[디지털 컨버전스]
스마트 콘텐츠와 웹 융합 응용SW 개발자 양성과정

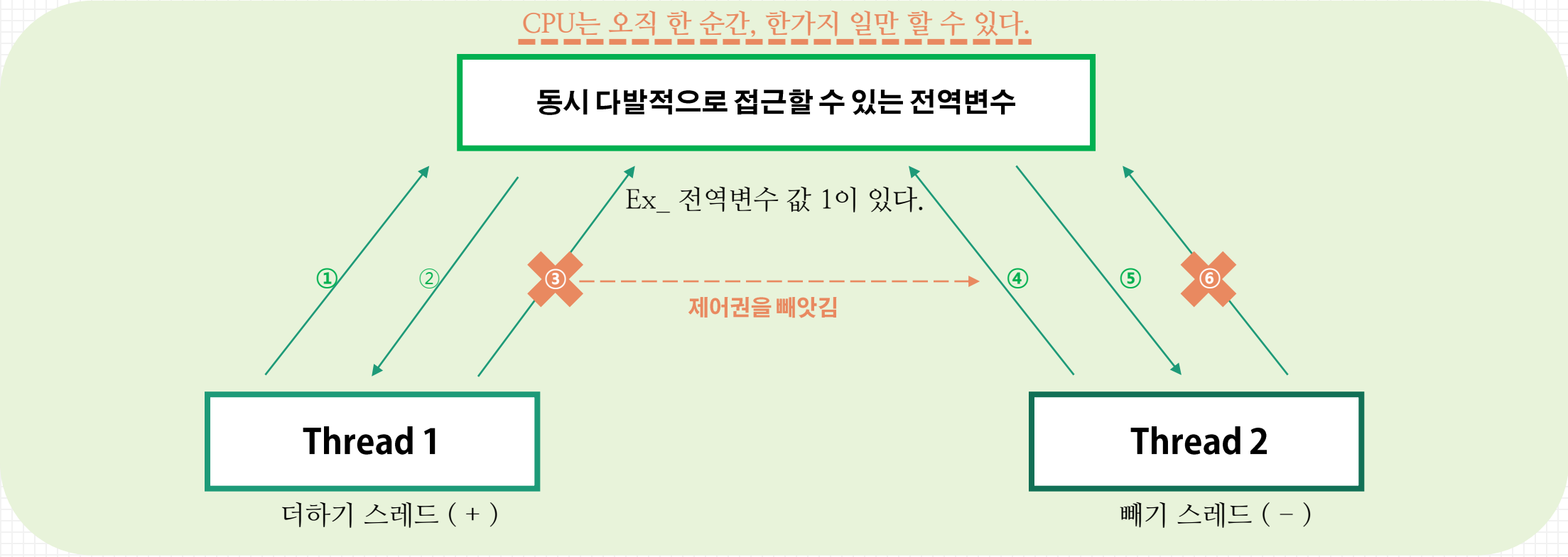
강사 : 이상훈
학생 : 임초롱

19회차

Thread

링크 https://github.com/limcholong/LectureContents/blob/main/java/CholongLim/Day19/src/DummyMemo.java

- 컴퓨터의 발전 : CISC ➔ RISC ➔ Multi-Core ➔ Heterogenous Architecture
  그러나 Multi-Core의 이슈가 있었다. 그 이슈는 Thread를 통한 작업에서 발생되었다.



