 사용하여 입출력 속도를 높일 수 있습니다.

4. 네트워크

분산 환경에서 병렬 프로그램을 실행하는 경우 네트워크 성능이 큰 영향을 미칩니다. 여러 시스템 간 데이터를 주고받는 속도가 전체 성능을 좌우하기도 합니다.

- **네트워크 대역폭:** 네트워크를 통해 데이터를 전송할 때, 대역폭이 충분하지 않으면 데이터 전송이 병목 현상이 되어 성능이 저하됩니다. 특히 클러스터 환경에서 높은 네트워크 대역폭이 필요합니다.
- **네트워크 지연:** 네트워크 지연 시간이 길면 데이터 전송 속도가 느려지고, 분산 시스템의 응답 시간이 증가합니다. 지연 시간이 짧은 네트워크 인프라가 이상적입니다.

- **InfiniBand 및 고속 네트워크:** 대규모 병렬 처리가 필요한 클러스터에서는 고속 네트워크(예: InfiniBand)를 사용하여 네트워크 전송 속도를 높일 수 있습니다.

5. GPU (Graphics Processing Unit)

데이터 병렬 처리를 위한 추가 컴퓨팅 리소스로 GPU가 활용됩니다. GPU는 수천 개의 코어로 구성되어 있으며, 병렬 연산에 특화된 구조를 가지고 있어 대규모 데이터 처리와 연산에 유리합니다.

- **GPU 메모리:** GPU 메모리 용량이 충분히 커야 대규모 데이터를 효율적으로 처리할 수 있습니다.
- **데이터 전송 속도:** GPU와 CPU 간 데이터 전송이 병목이 되지 않도록 PCIe 대역폭을 고려해야 합니다.
- **CUDA 및 OpenCL:** 병렬 프로그램이 GPU의 성능을 최대한 활용하려면, CUDA(엔비디아 GPU용) 또는 OpenCL(다양한 GPU 지원)과 같은 병렬 프로그래밍 인터페이스를 사용해야 합니다.

6. OS 및 스케줄러

운영체제(OS)와 스케줄러도 병렬 프로그램의 성능에 중요한 역할을 합니다. OS는 스레드와 프로세스를 관리하고, CPU 자원을 효율적으로 분배하는 책임이 있습니다.

- **스레드 스케줄링:** 운영체제가 스레드를 어떻게 스케줄링하는지에 따라 성능이 크게 달라집니다. OS 스케줄러는 스레드 우선순위와 프로세서 선호도 등을 고려하여 스케줄링을 최적화할 수 있습니다.
- **가상 메모리 관리:** OS는 메모리 관리를 위해 가상 메모리를 사용합니다. 메모리가 부족할 때 디스크 스왑을 발생시키지 않도록 메모리 용량을 충분히 할당하는 것이 중요합니다.
- **파일 시스템:** 파일 시스템의 성능 또한 병렬 데이터 입출력에 영향을 미칩니다. 병렬 프로그램에서 데이터 입출력이 잦을 경우 빠른 입출력을 지원하는 파일 시스템을 사용하는 것이 좋습니다.

5. 총평

스레드 순서 제어와 동기화의 필요성

스레드가 동시에 같은 자원에 접근하거나 순서가 중요한 작업을 수행할 때, 순서를 지정하거나 동기화를 통해 실행 순서를 제어하는 것이 중요합니다. 동기화를 적용하지 않으면 스레드가 예상하지 못한 순서로 실행되어 데이터 불일치, 자원 충돌, 데이터 경합 등의 문제가 발생할 수 있습니다.

순서 제어 방법

스레드의 실행 순서를 제어하거나 작업 완료를 보장하기 위해 여러 방법을 사용할 수 있습니다:

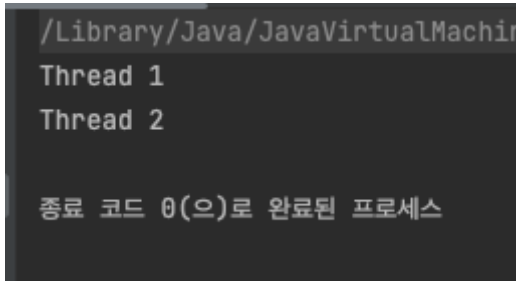
1. `join()` 메서드 사용:

- 특정 스레드가 완료될 때까지 다른 스레드가 대기하게 할 수 있습니다.

