1. 프로세스와 스레드 소개

#1.인강/0.자바/5.자바-고급1편

멀티태스킹과 멀티프로세싱

멀티스레드에 대해서 제대로 이해하려면 먼저 멀티태스킹과 프로세스 같은 운영체제의 기념 개념들에 대해서 알아야한다. 여기서는 멀티스레드를 이해하기 위한 목적으로 최대한 단순하게 핵심 내용만 알아보겠다.

단일 프로그램 실행

만약 프로그램을 2개 이상 동시에 실행한다고 가정해보자. 예를 들어서 음악 프로그램을 통해 음악을 들으면서, 동시에 워드 프로그램을 통해 문서를 작성하는 것이다. 여기서는 연산을 처리할 수 있는 CPU 코어가 1개만 있다고 가정하겠다.

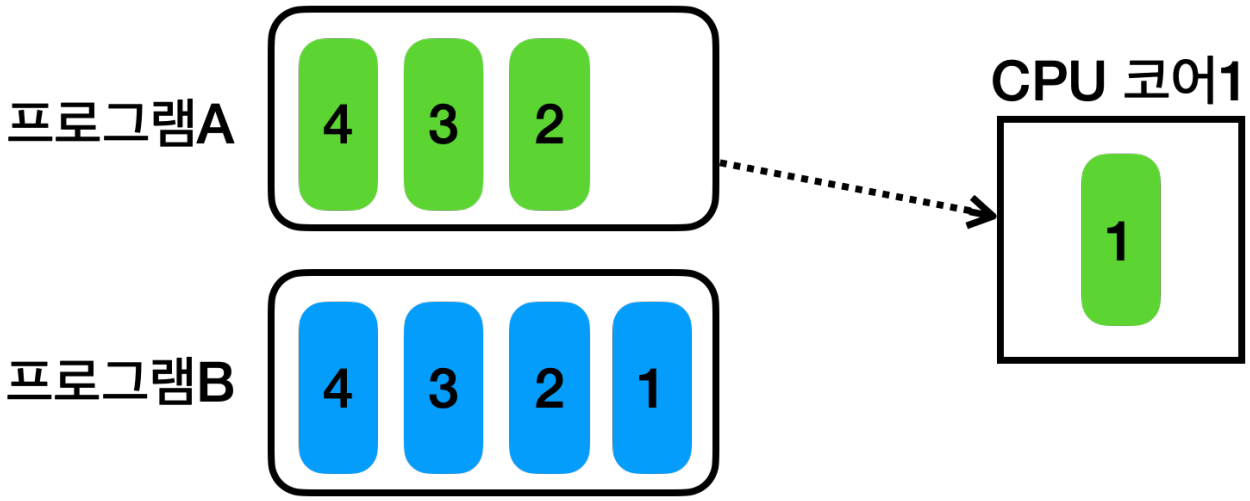


그림1 - 프로그램A의 코드1 실행 시작

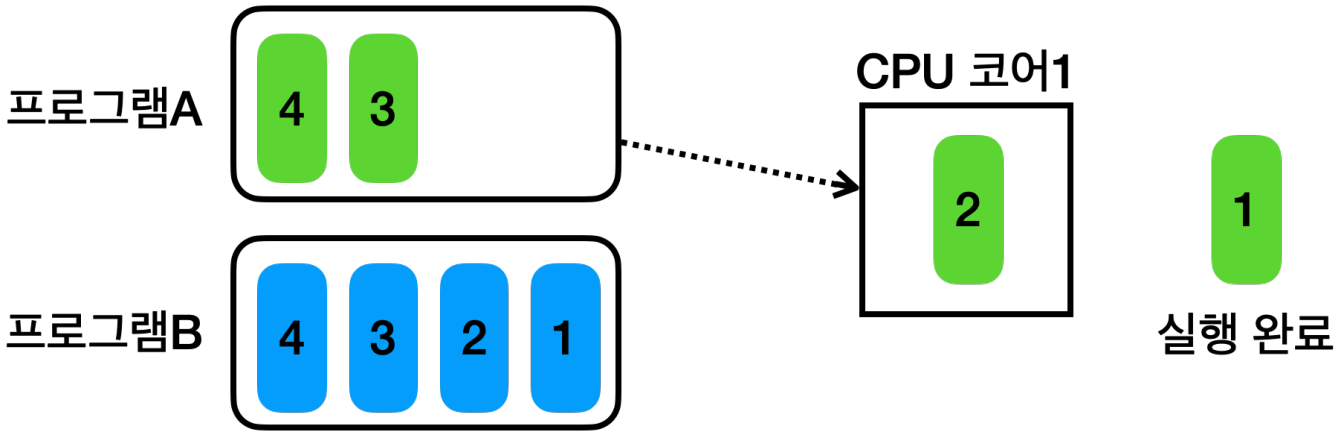


그림2 - 프로그램A의 코드2 실행 중

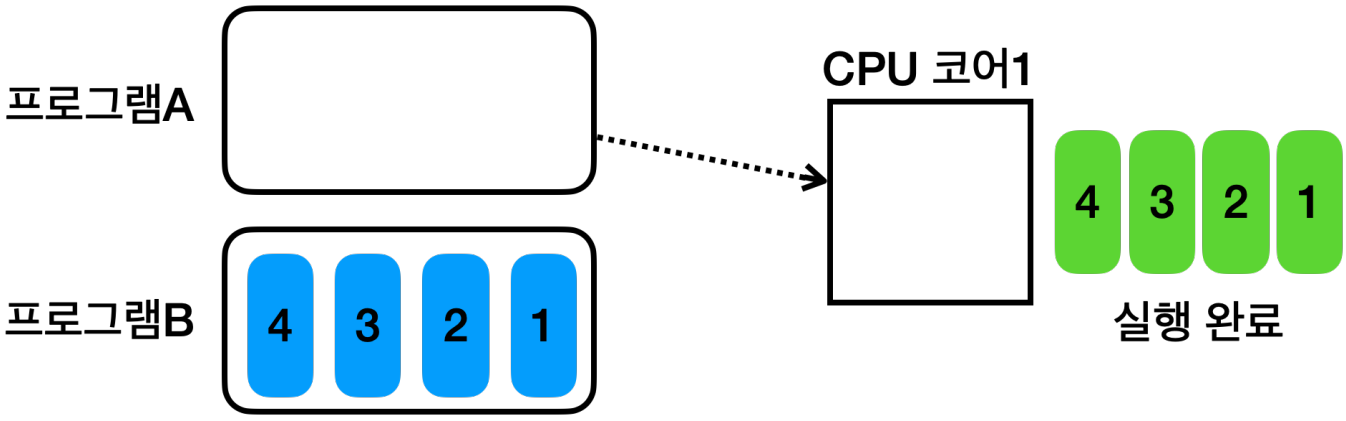


그림3 - 프로그램A의 실행 완료

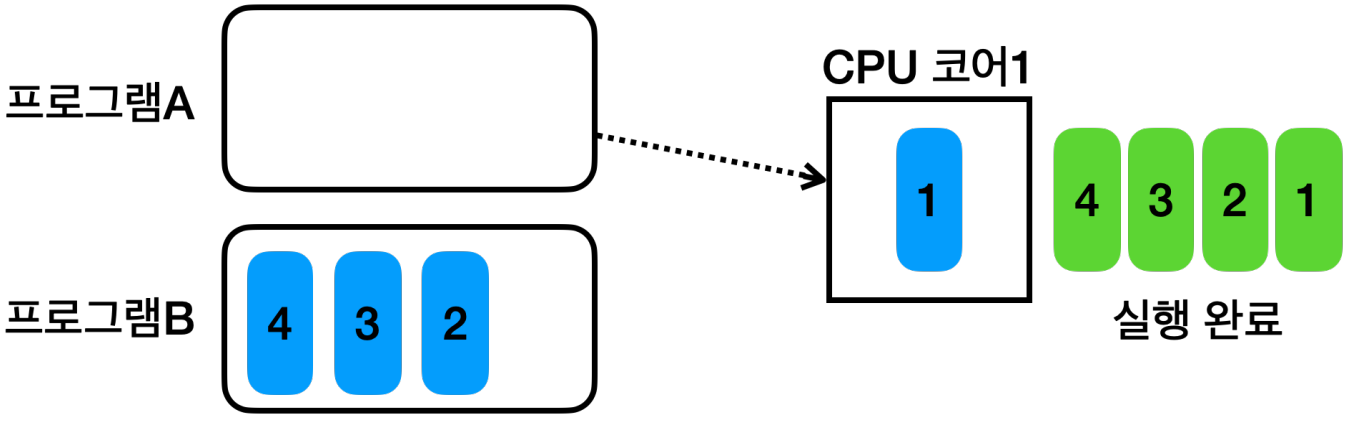


그림4 - 프로그램B 실행 시작

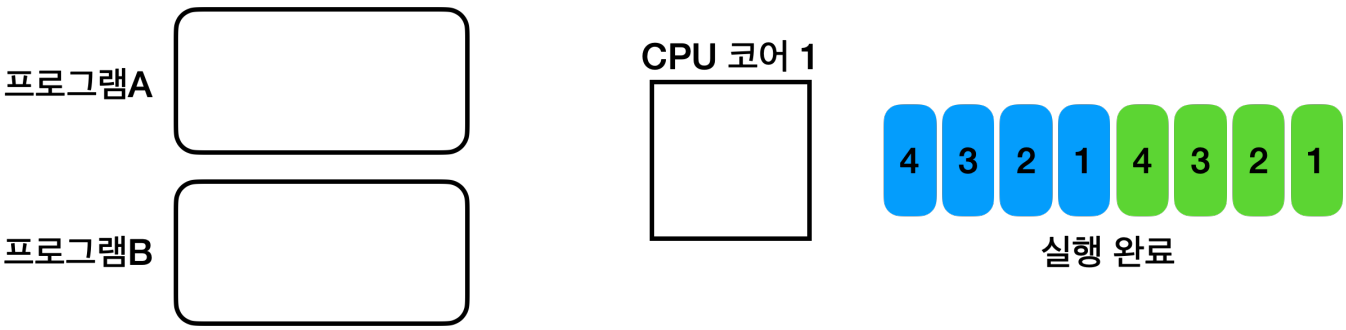


그림5 - 프로그램A 완료후 프로그램 B완료

- 프로그램의 실행이란 프로그램을 구성하는 코드를 순서대로 CPU에서 연산(실행)하는 일이다.
- 여기서 CPU 코어는 하나로 가정하므로, 한 번에 하나의 프로그램 코드만 실행할 수 있다.
- 이때, 하나의 프로그램 안에 있는 코드를 모두 후에야 다른 프로그램의 코드를 실행할 수 있다면? 예를 들어 음악 프로그램이 끝난 뒤에야 워드 프로그램을 실행할 수 있다면 컴퓨터 사용자는 매우 답답할 것이다.