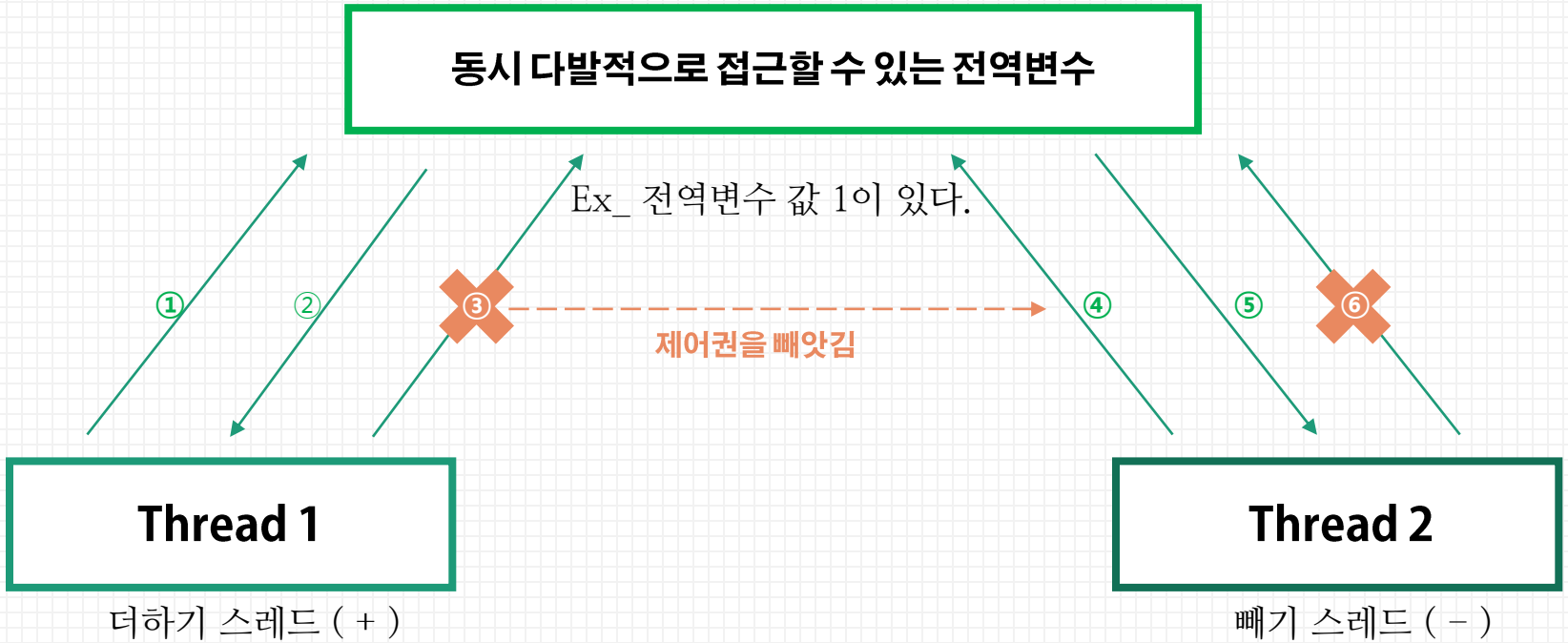
상황. Thread1에서 전역변수의 1을 받아와 덧셈 했다. Thread1 값이 2가 되었고 또 전역변수의 1을 받으려 한다. 이때 제어권을 Thread2에 빼앗겼다.
Thread2에서 전역변수의 1을 받아와 뺄셈 했다. Thread2 값이 0이 되었고 또 전역변수의 1을 받으려 한다. 이때 제어권을 Thread1에 빼앗겼다.

➔ 계산 값은 얼마인가?

Thread

링크 https://github.com/limcholong/LectureContents/blob/main/java/CholongLim/Day19/src/DummyMemo.java

CPU는 오직 한 순간, 한가지 일만 할 수 있다.



위 경우 두가지 문제가 있다.

1. Thread에서 제일 중요한 Synchronization(동기화) 문제

2. 어떤 특정 상황에서 데이터 처리가 무시되었고, **데이터의 무결성**이 지켜지지 않았다.

Thread

링크 https://github.com/limcholong/LectureContents/blob/main/java/CholongLim/Day19/src/DummyMemo.java

1. Thread에서 제일 중요한 **Synchronization(동기화)** 문제
: 데이터 영역에서 할당된 변수를 둘 이상의 Thread 동시에 접근할 때 문제가 발생한다. (계산결과가 덮어써진다.)
이렇게 메모리 접근에 있어서 동시 접근을 막는 것이 Thread의 동기화에 해당한다.

2. **어떤 특정 상황에서 데이터 처리가 무시되었고, 데이터의 무결성이 지켜지지 않았다.**
: 데이터의 무결성이란 데이터의 정확성과 일관성을 유지하고 보증하는 것이다.
1 + 1 = 2가 그 누구에게나 일관적인 값인 것처럼 어떠한 상황에서도 동작해야 하는 기능이 제 역할을 유지되어야 한다.