- 예를 들어, `thread1.join()` 을 호출하면 `thread1` 이 완료될 때까지 메인 스레드가 대기하게 됩니다.

```
val thread1 = Thread { println("Thread 1") }
val thread2 = Thread { println("Thread 2") }

thread1.start()
thread1.join() // thread1이 끝날 때까지 대기
thread2.start()
```



```
/Library/Java/JavaVirtualMachin
Thread 1
Thread 2

종료 코드 0(으)로 완료된 프로세스
```

2. synchronized 블록:

- `synchronized` 키워드를 사용하여 특정 코드 블록에 락(lock)을 걸어, 하나의 스레드만 해당 코드에 접근할 수 있도록 합니다. 이렇게 하면 스레드가 지정된 순서에 따라 자원에 접근할 수 있습니다.
- 예를 들어, 특정 객체에 대한 동기화를 보장하려면 `synchronized` 키워드를 사용하여 접근을 직렬화할 수 있습니다.

```
val lock = Any()

val thread1 = Thread {
    synchronized(lock) {
        println("Thread 1 - critical section")
    }
}

val thread2 = Thread {
    synchronized(lock) {
        println("Thread 2 - critical section")
    }
}
```

```
/Library/Java/JavaVirtualMachines
Thread 1 - critical section
Thread 2 - critical section

종료 코드 0(으)로 완료된 프로세스
```

3. Lock 객체 사용:

- Java Lock 클래스(ReentrantLock)를 사용하여 더 세밀하게 락을 제어할 수 있습니다. tryLock(), lockInterruptibly() 등의 다양한 락 옵션을 통해 동시성 제어를 세밀하게 할 수 있습니다.

4. wait() / notify() 메서드:

- 객체의 wait() 와 notify() 메서드를 사용하여, 스레드가 특정 조건을 만족할 때까지 대기하도록 하거나, 특정 조건을 만족한 스레드가 다른 스레드를 깨워 작업을 이어서 하도록 할 수 있습니다.

5. ExecutorService 사용:

- ExecutorService 는 스레드를 직접 관리하지 않고, 스레드 풀에서 스레드를 가져와 작업을 수행하게 해주는 관리 도구입니다. 작업을 큐에 넣어 순차적으로 실행하거나, 여러 스레드가 동시에 실행되지 않도록 제한할 수 있습니다.

```
val executor = Executors.newFixedThreadPool(2)
executor.submit { println("Task 1") }
executor.submit { println("Task 2") }
executor.shutdown()
```

6. CountdownLatch, CyclicBarrier:

- 여러 스레드가 일정한 지점에서 만나거나 대기할 때 유용합니다.
- 예를 들어 CountdownLatch 는 특정 작업이 완료될 때까지 다른 스레드가 기다리게 하며, CyclicBarrier 는 지정된 수의 스레드가 도달하면 함께 다음 단계로 넘어가게 합니다.

스레드 순서 제어의 필요성 예시

예를 들어, 은행 계좌의 잔액을 업데이트하는 작업을 병렬로 처리한다고 가정할 때, 잔액을 읽고 쓰는 과정이 여러 스레드에서 동시에 이루어지면 값이 예상치 못하게 변경될 수 있습니다. 이를 방지하려면 특정 스레드가 잔액을 수정하는 동안 다른 스레드는 대기하게 해야 합니다.

```
var balance = 100

val thread1 = Thread {
    synchronized(lock) {
```

```
        balance += 50
        println("Thread 1 updated balance: $balance")
    }
}

val thread2 = Thread {
    synchronized(lock) {
        balance -= 30
        println("Thread 2 updated balance: $balance")
    }
}
```

위처럼 동기화 없이 단순히 스레드를 시작하면, `balance` 값이 잘못될 수 있지만, `synchronized`를 사용하여 각 스레드가 작업을 완료할 때까지 다른 스레드가 기다리게 함으로써 올바르게 업데이트할 수 있습니다.