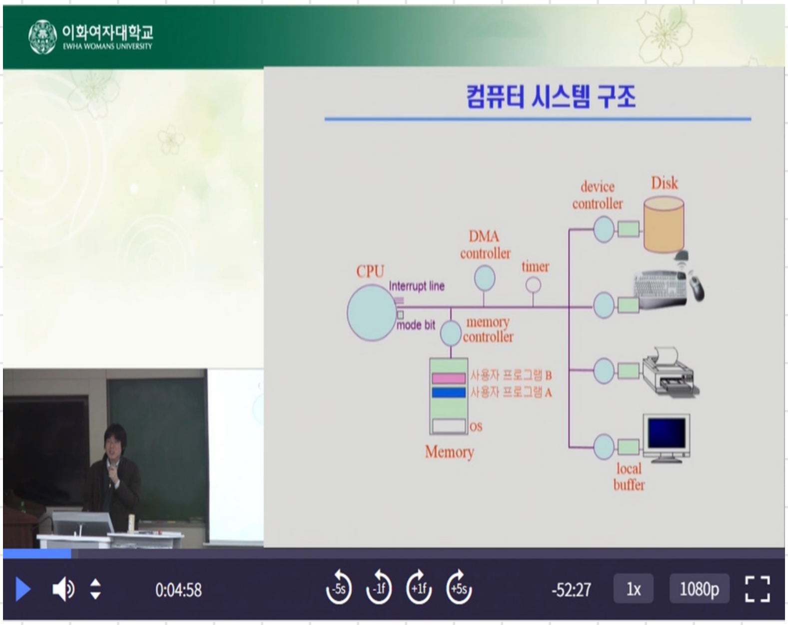


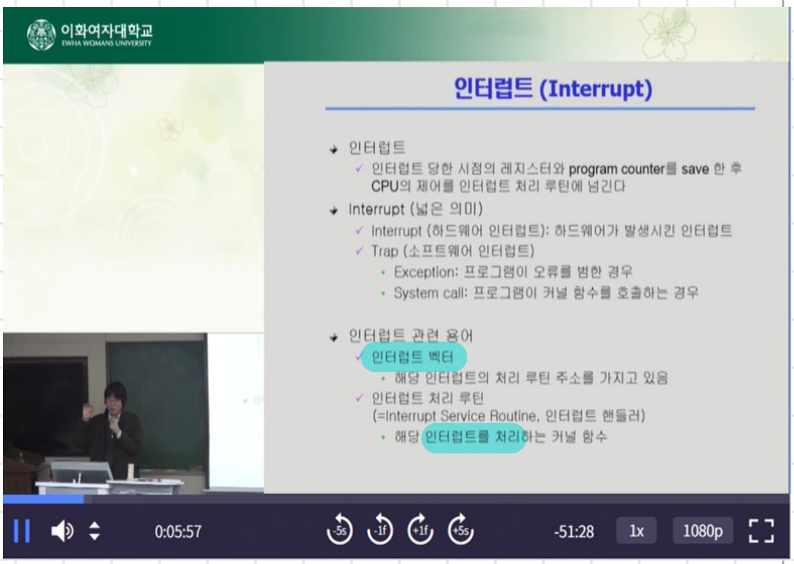
CPU : 4byte 짜리 처리. 메모리에 있는 가리키고 있는 프로그램 카운터(레지스터)(CPU 내에 있음.) 기계어를 읽어들인다.
↳ 4byte 증가시켜서.

기계어 중, 점프하는 인스트럭션을 만나면 Jump! (ex if)