SHIELD

동시 다발적으로 접근할 수 있는 전역변수

동기화 문제를 막기 위해 전역변수를 shield로 보호하여 문제를 해결
여기서 말하는 shield는 Mutex, Semaphore, Spinlock 등을 말한다.
Shield 친 영역 = Critical Section (**임계 영역**)

Critical Section (임계 영역) 이란 여러 개의 Thread가 동시다발적으로 접근할 수 있는 데이터를 말한다.

Thread

링크 https://github.com/limcholong/LectureContents/blob/main/java/CholongLim/Day19/src/DummyMemo.java

Critical Section (임계 영역) : 여러 개의 Thread가 동시다발적으로 접근할 수 있는 데이터를 말한다.

동시다발적인 접근이란 ?
1) Multi - Tasking의 원리
2) Contect - Switching
3) 실제 컴퓨터가 연산하는 방식

1) Multi - Tasking의 원리

스타크래프트 선수들은 보면 화면 전환이 빠르고 유닛 컨트롤을 하면서 생산 / 일꾼 찍기 / 견제를 한다.
하지만 이것들은 아주 빠르게 순차적으로 일어날 뿐 실제로 동시에 하고있는 것이 아니다.

컴퓨터의 Multi - Tasking 개념도 위 개념과 동일하게,
빠르게 순차적으로 동작하지만 그 속도가 매우 빨라 우리가 느끼기엔 동시에 발생하는 것처럼 느껴진다.

< 의문 >

CPU는 오직 한 순간, 한가지 일만 할 수 있다.

Thread

링크 https://github.com/limcholong/LectureContents/blob/main/java/CholongLim/Day19/src/DummyMemo.java

CPU는 오직 한 순간, 한가지 일만 할 수 있다.

현재 우리 컴퓨터의 예를 들어보자.

1. 인터넷 창이 실행 중이다.
2. 프로그래밍을 위해 인텔리제이가 실행 중이다.
3. 줌 화상회의 프로그램이 실행 중이다.
4. 문서 프로그램이 실행 중이다.
5. 카카오톡이 실행 중이다
6. 슬랙 앱이 실행중이다

…

CPU가 한 순간에 한 가지 일만 할 수 있다는 것에 대해 이 경우 어떻게 설명해야 할까 ?

- CPU의 동작 주파수를 확인해보자 !

: 2.20 GHz가 나오고 있다.

(Hz, 1초에 몇 번 진동하는가를 의미하는 단위.
컴퓨터에서 주파수는 결국 clock(클럭)
초 당, 몇 개의 명령어를 처리할 수 있는지에 귀결된다.)

G = 10^9 = 10 0000 0000 = 10억

2 GHz = 20억

1초에 20억 개의 명령을 처리할 수 있다.

Thread

링크 https://github.com/limcholong/LectureContents/blob/main/java/CholongLim/Day19/src/DummyMemo.java

CPU는 오직 한 순간, 한가지 일만 할 수 있다.

20억을 기준으로 하였을 때 2000개는 ?
20 0000 0000 / 2000 = 100 0000 만 배

최고의 운동 신경을 가진 운동 선수가 육안으로 식별할 수 있는 가장 짧은 시간은 몇 초?
= 0.3x 초
1초 → 20억, 0.1초 → 2억, 0.3초 → 6억,
0.3x초 → 6 ~ 7억 사이의 명령어를 처리할 수 있음을 의미한다.

현재 작업표시줄을 보았을 때 나타나는 프로세스가 대략 100개,
100개의 프로세스가 모두 명령 2000개씩을 실행한다고 해봐야 20 0000 << 20만개

만약 모든 프로세스가 0.0001초 씩 동작한다고 가정한다면,
1000개의 프로세스가 모두 한 바퀴를 순회하는데 걸리는 시간은 ?
0.1초
(아주 빠르게 순차적으로 일어나는 것일 뿐 동시에 일어나는 것이 아니다.)
이 모든 과정을 **Multi - Tasking** 이라고 부른다.

: 굉장히 빠르게 순차적으로 일어난다.

(하지만 현재, Context Switching 의 개념이 생략되어 있다.)

