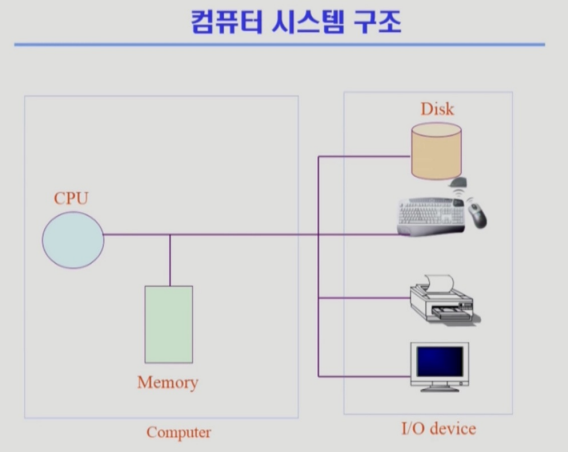
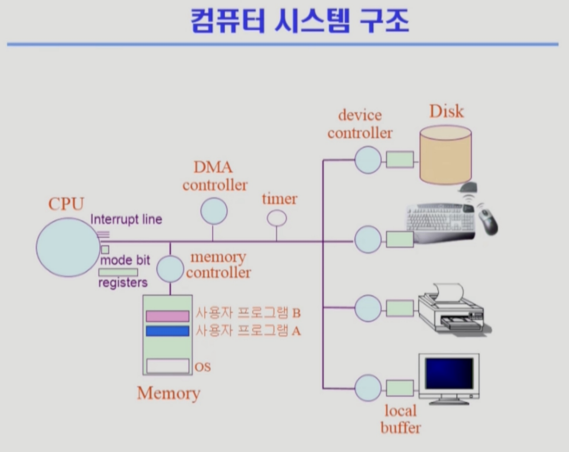
2강 System Structure & Program Execution (시스템 구조 & 프로그램 실행)

: 운영체제를 설명하기에 앞서서 컴퓨터 하드웨어적인 동작에 대해 설명하는 챕터

컴퓨터 시스템 구조





- `메모리`: CPU 의 작업 공간, CPU 의 기계어를 실행
- `CPU` : 메모리에서 기계어를 하나씩 읽어서 수행함
- `하드디스크` : 보조기억장치 & I/O device, 데이터를 읽어서 메모리로 읽어들이기도 하고

  즉, Input device 로의 역할도 하고, 처리 결과를 디스크의 파일 시스템에 저장을 하기도 함

  즉, output device 로의 역할도 수행
- 각각의 I/O device 들은 작은 CPU 같은 것들이 붙어있게 된다. → `device controller` 라고 부름

디스크에서 어떻게 헤드가 움직이고 어떤 데이터를 읽을지 디스크의 내부를 통제하는 것은 CPU 의 역할이

아니고 디스크에 붙어있는 디스크 컨트롤러가 작업을 하게 된다.

- `Interrupt line` : 주변기기가 CPU에게 어떤 사실을 알리는 일을 Interrupt 라고 하고 이런 라인

- `timer` 하드웨어 : 특정 프로그램이 CPU를 독점하는 것을 막기 위한 것

  - 정해진 시간이 흐른 뒤 운영체제에게 제어권이 넘어가도록 인터럽트를 발생시킴

  - 타이머는 매 클럭 틱 때마다 1씩 감소

  - 타이머 값이 0이 되면 타이머 인터럽트 발생

  - CPU를 특정 프로그램이 독점하는 것으로부터 보호

  ⇒ 타이머는 time sharing을 구현하기 위해 널리 이용됨

  ⇒ 타이너는 현재 시간을 계산하기 위해서도 사용

- `Mode bit`

: 사용자 프로그램의 잘못된 수행으로 다른 프로그램 및 운영체제에 피해가 가지 않도록 하기 위한 보호 장치 필요

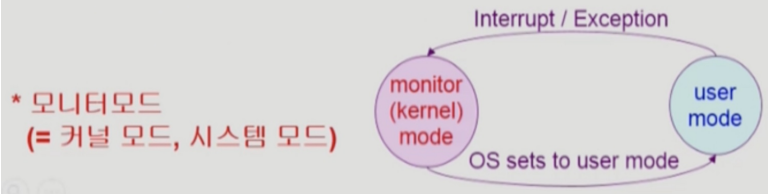
⇒ Mode bit 을 통해 하드웨어적으로 두 가지 모드의 operation 지원

```
Mode bit 역할
1 사용자 모드 : 사용자 프로그램 수행
0 모니터 모드* : OS 코드 수행
```