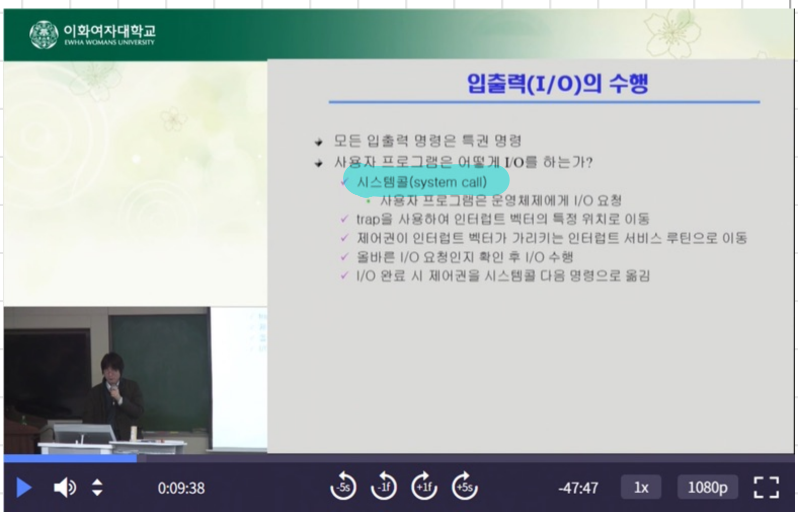
but Interrupt line이 걸려오면, 그 작업우선시 함
(interrupt vector) → 몇

어떻게 행동할 지는 커널함수로 정의되어 있다

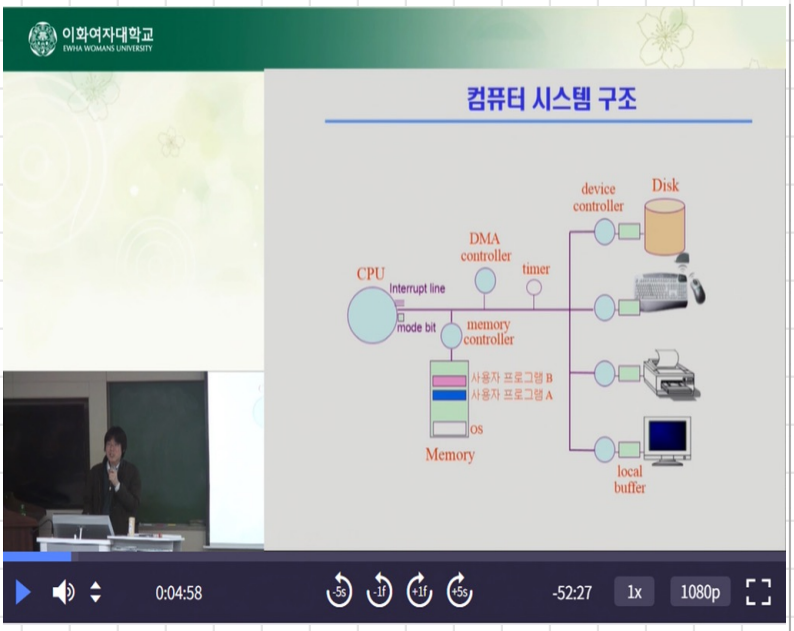
modbit : 0 : 모든 instruction 실행가능. (이때만 I/O 가능)
1 : 한정된 instruction만 실행.



Exception: mode bit이 1일때도 0으로 바꾸려고 한다든지
잘못된 행동을 하면 발생하는 interrupt.

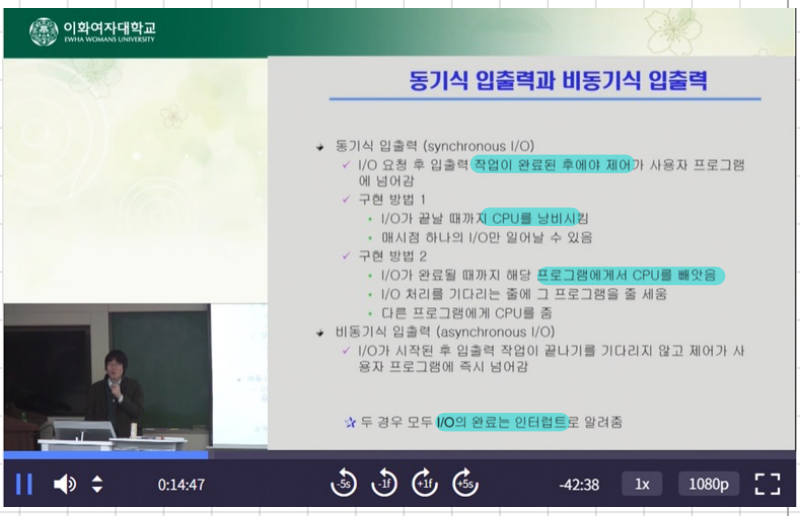


사용자 프로그램이 직접 입출력할 수 없으므로 system call.
의도적으로 interrupt line 설정해서 CPU 제어권이 운영체제에게 넘어감.