- 실제로 초창기의 컴퓨터는 이 처럼 한 번에 하나의 프로그램만 실행했다.
- 이를 해결하기 위해 하나의 CPU 코어로 여러 프로그램을 동시에 실행하는 '멀티태스킹' 기술이 등장했다.

멀티태스킹

순서대로 촬영한 연속된 사진을 빠르게 교차해서 보여줄 경우 사람은 이를 움직이는 영상으로 인지한다. 애니메이션이 바로 이 원리를 이용한다. 예를 들어 우리가 애니메이션을 볼 때 1초에 30 ~ 60장의 사진이 지나간다. 이 정도 속도면 사람은 사진이 아니라 연속해서 움직이는 영상으로 인지한다. 현대의 CPU는 초당 수십억 번 이상의 연산을 수행한다. 쉽게 이야기해서 초당 수십억 장의 사진이 빠르게 교차되는 것이다.

만약 CPU가 매우 빠르게 두 프로그램의 코드를 번갈아 수행한다면, 사람이 느낄 때 두 프로그램이 동시에 실행되는 것처럼 느껴질 것이다. (대략 0.01초(10ms) 단위로 돌아가며 실행한다.)

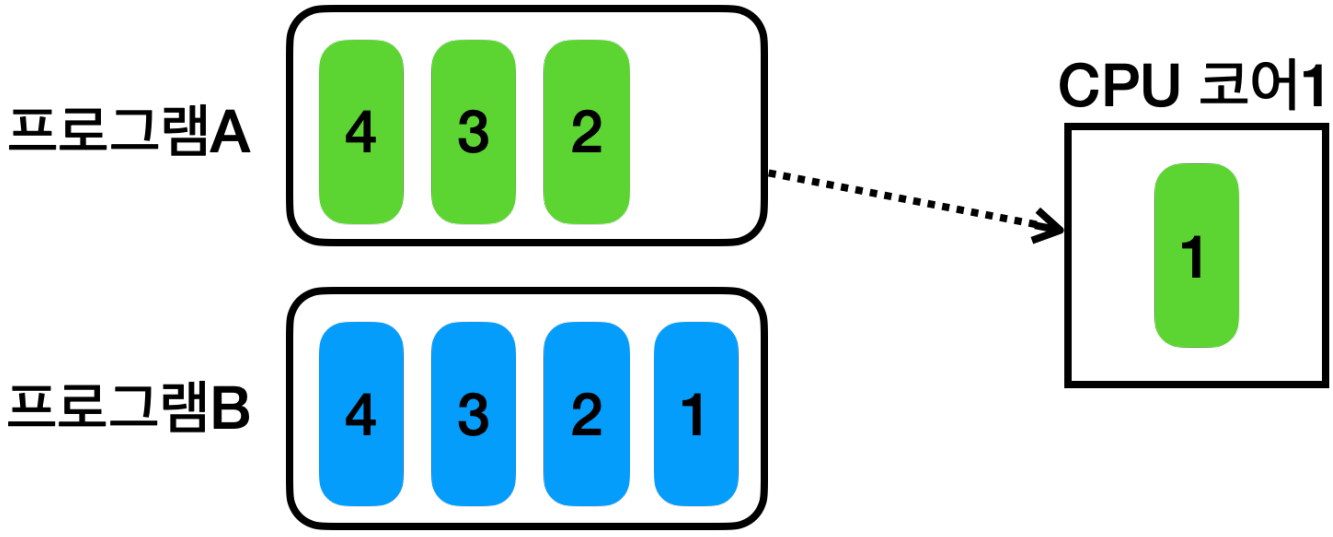


그림1 - 프로그램A의 코드1 수행

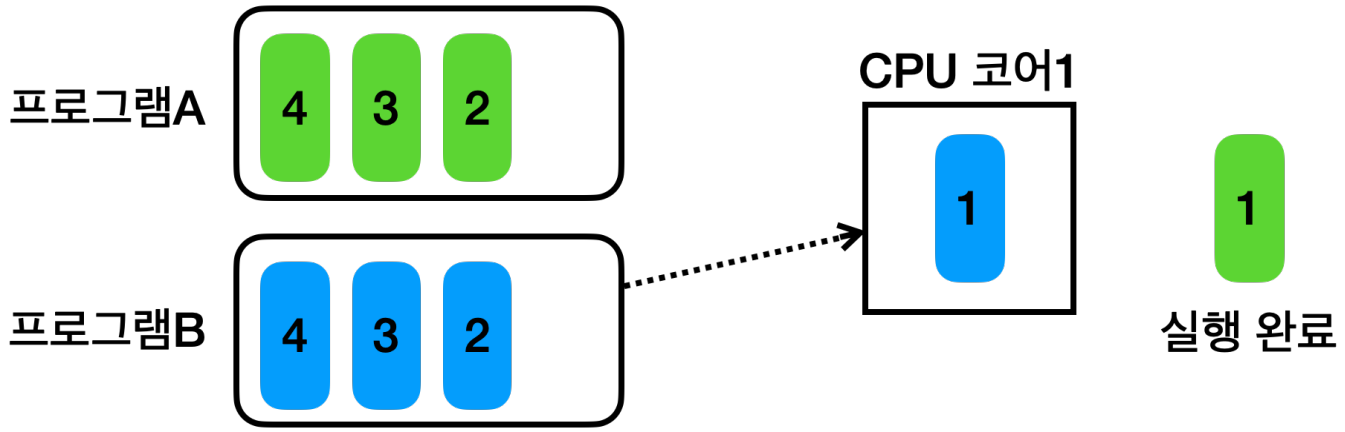


그림1 - 프로그램B의 코드1 수행

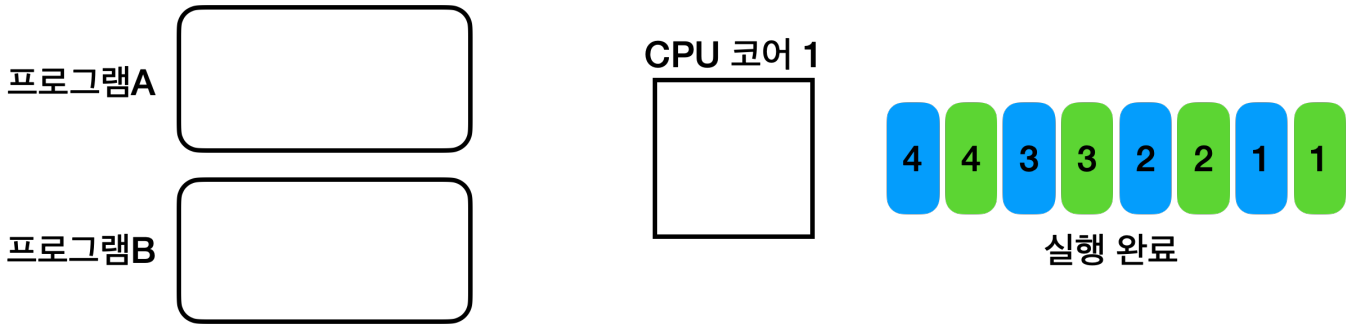


그림3 - 프로그램 수행 완료

이 방식은 CPU 코어가 프로그램A의 코드를 0.01초 정도 수행하다가 잠시 멈추고, 프로그램B의 코드를 0.01초 정도 수행한다. 그리고 다시 프로그램A의 이전에 실행중인 코드로 돌아가서 0.01초 정도 코드를 수행하는 방식으로 반복 동작한다.