- 보안을 해칠 수 있는 중요한 명령어는 모니터 모드에서만 수행 가능한 '특권 명령'으로 규정

- Interrupt 나 Exception 발생시 하드웨어가 mode bit 을 0으로 바꿈

- 사용자 프로그램에게 CPU를 넘기기 전에 mode bit을 1로 셋팅



- mode bit 이 1일때는 사용자 프로그램이 CPU를 가지고 있을 때는 제한된 지시만 CPU 에서만 실행을 할 수 있게 되어 있다. 보안상의 목적이 있다.

- 운영체제가 CPU를 가지고 있을 때는 아무것이나 해도 되지만 사용자 프로그램한테 CPU 를 넘겨줄 때 Mode bit을 1로 바꿔서 넘겨준다. 그러면 사용자 프로그램이 instruction 을 실행하다가 mode bit이 1인 것을 보고 instruction 실행이 안되게 하드웨어적 구현을 해 놓은 것

- mode bit이 0일 때는 아무거나 CPU에서 실행이되고 CPU를 운영체제가 사용자 프로그램을 넘겨줄때 mode bit을 1로 바꿔서 넘겨주기 때문에 한정된 instruction만 실행을 할 수 있게 되어있다.

- Interrupt가 들어오게 되면 cpu 제어권이 운영체제에게 넘어가면서 mode bit은 자동으로 0으로 바뀌게 된다.

- `Device Controller`

  - I/O device controller

    : 해당 I/O 장치유형을 관리하는 일종의 작은 CPU

    : 제어 정보를 위해 control register, status register 를 가짐

    : local buffer를 가짐(일종의 data register)

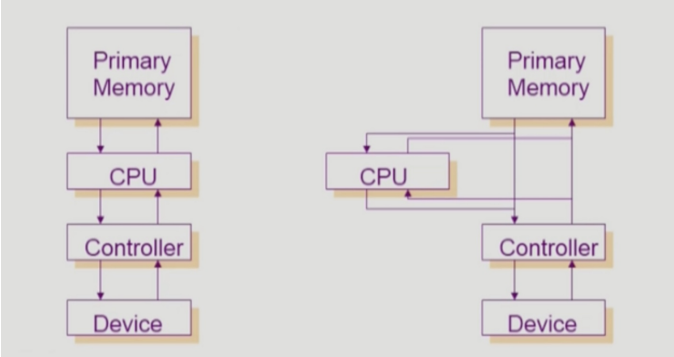
  - I/O 는 실제 device 와 local buffer 사이에서 일어남

  - Device controller는 I/O 가 끝났을 경우 interrupt 로 CPU 에 그 사실을 알림

  + `device driver`(장치구동기) : OS 코드 중 각 장치별 처리루틴 → software

  + `device controller`(장치제어기) : 각 장치를 통제하는 일종의 작은 CPU → hardware

- `DMA(Direct Memory Access) Controller`



- 빠른 입출력 장치를 메모리에 가까운 속도로 처리하기 위해 사용

- CPU의 중재 없이 device controller 가 device의 buffer storage 의 내용을 메모리에 block 단위로 직접 전송

- 바이트 단위가 아니라 block 단위로 인터럽트를 발생시킴

: 원래는 메모리에 접근할 수 있는 장치는 CPU 뿐이었는데,

DMA Controller 를 두게 되면 메모리를 CPU도 접근할 수 있고, DMA도 CPU에 접근할 수 있게 되어있다.