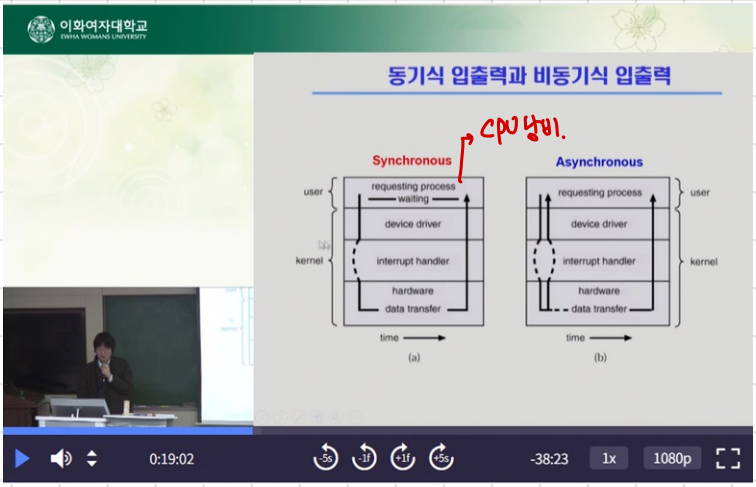


timer가 interrupt 에 사용.

소프트웨어 계속 사용을 방지하기 위해 timer를 설정하여 CPU 제어권 넘겨줌



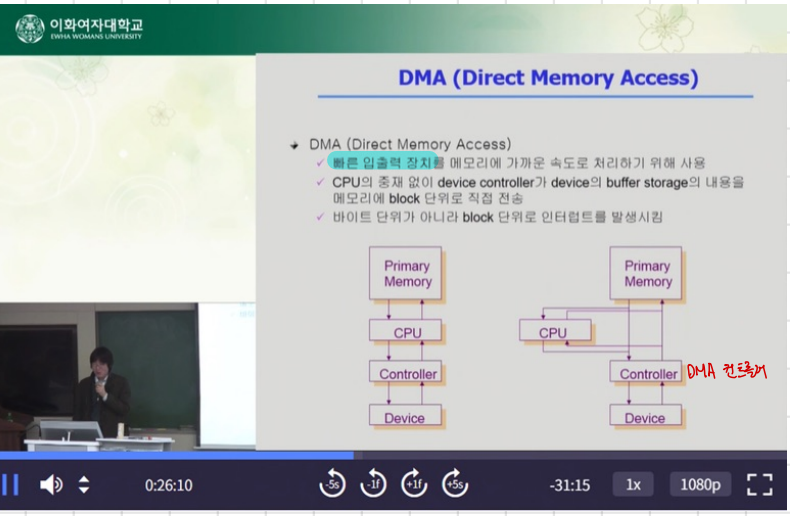
동기식 입출력 :



Synchronous I/O : 요청된 I/O를 바탕으로 작업을 해야하는 경우.

ASynchronous I/O : 읽어온 데이터와 상관없이 작업실행

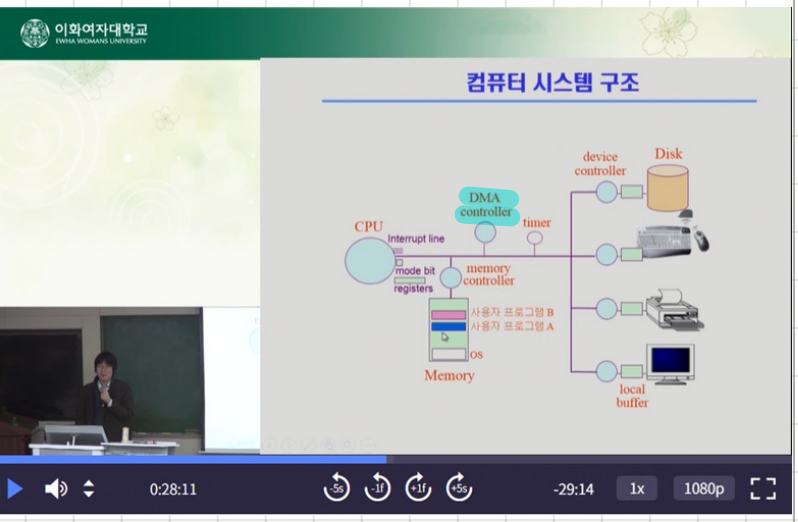
↳ write는 보통 이로 구현.