이렇게 각 프로그램의 실행 시간을 분할해서 마치 동시에 실행되는 것 처럼 하는 기법을 시분할(Time Sharing, 시간 공유) 기법이라 한다. 이런 방식을 사용하면 CPU 코어가 하나만 있어도 여러 프로그램이 동시에 실행되는 것 처럼 느낄 수 있다.

이렇게 하나의 컴퓨터 시스템이 동시에 여러 작업을 수행하는 능력을 멀티태스킹(Multitasking)이라 한다.

> **참고**: CPU에 어떤 프로그램이 얼마만큼 실행될지는 운영체제가 결정하는데 이것을 스케줄링(Scheduling)이라 한다. 이때 단순히 시간으로만 작업을 분할하지는 않고, CPU를 최대한 활용할 수 있는 다양한 우선순위와 최적화 기법을 사용한다. 우리는 운영체제가 스케줄링을 수행하고, CPU를 최대한 사용하면서 작업이 골고루 수행될 수 있게 최적화한다는 정도로 이해하면 충분하다. 자세한 내용이 궁금한 분들은 운영체제 이론을 참고하자

멀티프로세싱

CPU 코어가 둘 이상이면 어떻게 될까?

여기서는 프로그램은 A, B, C 3가지이고, CPU 코어는 2개이다.

> **참고**: CPU 안에는 실제 연산을 처리할 수 있는 코어라는 것이 있다. 과거에는 하나의 CPU 안에 보통 하나의 코어만 들어있었다. 그래서 CPU와 코어를 따로 분리해서 이야기하지 않았다. 최근에는 하나의 CPU 안에 보통 2개 이상의 코어가 들어있다.

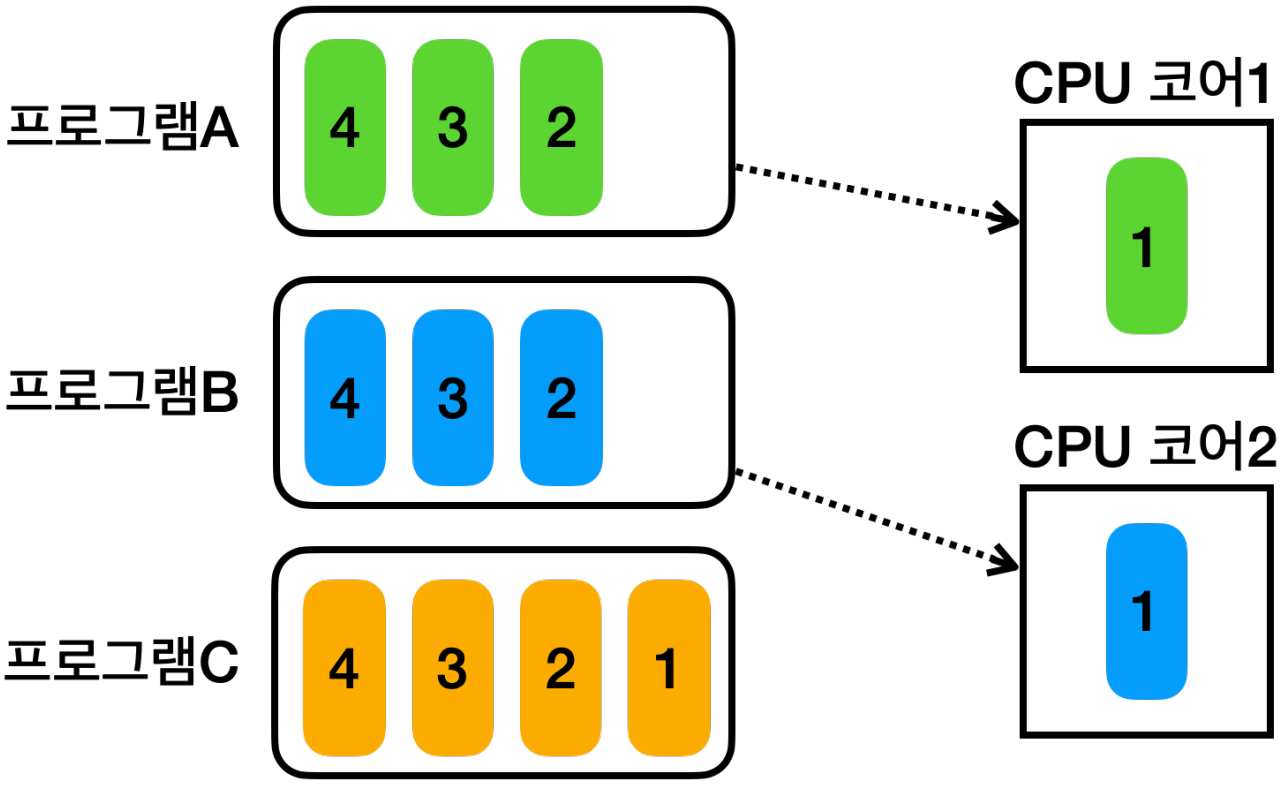


그림1 - 프로그램A, 프로그램B 실행

- CPU 코어가 2개이므로 물리적으로 동시에 2개의 프로그램을 처리할 수 있다.
- 여기서는 CPU 코어가 먼저 프로그램A와 프로그램B를 실행한다.

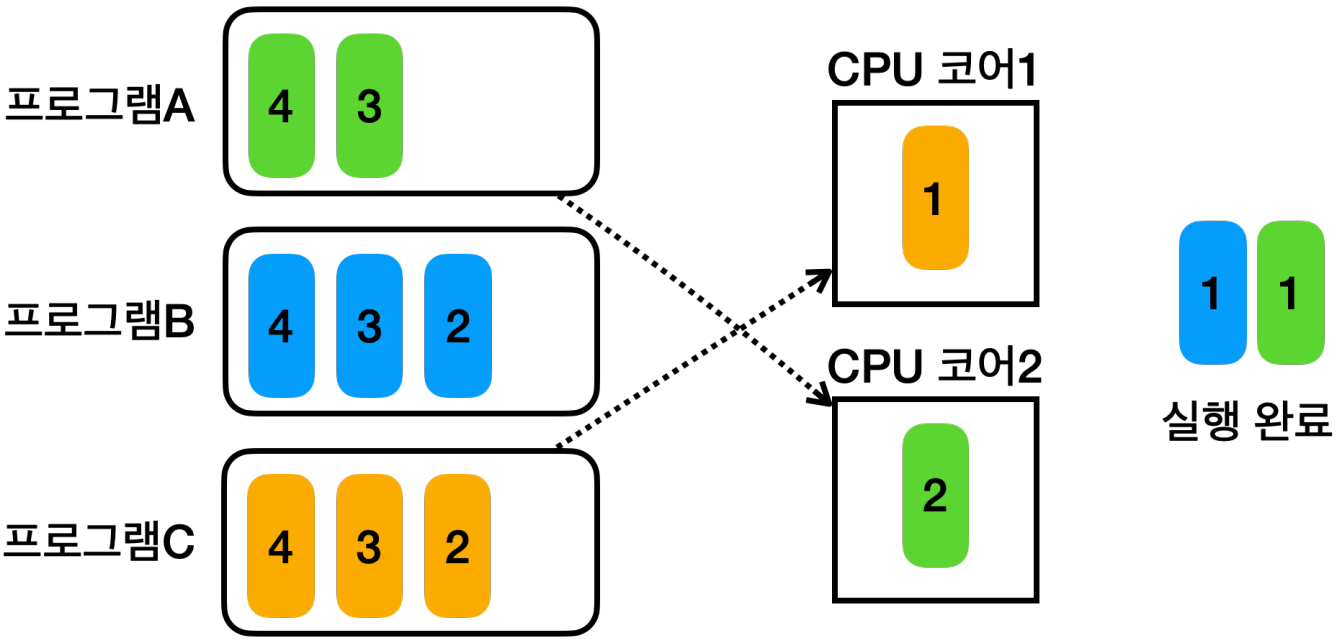


그림2 - 프로그램C, 프로그램A 실행

- CPU 코어들이 프로그램A와 프로그램B를 실행하다가 잠시 멈추고, 프로그램C와 프로그램A를 수행한다. 이런식

으로 코어가 2개여도 2개보다 더 많은 프로그램을 실행할 수 있다.