: I/O 장치가 너무 자주 interrrupt 를 거니까 CPU가 방해를 많이 받는다. 이것을 막기 위해서 중간 중간 작업이 들어왔을 때 그것을 CPU 한테 인터럽트를 걸어서 CPU가 카피하게하는게 너무 크다 그래서 그런식으로 하지않고 씨피유는 자기일을 하고 중간중간에 들어오는 내용이 작업이 끝났으면 DMA가 직접 여기있는 내용을 메모리로 복사하는 일까지 해준다. 그 작업이 다 끝나면 씨피유한테 인터럽트를 한번만 걸어서 그 내용이 다 메모리에 올라왔다 했다라는 것을 보고를 해주고 그렇게 되면 씨피유가 중간에 인터럽트 당하는 그런 빈도가 줄어들어서 빠른 장치를 좀더 효율적으로 쓸수가 있다

- `memory controller`

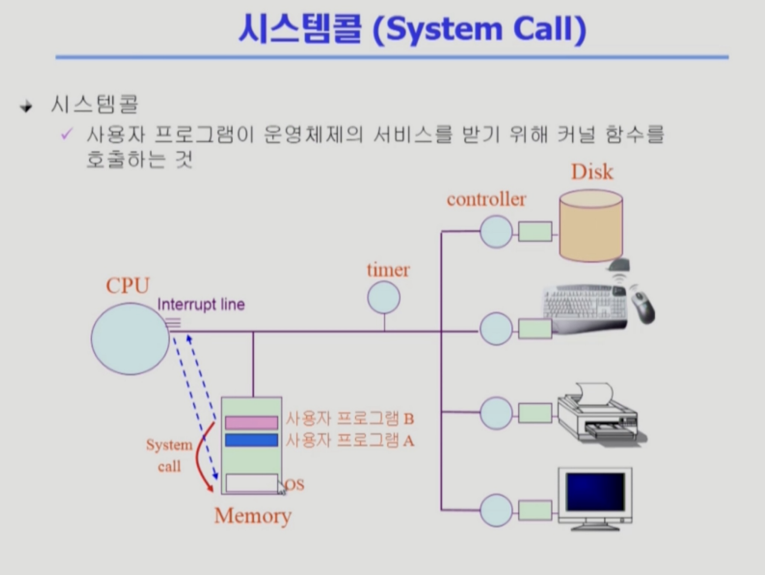
: 둘이서 만약에 특정 메모리영역을 동시에 접근하거나 이럴때 문제가 생길 수 있어서 이런 것을 중재하는 역할을 담당한다. 누가 먼저 접근하게 할지 교통정리하게 하는 역할을 한다.

- `입출력(I/O)의 수행`

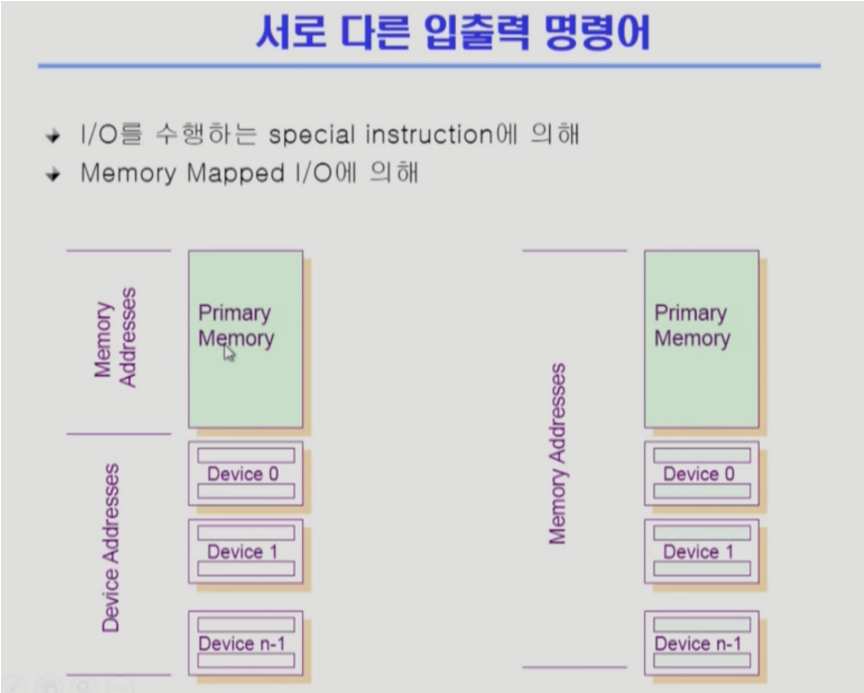
: 모든 입출력 명령은 특권 명령

: 사용자 프로그램은 어떻게 I/O를 하는가?