메모리 접근은 CPU만 가능...

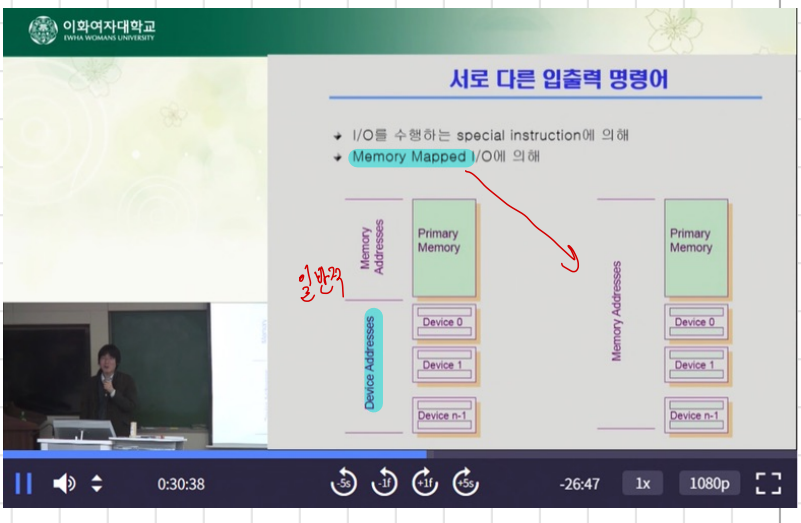
but. DMA도 접근 가능하도록 설계.

CPU만 접근할 경우
interrupt를 너무 자주 당하게 됨.



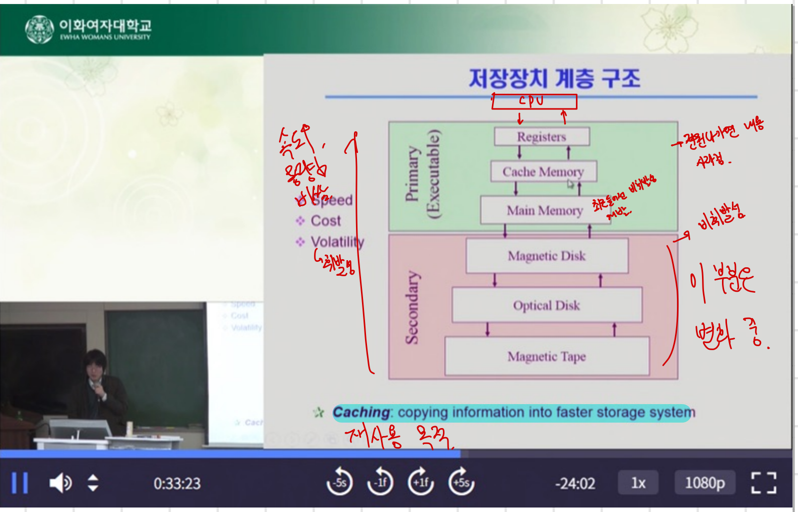
1byte 같은 작은 단위는 I/O 버퍼에 쌓은 후, 차면 1번 CPU에게 interrupt를 걸음.

DMA가 작업하고, interrupt는 작업이 처리됐다고 알려줌.



방법1. 메모리 접근하는 instruction이 있고,
I/O 접근은 special instruction에 의해. I/O 하게 만듦.