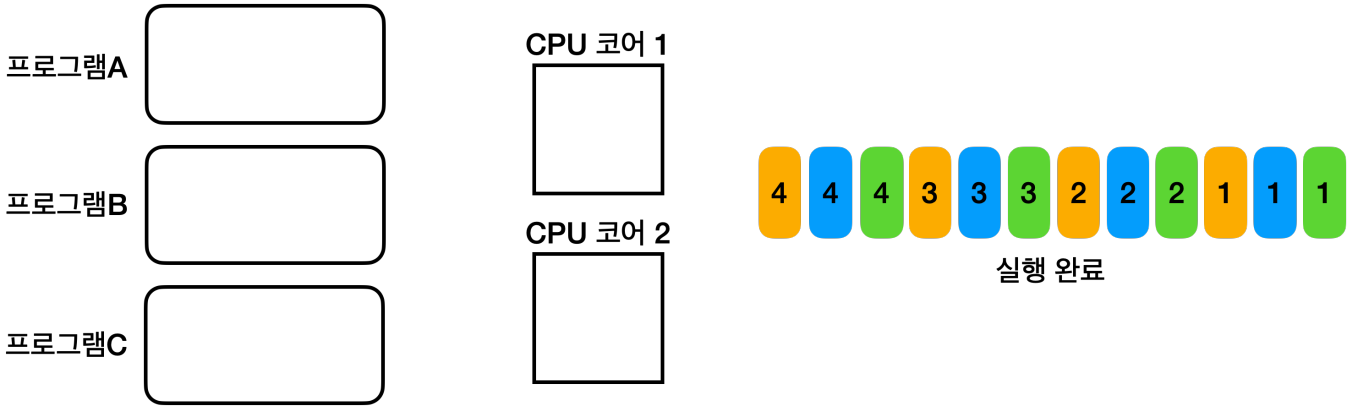


그림3 - 실행 완료

멀티프로세싱(Multiprocessing)은 컴퓨터 시스템에서 둘 이상의 프로세서(CPU 코어)를 사용하여 여러 작업을 동시에 처리하는 기술을 의미한다. 멀티프로세싱 시스템은 하나의 CPU 코어만을 사용하는 시스템보다 동시에 더 많은 작업을 처리할 수 있다.

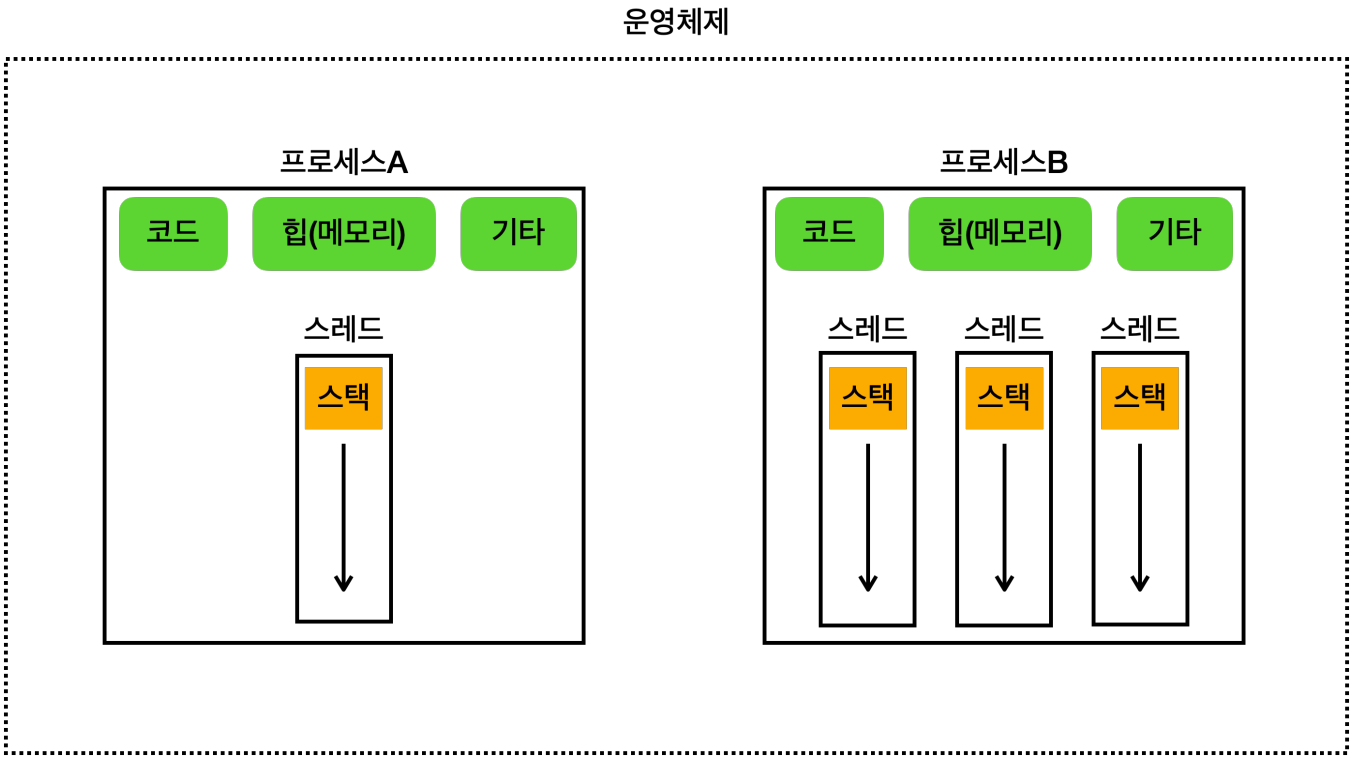
멀티프로세싱 vs. 멀티태스킹

멀티프로세싱은 하드웨어 장비의 관점이고, 멀티태스킹은 운영체제 소프트웨어의 관점이다.

- **멀티프로세싱**
  - 여러 CPU(여러 CPU 코어)를 사용하여 동시에 여러 작업을 수행하는 것을 의미한다.
  - 하드웨어 기반으로 성능을 향상시킨다.
  - 예: 다중 코어 프로세서를 사용하는 현대 컴퓨터 시스템
- **멀티태스킹**
  - 단일 CPU(단일 CPU 코어)가 여러 작업을 동시에 수행하는 것처럼 보이게 하는 것을 의미한다.
  - 소프트웨어 기반으로 CPU 시간을 분할하여 각 작업에 할당한다.
  - 예: 현대 운영 체제에서 여러 애플리케이션이 동시에 실행되는 환경

참고로 이 예는 여러 CPU 코어를 사용하기 때문에 멀티프로세싱이다. 동시에 각각의 단일 CPU 코어에 여러 작업을 분할해서 수행하기 때문에 멀티태스킹이다.

프로세스와 스레드



프로세스

- 프로그램은 실제 실행하기 전까지는 단순한 파일에 불과하다.
- 프로그램을 실행하면 프로세스가 만들어지고 프로그램이 실행된다.
- 이렇게 운영체제 안에서 **실행중인 프로그램을 프로세스**라 한다.
- 프로세스는 실행 중인 프로그램의 **인스턴스**이다.
- 자바 언어로 비유를 하자면 클래스는 프로그램이고, 인스턴스는 프로세스이다.

프로세스는 실행 중인 프로그램의 인스턴스이다. 각 프로세스는 독립적인 메모리 공간을 갖고 있으며, 운영체제에서 별도의 작업 단위로 분리해서 관리된다. 각 프로세스는 별도의 메모리 공간을 갖고 있기 때문에 서로 간섭하지 않는다. 그리고 프로세스가 서로의 메모리에 직접 접근할 수 없다. 프로세스는 이렇듯 서로 격리되어 관리되기 때문에, 하나의 프로세스가 충돌해도 다른 프로세스에는 영향을 미치지 않는다. 쉽게 이야기해서 특정 프로세스(프로그램)에 심각한 문제가 발생하면 해당 프로세스만 종료되고, 다른 프로세스에 영향을 주지 않는다.