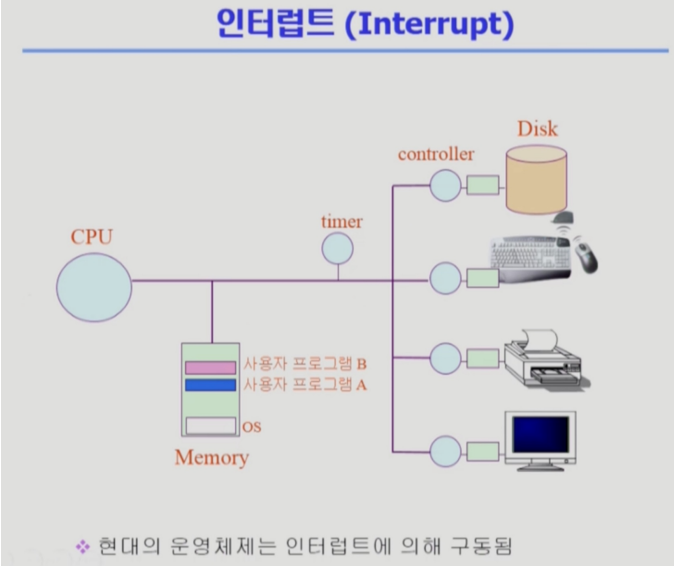
- `시스템 콜(system call)` : 사용자 프로그램은 운영체제에게 I/O 요청



- trap을 사용하여 인터럽트 벡터의 특정 위치로 이동
- 제어권이 인터럽트 벡터가 가리키는 인터럽트 서비스 루틴으로 이동
- 올바른 I/O 요청인지 확인 후 I/O 수행
- I/O 완료시 제어권을 시스템콜 다음 명령으로 옮김



- `인터럽트(Interrupt)`



: Interrupt 당한 시점의 레지스터와 program counter를 save한 후 CPU의 제어를 Interrupt 처리 루틴에 넘긴다.

⇒ Interrupt (넓은 의미)