Thread

링크 https://github.com/limcholong/LectureContents/blob/main/java/CholongLim/Day19/src/DummyMemo.java

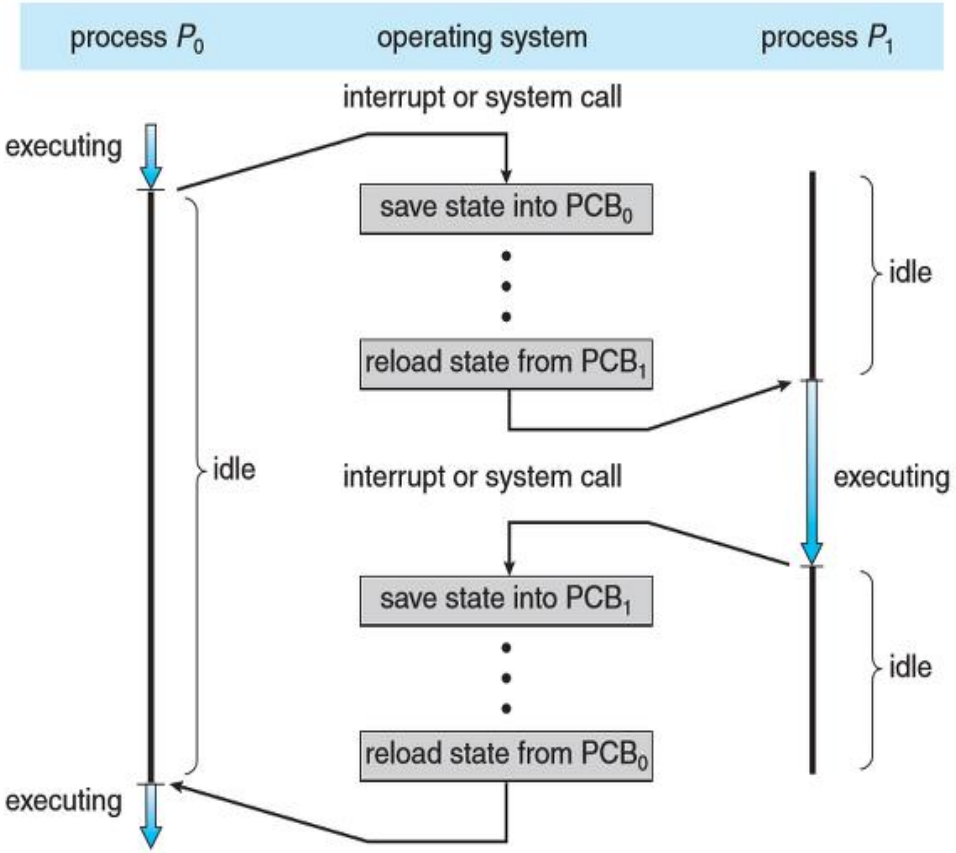


Figure 3.4 Diagram showing CPU switch from process to process.

프로세스 P0와 P1이 존재할 때,
P0이 CPU를 점유 중, P1이 대기중인 상태이다가 잠시 후
P1이 실행되고 P0이 대기가 되는 상태가 찾아온다.

이때, P0이 실행 중에서 대기로 변하게 될 때 지금까지 작업해오던 내용을 모두
'어딘가'에 저장해야 하는데 그것이 PCB라는 곳이다.
즉, P0는 PCB에 저장해야 하고 P1이 가지고 있던 데이터는 PCB에서 가져와야 한다.

이러한 과정에서,
P0 와 P1이 서로 대기 ←→실행을 번갈아 가며 하는 것을
컨텍스트 스위칭(Context Switching)이라고 한다.

: CPU가 어떤 프로세스를 실행하고 있는 상태에서 인터럽트에 의해
다음 우선 순위를 가진 프로세스가 실행되어야 할 때, 기존의 프로세스 정보들은
PCB에 저장하고 다음 프로세스 정보를 PCB에서 가져와 교체하는 작업을 말한다.
컨텍스트 스위칭을 통해 멀티 프로세싱, 멀티 스레딩 운영이 가능하다.

Thread

링크 https://github.com/limcholong/LectureContents/blob/main/java/CholongLim/Day19/src/DummyMemo.java

Context Switching(컨텍스트 스위칭) 도입이 없는 상태

Thread 1

mov eax, 3 // eax = 3 (1)

mov ebx, 4 // ebx = 4 (4)

add eax, ebx // eax = eax + ebx ---> ??? (5) 7 + 4 = 11 ??????