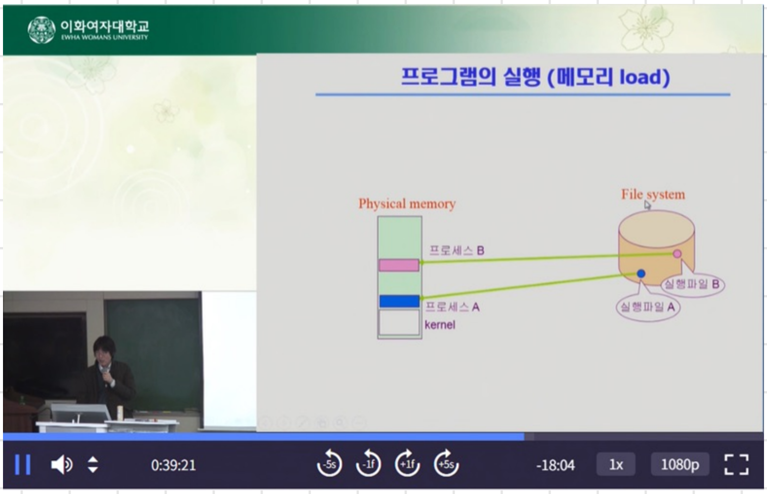
방법2 I/O 장치도 메모리주소에 할당.
따라서 메모리 접근하면 I/O 실행.



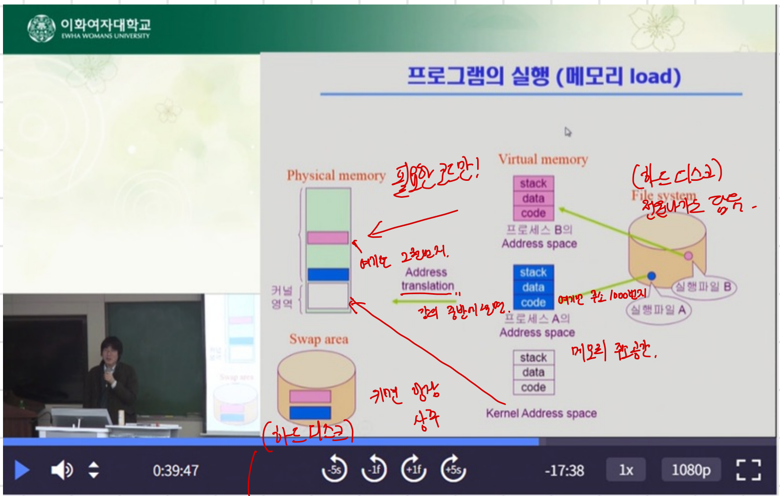
Primary: CPU에서 직접 접근하며 처리 가능한 것. Executable (byte 단위 접근 가능한 것)

Hard disk: sector 단위접근이며 unExcutable!

CPU는 1클럭 당 1개의 instruction을 처리하나, Main 메모리는 100클럭 정도 소요. 그 속도차이를 완화하기 위해 registers, Cache Memory 등.



프로그램은 실행파일 형태로 하드디스크에 저장. 이를 실행시키면 메모리로 올라가서 프로세스가 됨.