프로세스의 메모리 구성

- **코드 섹션**: 실행할 프로그램의 코드가 저장되는 부분
- **데이터 섹션**: 전역 변수 및 정적 변수가 저장되는 부분(그림에서 기타에 포함)
- **힙 (Heap)**: 동적으로 할당되는 메모리 영역
- **스택 (Stack)**: 메서드(함수) 호출 시 생성되는 지역 변수와 반환 주소가 저장되는 영역(스레드에 포함)

스레드 (Thread)

프로세스는 하나 이상의 스레드를 반드시 포함한다.

스레드는 프로세스 내에서 실행되는 작업의 단위이다. 한 프로세스 내에서 여러 스레드가 존재할 수 있으며, 이들은 프로세스가 제공하는 동일한 메모리 공간을 공유한다. 스레드는 프로세스보다 단순하므로 생성 및 관리가 단순하고 가볍다.

메모리 구성

- **공유 메모리**: 같은 프로세스의 코드 섹션, 데이터 섹션, 힙(메모리)은 프로세스 안의 모든 스레드가 공유한다.
- **개별 스택**: 각 스레드는 자신의 스택을 갖고 있다.

프로그램이 실행된다는 것은 어떤 의미일까?

프로그램을 실행하면 운영체제는 먼저 디스크에 있는 파일 덩어리인 프로그램을 메모리로 불러오면서 프로세스를 만든다. 그럼 만들어진 프로세스를 어떻게 실행할까?

프로그램이 실행된다는 것은 사실 프로세스 안에 있는 코드가 한 줄씩 실행되는 것이다. 코드는 보통 `main()` 부터 시작해서 하나씩 순서대로 내려가면서 실행된다.

```
public class Operator {

    public static void main(String[] args) {
        int sum1 = 1;
        int sum2 = sum1 + 1;
        System.out.println("sum1 = " + sum1);
        System.out.println("sum2 = " + sum2);
    }

}
```

생각해보면 어떤 무언가가 코드를 하나씩 순서대로 실행하기 때문에 프로그램이 작동하고 계산도 하고, 출력도 할 수 있다.

이 코드를 하나씩 실행하면서 내려가는 것의 정체가 무엇일까?

비유를 하자면 마치 실(thread) 같은 것이 코드를 위에서 아래로 하나씩 꿰면서 하나씩 내려가는 것 같다.

이렇듯 프로세스의 코드를 실행하는 흐름을 스레드(thread)라 한다. 스레드는 번역하면 "실", "실을 꿰다"라는 뜻이다.

스레드는 프로세스 내에서 실행되는 작업의 단위이다. 한 프로세스 내에 하나의 스레드가 존재할 수 있고, 한 프로세스 내에 여러 스레드가 존재할 수도 있다. 그리고 스레드는 프로세스가 제공하는 동일한 메모리 공간을 공유한다.

- **단일 스레드**: 한 프로세스 내에 하나의 스레드만 존재
- **멀티 스레드**: 한 프로세스 내에 여러 스레드가 존재

하나의 프로세스 안에는 최소 하나의 스레드가 존재한다. 그래야 프로그램이 실행될 수 있다.
참고로 우리가 지금까지 작성한 자바 코드들은 모두 한 프로세스 내에서 하나의 스레드만 사용하는 단일 스레드였다.

정리하면 프로세스는 실행 환경과 자원을 제공하는 컨테이너 역할을 하고, 스레드는 CPU를 사용해서 코드를 하나하나 실행한다.

멀티스레드가 필요한 이유
하나의 프로그램도 그 안에서 동시에 여러 작업이 필요하다.

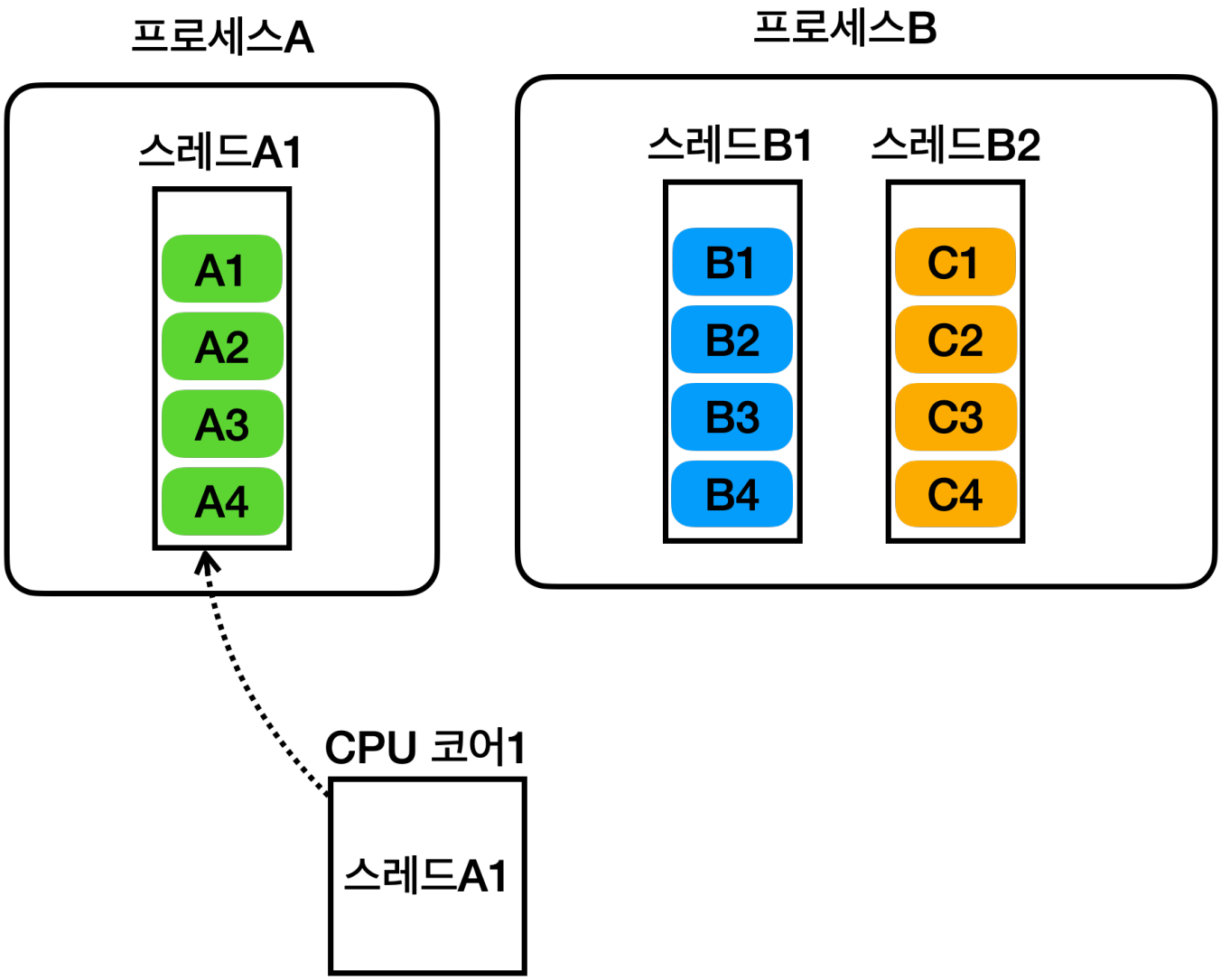
- 워드 프로그램으로 문서를 편집하면서, 문서가 자동으로 저장되고, 맞춤법 검사도 함께 수행된다.
- 유튜브는 영상을 보는 동안, 댓글도 달 수 있다.

운영제체 관점에서 보면 다음과 같이 구분할 수 있다.