Thread 2

mov eax, 7 // eax = 7 (2)

mov ebx, 2 // ebx = 2 (3)

add eax, ebx // eax = eax + ebx

제어권 변화에 의해 계산이 (1) ~ (5) 순서대로 일어났을 때,
계산값은 얼마인가?

= ????

Context Switching(컨텍스트 스위칭) 도입이 있는 상태

Thread 1

mov eax, 3 // eax = 3 (1)

→ 제어권을 넘기기 전에 현재 하드웨어 레지스터 정보를 메모리에 저장함 (이 정보는 운영체제가 관리하고 있어서 찾을 수 있음)

mov ebx, 4 // ebx = 4 (4)

→ 다시 돌아와서 이전에 저장한 정보를 메모리에서 찾아서 복원함

add eax, ebx // eax = eax + ebx (5)

→3 + 4 = 7 (데이터의 무결성을 보장함)

Thread 2

mov eax, 7 // eax = 7 (2)

mov ebx, 2 // ebx = 2 (3)

→ 자신의 제어권이 역시 넘어가기 전에 하드웨어 레지스터 정보를 백업함

add eax, ebx // eax = eax + ebx

제어권 변화에 의해 계산이 (1) ~ (5) 순서대로 일어났을 때,
계산값은 얼마인가?

Thread

링크 https://github.com/limcholong/LectureContents/blob/main/java/CholongLim/Day19/src/DummyMemo.java

Mutex VS Spinlock

Mutex와 Spinlock의 차이점은 Context Switching의 유무 차이이다.
Mutex는 Context Switching의 있고,
Spinlock은 Context Switching이 없다.

그렇다면 둘 중에 뭐가 더 좋을까?
= 상황에 따라 다르다.

복잡한 작업 시 사용: Mutex
컨텍스트 스위칭을 유발한다.
단순한 작업에 사용 시 스위칭을 사용하게 되면 스위칭 비용이 올라간다.

단순 작업 시 사용: Spinlock
복잡 작업 시 Spinlock 사용할 때 끝날 때 까지
다른 애들은 아무것도 하지 못하게 된다.

따라서, 복잡한 작업일 경우에는 Mutex를 단순한 작업일 때는 Spinlock을 사용하는 것이 효율적이다.

Thread

링크 https://github.com/limcholong/LectureContents/blob/main/java/CholongLim/Day19/src/ThreadTest.java

```java
import java.util.Random;

// Runnable의 경우 run() 메서드에 대한 구현이 필요함
class Car implements Runnable {

    String name;
    private int sleepTime;
    private final static Random generator = new Random();

    public Car(String name) {
        this.name = name;
        // Random 클래스로 만든 객체에 nextInt() 메서드를 통해서도 랜덤값을 생성할 수 있다.
        sleepTime = generator.nextInt( bound: 3000) + 3000;
    }

    // 스레드를 돌릴땐 무조건 이 run() 부분을 구동시키게 되어있다.
    // 매우 중요하니 이 run()을 반드시 기억해두자!
    @Override
    public void run() {
        // try라는 키워드를 적는 경우는 I/O 나 특정한 이벤트,
        // 인터럽트 등등이 발생하는 경우에 작성하게 됨
        // 이 녀석은 에러를 유발할 수도 있다! 를 암시함
        try {
            Thread.sleep(sleepTime);
        } catch (InterruptedException e) {
            // 정말 에러가 발생했다면 여기로 오게 됩니다.
            // 물론 Thread.sleep()에서 에러가 발생할 일은 99.999999999999999%로 없습니다.
            System.out.println("출력도 안 될 것이고 에러가 발생할 일이 없습니다!");
        }
        System.out.println(name + "이 경주를 완료하였습니다!");
    }
}

public class ThreadTest {
    public static void main(String[] args) {
        Thread t1 = new Thread(new Car( name: "Ferrari"));
        Thread t2 = new Thread(new Car( name: "Spyder 918"));
        Thread t3 = new Thread(new Car( name: "Maserati MC20"));

        t1.start();
        t2.start();
        t3.start();
    }
}
```

ThreadTest

```
"C:\Program Files\Java\jdk-15.0.2\bin\java.exe" -javaagent:C:\Users\user\AppD
Maserati MC20이 경주를 완료하였습니다!
Ferrari이 경주를 완료하였습니다!
Spyder 918이 경주를 완료하였습니다!

Process finished with exit code 0
```