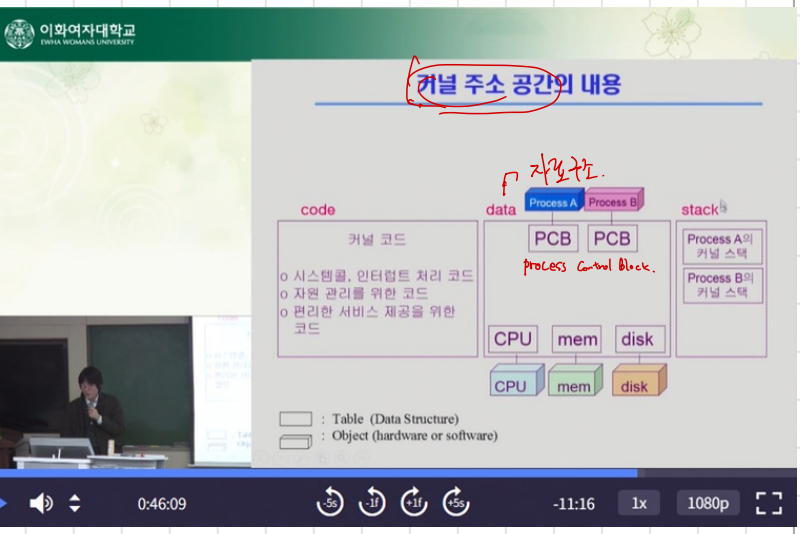


전원 나가면 의미없는 데이터.
메모리 용량의 한계로 담아두는 용도.
강의 후반부 자세히 설명

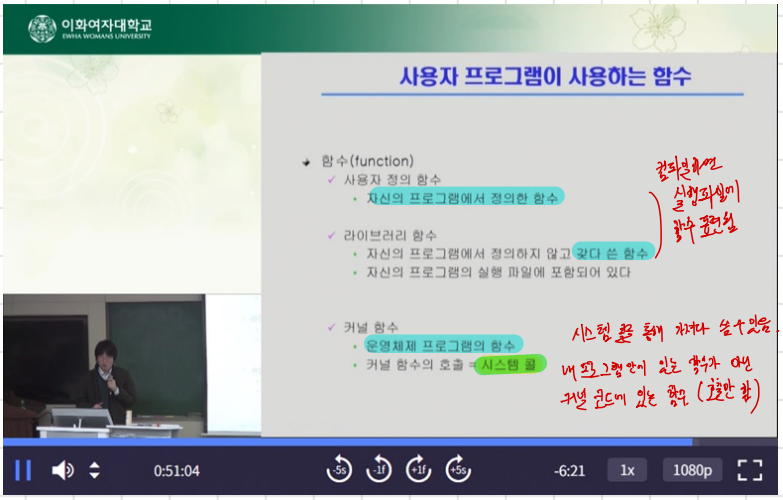
바로 메모리에 올라가는 것이 아니라 가상메모리로 넘어감 ⇒ 실제로 메모리에 올라간 단계는 아님.

독자적인 주소공간 형성. (code, data, stack)
↓ 실행코드. ↓ 자료구조 같은 ↓ 함수

• 실행시 당장 필요한 코드만 올림.
그 이는 Swap area에 내려 놓는다.



인터럽트 코드 : 인터럽트 왔을 때 어떻게 처리해야 하는가?
코드 작성되었음.

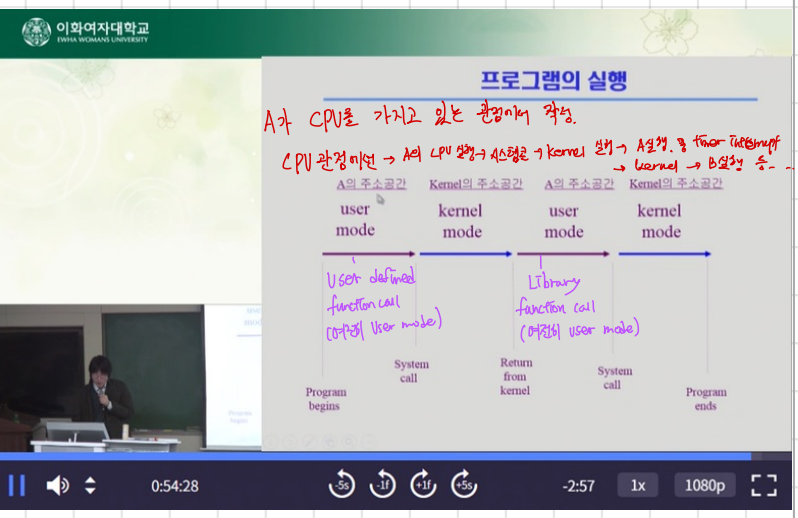


함수를 고급언어가 아닌 instruction level에서도 사용할 수 있다.
모든 프로그램은 함수구조로 되어있음.

컴파일러의 개념과 디버거로 함수정보 보기.

커널함수 호출 이유.
1. 메모리 주소를 프로세스에서 kernel로 점프할 때
일반 Jump는 인스트럭션 물리주소가 아닌 논리적인 주소상 점프

2. 물리적 차원에서 사용과 프로그램이 운영체제 못 건드림.



프로그램이 실행되는 순간.

A가 시작하여 종료될 때까지 모습.