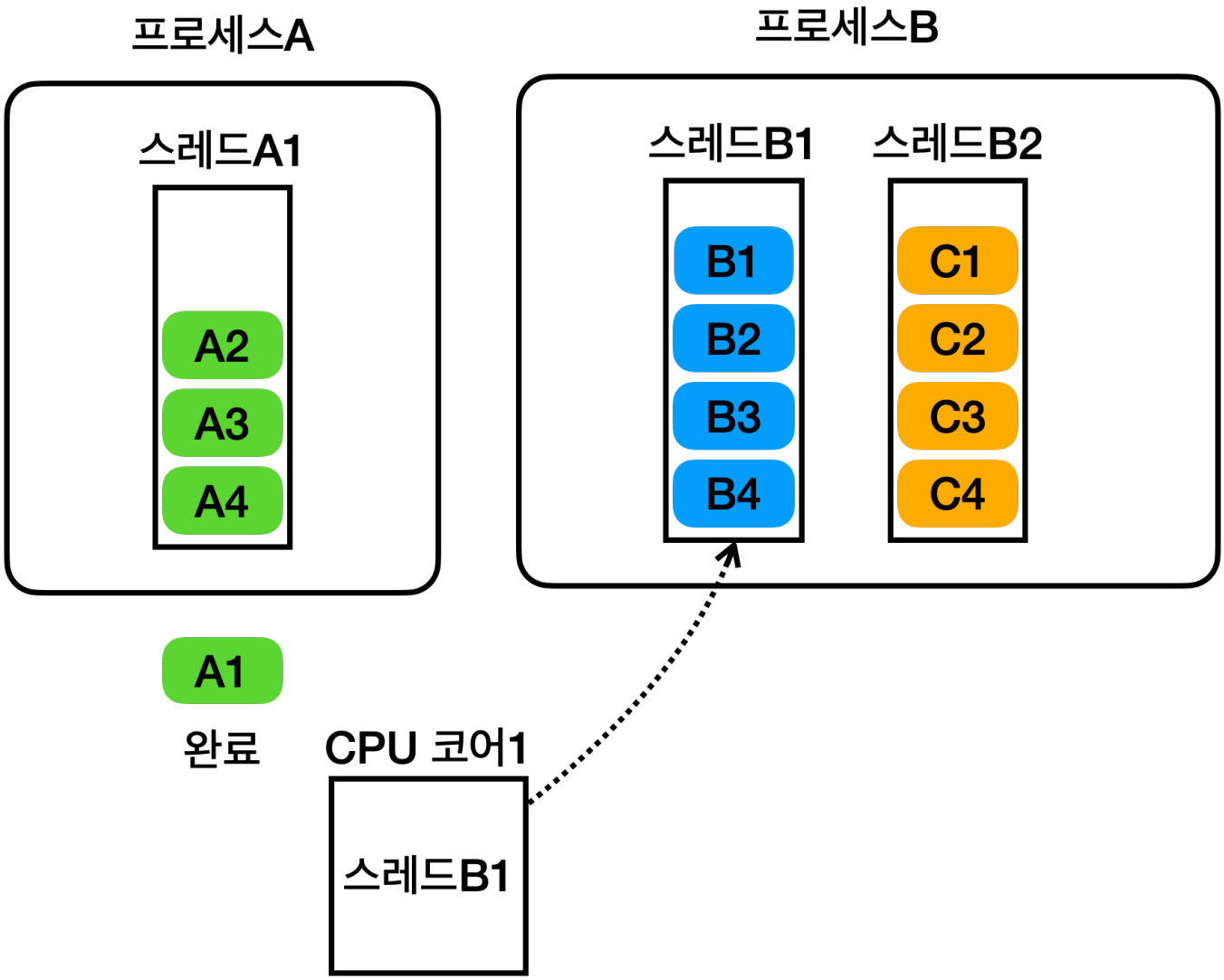
- **워드 프로그램 - 프로세스A**
  - 스레드1: 문서 편집
  - 스레드2: 자동 저장
  - 스레드3: 맞춤법 검사
- **유튜브 - 프로세스B**
  - 스레드1: 영상 재생
  - 스레드2: 댓글

스레드와 스케줄링

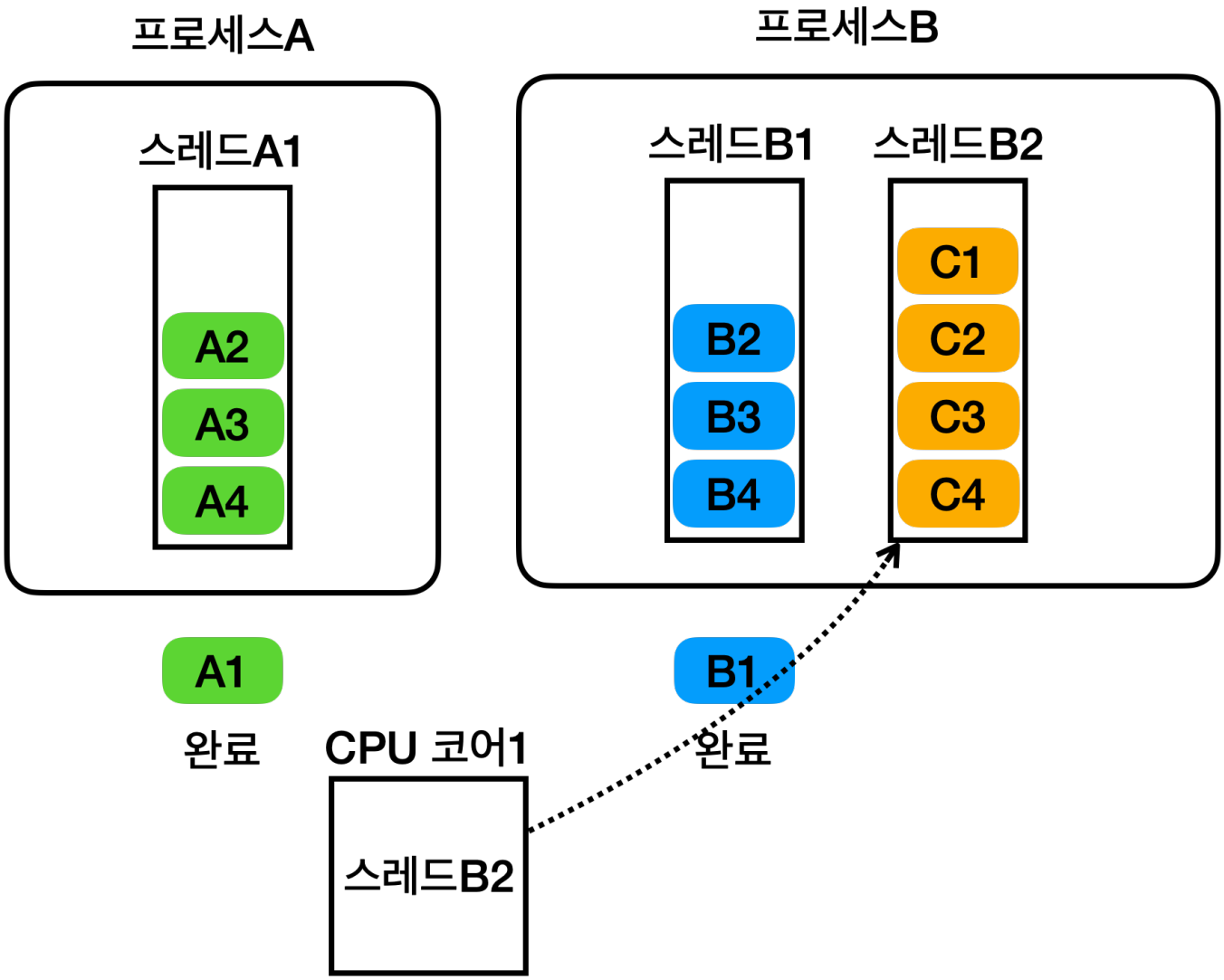
앞서 멀티태스킹에서 설명한 운영체제의 스케줄링을 과정을 더 자세히 알아보자.
CPU 코어는 1개이고, 프로세스는 2개이다. 프로세스 A는 스레드1개 프로세스B는 스레드가 2개 있다.
프로세스는 실행 환경과 자원을 제공하는 컨테이너 역할을 하고, 실제 CPU를 사용해서 코드를 하나하나 실행하는 것은 스레드이다.



- 프로세스A에 있는 스레드A1을 실행한다.



- 프로세스A에 있는 스레드A1의 실행을 잠시 멈추고 프로세스B에 있는 스레드 B1을 실행한다.



- 프로세스B에 있는 스레드 B1의 실행을 잠시 멈추고 같은 프로세스의 스레드 B2를 실행한다.
- 이후에 프로세스A에 있는 스레드A1을 실행한다.
- 이 과정을 반복한다.

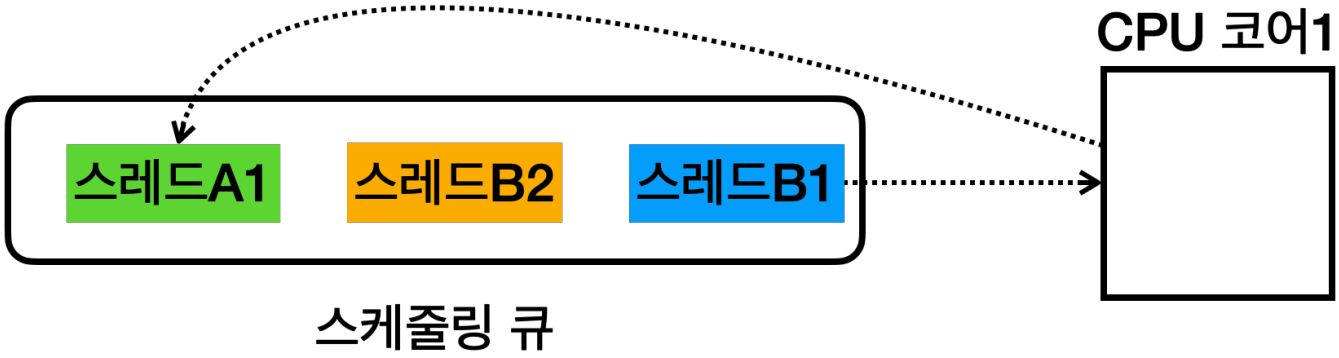
단일 코어 스케줄링

운영체체가 스레드를 어떻게 스케줄링 하는지, 스케줄링 관점으로 알아보자.

운영체제는 내부에 스케줄링 큐를 가지고 있고, 각각의 스레드는 스케줄링 큐에서 대기한다.

- 스레드A1, 스레드B1, 스레드B2가 스케줄링 큐에 대기한다.

- 운영체제는 스레드A1을 큐에서 꺼내고 CPU를 통해 실행한다.
- 이때 스레드A1이 프로그램의 코드를 수행하고, CPU를 통한 연산도 일어난다.



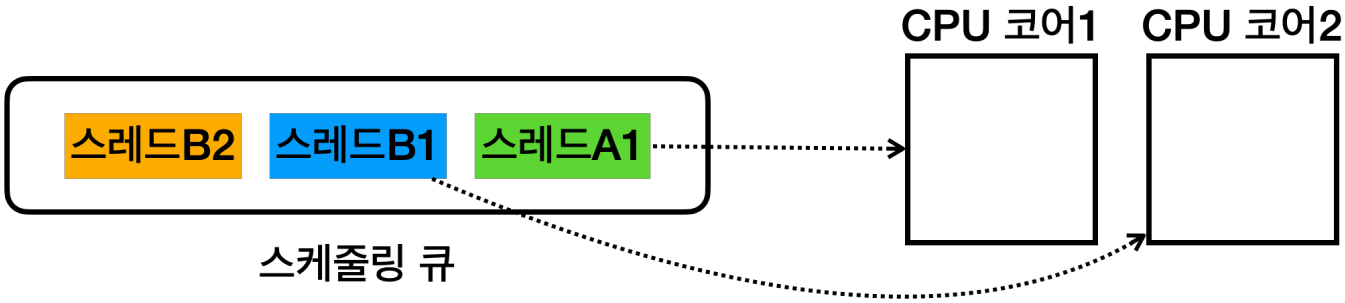
- 운영체제는 스레드A1을 잠시 멈추고, 스케줄링 큐에 다시 넣는다.

- 운영체제는 스레드B1을 큐에서 꺼내고 CPU를 통해 실행한다.

- 이런 과정을 반복해서 수행한다.

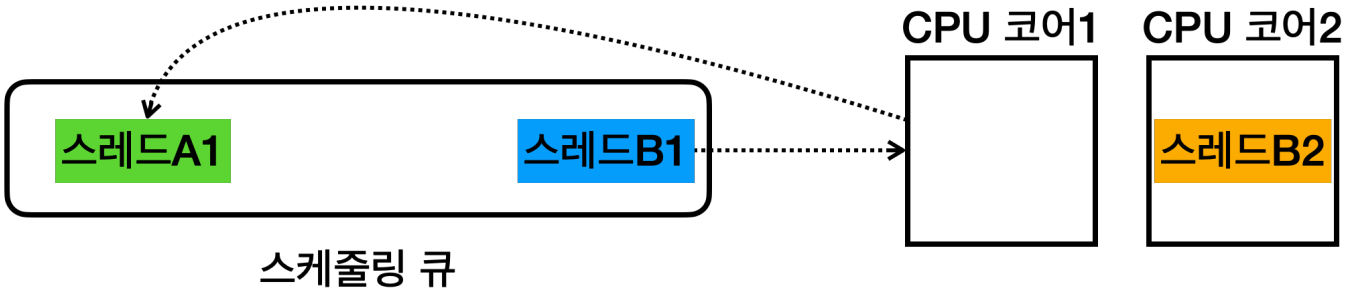
멀티 코어 스케줄링

CPU 코어가 2개 이상이면 한 번에 더 많은 스레드를 물리적으로 진짜 동시에 실행할 수 있다.



- 스레드A1, 스레드B1, 스레드B2가 스케줄링 큐에 대기한다.

- 스레드A1, 스레드B1을 병렬로 실행한다. 스레드B2는 스케줄링 큐에 대기한다.



- 스레드A1의 수행을 잠시 멈추고, 스레드A1을 스케줄링 큐에 다시 넣는다.

- 스케줄링 큐에 대기 중인 스레드B1을 CPU 코어1에서 실행한다.
- 물론 조금 있다가 CPU 코어2에서 실행중인 스레드B2도 수행을 멈추고, 스레드 스케줄링 큐에 있는 다른 스레드가 실행 될 것이다.

- 이런 과정을 반복해서 수행한다.

> **참고**: CPU에 어떤 프로그램이 얼마만큼 실행될지는 운영체제가 결정하는데 이것을 스케줄링(Scheduling)이라 한다. 이때 단순히 시간으로만 작업을 분할하지는 않고, CPU를 최대한 활용할 수 있는 다양한 우선순위와 최적화 기법을 사용한다. 우리는 운영체제가 스케줄링을 수행하고, CPU를 최대한 사용하면서 작업이 골고루 수행될 수 있게 최적화한다는 정도로 이해하면 충분하다. 자세한 내용이 궁금한 분들은 운영체제 이론을 참고하자

프로세스, 스레드와 스케줄링 - 정리

멀티태스킹과 스케줄링

- 멀티태스킹이란 동시에 여러 작업을 수행하는 것을 말한다. 이를 위해 운영체제는 스케줄링이라는 기법을 사용한다. 스케줄링은 CPU 시간을 여러 작업에 나누어 배분하는 방법이다.

프로세스와 스레드

- **프로세스**는 실행 중인 프로그램의 인스턴스이다. 각 프로세스는 독립적인 메모리 공간을 가지며, 운영체제에서 독립된 실행 단위로 취급된다.
- **스레드**는 프로세스 내에서 실행되는 작은 단위이다. 여러 스레드는 하나의 프로세스 내에서 자원을 공유하며, 프로세스의 코드, 데이터, 시스템 자원등을 공유한다. 실제로 CPU에 의해 실행되는 단위는 스레드이다.

프로세스의 역할

- 프로세스는 실행 환경을 제공한다. 여기에는 메모리 공간, 파일 핸들, 시스템 자원(네트워크 연결) 등이 포함된다. 이는 프로세스가 컨테이너 역할을 한다는 의미이다.
- 프로세스 자체는 운영체제의 스케줄러에 의해 직접 실행되지 않으며, 프로세스 내의 스레드가 실행된다. 참고로 1개의 프로세스 안에 하나의 스레드만 실행되는 경우도 있고, 1개의 프로세스 안에 여러 스레드가 실행되는 경우도 있다.

컨텍스트 스위칭

멀티태스킹이 반드시 효율적인 것 만은 아니다.

사람의 멀티태스킹

비유를 하자면 내가 프로그램A를 개발하고 있는데, 갑자기 기획자가 프로그램B를 수정해달라고 한다. 프로그램A의 개발을 멈추고, 프로그램B를 수정한다고 가정해보자. 여기서 프로그램B의 수정을 잘 마치고, 다시 프로그램A를 개발하기

위해 돌아간다.
이때 먼저 프로그램A의 어디를 개발하고 있었는지 해당 코드의 위치를 찾아야 한다. 그리고 개발할 때 변수들을 많이 선언하는데, 변수들에 어떤 값들이 들어가는지 머리속에 다시 불러와야 한다.
만약 프로그램A의 개발이 다 끝나고 나서, 프로그램B를 수정한다면, 전체 시간으로 보면 더 효율적으로 개발할 수 있을 것이다.

컴퓨터의 멀티태스킹
운영체제의 멀티태스킹을 생각해보자. CPU 코어는 하나만 있다고 가정하자.
스레드A, 스레드B가 있다.
운영체제는 먼저 스레드A를 실행한다. 멀티태스킹을 해야 하기 때문에 스레드A를 계속 실행할 수 없다. 스레드A를 잠시 멈추고, 스레드B를 실행한다. 이후에 스레드A로 그냥 돌아갈 수 없다. CPU에서 스레드를 실행하는데, 스레드A의 코드가 어디까지 수행되었는지 위치를 찾아야 한다. 그리고 계산하던 변수들의 값을 CPU에 다시 불러들여야 한다. 따라서 **스레드A를 멈추는 시점에 CPU에서 사용하던 이런 값들을 메모리에 저장해두어야 한다. 그리고 이후에 스레드A를 다시 실행할 때 이 값들을 CPU에 다시 불러와야 한다.**

이런 과정을 **컨텍스트 스위칭(context switching)**이라 한다.
컨텍스트는 현재 작업하는 문맥을 뜻한다. 현재 작업하는 문맥이 변하기 때문에 컨텍스트 스위칭이다.
컨텍스트 스위칭 과정에서 이전에 실행 중인 값을 메모리에 잠깐 저장하고, 이후에 다시 실행하는 시점에 저장한 값을 CPU에 다시 불러와야 한다.
결과적으로 컨텍스트 스위칭 과정에는 약간의 비용이 발생한다.

멀티스레드는 대부분 효율적이지만, 컨텍스트 스위칭 과정이 필요하므로 항상 효율적인 것은 아니다
예를 들어서 1 ~ 10000까지 더해야 한다고 가정해보자. 이 문제는 둘로 나눌 수 있다.

- 스레드1: 1 ~ 5000까지 더함
- 스레드2: 5001 ~ 10000까지 더함
- 마지막에 스레드1의 결과와 스레드2의 결과를 더함

CPU 코어가 2개
CPU 코어가 2개 있다면 스레드1, 스레드2로 나누어 멀티스레드로 병렬 처리하는게 효율적이다. 모든 CPU를 사용하므로 연산을 2배 빠르게 처리할 수 있다.

CPU 코어가 1개
CPU 코어가 1개 있는데, 스레드를 2개로 만들어서 연산하면 중간중간 컨텍스트 스위칭 비용이 발생한다. 운영체제 스케줄링 방식에 따라서 다르겠지만, 스레드1을 1 ~ 1000 정도까지 연산한 상태에서 잠시 멈추고 스레드2를 5001 ~ 6001까지 연산하는 식으로 반복할 수 있다. 이때 CPU는 스레드1을 멈추고 다시 실행할 때 어디까지 연산했는지 알아야 하고, 그 값을 CPU에 다시 불러와야 한다. 결과적으로 이렇게 반복할 때 마다 컨텍스트 스위칭 비용(시간)이든다.
결과적으로 연산 시간 + 컨텍스트 스위칭 시간이 든다.

이런 경우 단일 스레드로 1 ~ 10000까지 더하는 것이 컨텍스트 스위칭 비용 없이, 연산 시간만 사용하기 때문에 더 효율적이다.

예를 이렇게 들었지만 실제로 컨텍스트 스위칭에 걸리는 시간은 아주 짧다. 하지만 스레드가 매우 많다면 이 비용이 커질 수 있다.

물론 최신 CPU는 초당 수 십억 단위를 계산하기 때문에 실제로는 계산에 더 큰 숫자를 사용해야 컨텍스트 스위칭이 발생한다.

> 참고로 다른 이야기이지만 집중력 관점에서 사람의 컨텍스트 스위칭 비용은 매우 크기 때문에 보통 하나의 업무를 끝내고 다음 업무로 넘어가는 것이 좋다.

참고 - 실무 이야기

이 부분은 지금 다 이해하지 않아도 괜찮다. 편안하게 듣고 넘어가면 충분하다.

CPU - 4개, 스레드 2개

스레드의 숫자가 너무 적으면 모든 CPU를 100% 다 활용할 수 없지만, 스레드가 몇 개 없으므로 컨텍스트 스위칭 비용이 줄어든다.

CPU - 4개, 스레드 100개

스레드의 숫자가 너무 많으면 CPU를 100% 다 활용할 수 있지만 컨텍스트 스위칭 비용이 늘어난다.

CPU - 4개, 스레드 4개

스레드의 숫자를 CPU의 숫자에 맞춘다면 CPU를 100% 활용할 수 있고, 컨텍스트 스위칭 비용도 자주 발생하지 않기 때문에 최적의 상태가 된다. 이상적으로는 CPU 코어 수 + 1개 정도로 스레드를 맞추면 특정 스레드가 잠시 대기할 때 남은 스레드를 활용할 수 있다.

CPU 바운드 작업 vs I/O 바운드 작업

각각의 스레드가 하는 작업은 크게 2가지로 구분할 수 있다.

- **CPU-바운드 작업 (CPU-bound tasks)**
  - CPU의 연산 능력을 많이 요구하는 작업을 의미한다.
  - 이러한 작업은 주로 계산, 데이터 처리, 알고리즘 실행 등 CPU의 처리 속도가 작업 완료 시간을 결정하는 경우다.
  - 예시: 복잡한 수학 연산, 데이터 분석, 비디오 인코딩, 과학적 시뮬레이션 등

- **I/O-바운드 작업 (I/O-bound tasks)**
  - 디스크, 네트워크, 파일 시스템 등과 같은 입출력(I/O) 작업을 많이 요구하는 작업을 의미한다.
  - 이러한 작업은 I/O 작업이 완료될 때까지 대기 시간이 많이 발생하며, CPU는 상대적으로 유휴(대기) 상태에 있는 경우가 많다. 쉽게 이야기해서 스레드가 CPU를 사용하지 않고 I/O 작업이 완료될 때 까지 대기한다.
  - 예시: 데이터베이스 쿼리 처리, 파일 읽기/쓰기, 네트워크 통신, 사용자 입력 처리 등.

웹 애플리케이션 서버

분야마다 다르겠지만, 실무에서는 CPU-바운드 작업 보다는 I/O-바운드 작업이 많다.
예를 들어서 백엔드 개발자의 경우 주로 웹 애플리케이션 서버를 개발하는데, 스레드가 1 ~ 10000까지 더하는 CPU의 연산이 필요한 작업보다는, 대부분 사용자의 입력을 기다리거나, 데이터베이스를 호출하고 그 결과를 기다리는 등, 기다리는 일이 많다. 쉽게 이야기해서 스레드가 CPU를 많이 사용하지 않는 I/O-바운드 작업이 많다는 뜻이다.

일반적인 자바 웹 애플리케이션 서버의 경우 사용자의 요청 하나를 처리하는데 1개의 스레드가 필요하다. 사용자 4명이 동시에 요청하면 4개의 스레드가 작동하는 것이다. 그래야 4명의 사용자의 요청을 동시에 처리할 수 있다.
사용자의 요청을 하나 처리하는데, 스레드는 CPU를 1% 정도 사용하고, 대부분 데이터베이스 서버에 어떤 결과를 조회하면서 기다린다고 가정하자. 이때는 스레드는 CPU를 거의 사용하지 않고 대기한다. 바로 I/O-바운드 작업이 많다는 것이다.
이 경우 CPU 코어가 4개 있다고해서 스레드 숫자도 CPU 코어에 맞추어 4개로 설정하면 안된다! 그러면 동시에 4명의 사용자 요청만 처리할 수 있다. 이때 CPU는 단순하게 계산해서 4% 정도만 사용할 것이다. 결국 사용자는 동시에 4명 밖에 못받지만 CPU는 4%만 사용하며 CPU가 놀고 있는 사태가 벌어질 수 있다.

사용자의 요청 하나를 처리하는데 CPU를 1%만 사용한다면 단순하게 생각해도 100개의 스레드를 만들 수 있다. 이렇게 하면 동시에 100명의 사용자 요청을 받을 수 있다. 물론 실무에서는 성능 테스트를 통해서 최적의 스레드 숫자를 찾는 것이 이상적이다.

결국 스레드 숫자만 늘리면 되는데, 이런 부분을 잘 이해하지 못해서 서버 장비에 문제가 있다고 생각하고 2배 더 좋은 장비로 구매하는 사태가 발생하기도 한다! 이렇게 되면 CPU는 4%의 절반인 2%만 사용하고 사용자는 여전히 동시에 4명 밖에 받지 못하는 사태가 벌어진다.

정리하면 스레드의 숫자는 CPU-바운드 작업이 많은가, 아니면 I/O-바운드 작업이 많은가에 따라 다르게 설정해야 한다.

- **CPU-바운드 작업**: CPU 코어 수 + 1개
  - CPU를 거의 100% 사용하는 작업이므로 스레드를 CPU 숫자에 최적화
- **I/O-바운드 작업**: CPU 코어 수 보다 많은 스레드를 생성, CPU를 최대한 사용할 수 있는 숫자까지 스레드 생성
  - CPU를 많이 사용하지 않으므로 성능 테스트를 통해 CPU를 최대한 활용하는 숫자까지 스레드 생성
  - 단 너무 많은 스레드를 생성하면 컨텍스트 스위칭 비용도 함께 증가 - 적절한 성능 테스트 필요

참고로 웹 애플리케이션 서버라도 상황에 따라 CPU 바운드 작업이 많을 수 있다. 이 경우 CPU-바운드 작업에 최적화된 CPU 숫자를 고려하면 된다.