* `Interrupt(하드웨어 Interrupt)` : 하드웨어가 발생시킨 Interrupt

* `Trap(소프트웨어 Interrupt)` - Exception : 프로그램이 오류를 범한 경우

- System call : 프로그램이 커널 함수를 호출하는 경우

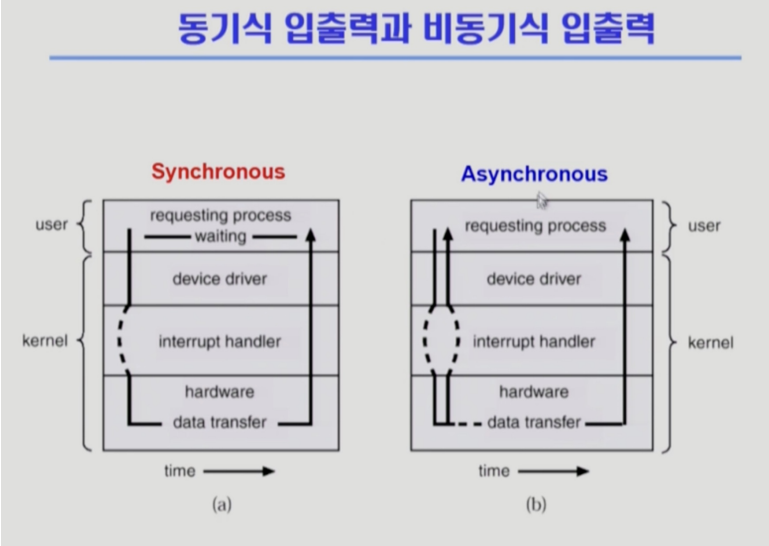
* Interrupt 관련 용어

- `Interrupt 벡터` : 해당 Interrupt의 처리 루틴 주소를 가지고 있음

- `Interrupt 처리 루틴(= Interrupt Service Routine, Interrupt 핸들러)`

: 해당 Interrupt 를 처리하는 커널함수, 실제 해야할 일

동기식 입출력과 비동기식 입출력



- `동기식 입출력(synchronous I/O)`

  : I/O 요청 후 입출력 작업이 완료된 후에야 제어가 사용자 프로그램에 넘어감

  - **구현방법 1**

    : I/O 가 끝날 때까지 CPU를 낭비시킴

    : 매시점 하나의 I/O만 일어날 수 있음

  - **구현방법 2**

    : I/O가 완료될 때까지 해당 프로그램에게서 CPU를 빼앗음

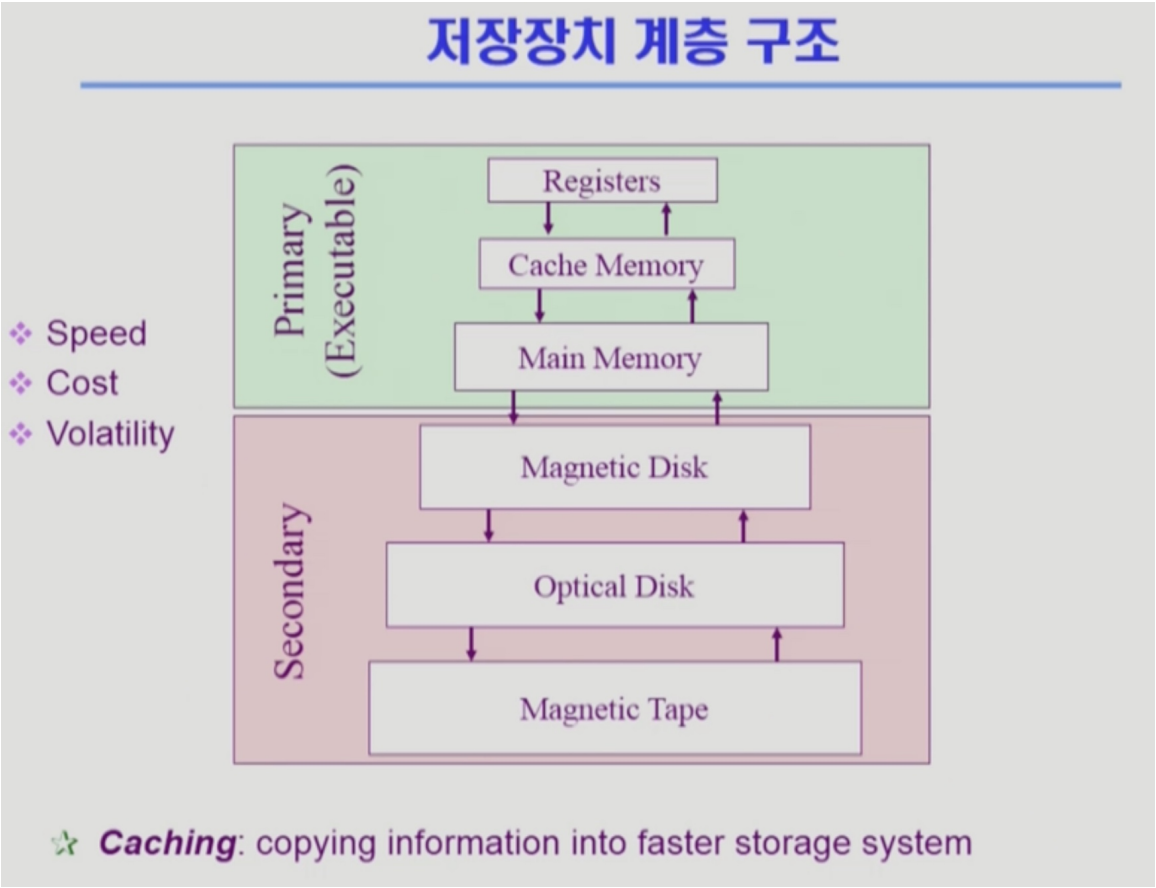
    : I/O 처리를 기다리는 줄에 그 프로그램을 줄 세움

    : 다른 프로그램에게 CPU를 줌

- `비동기식 입출력(asynchronous I/O)`

  : I/O 가 시작된 후 입출력 작업이 끝나기를 기다리지 않고 제어가 사용자 프로그램에 즉시 넘어감

⇒ 두 경우 모두 I/O의 완료는 인터럽트로 알려줌



맨위에 CPU 가 있다

연두색 : 휘발성 매체

분홍색 : 비휘발성 매체

요즘에는 메인메모리도 비휘발성 매체가 있기도 하지만 전통적으로는 휘발성 매체

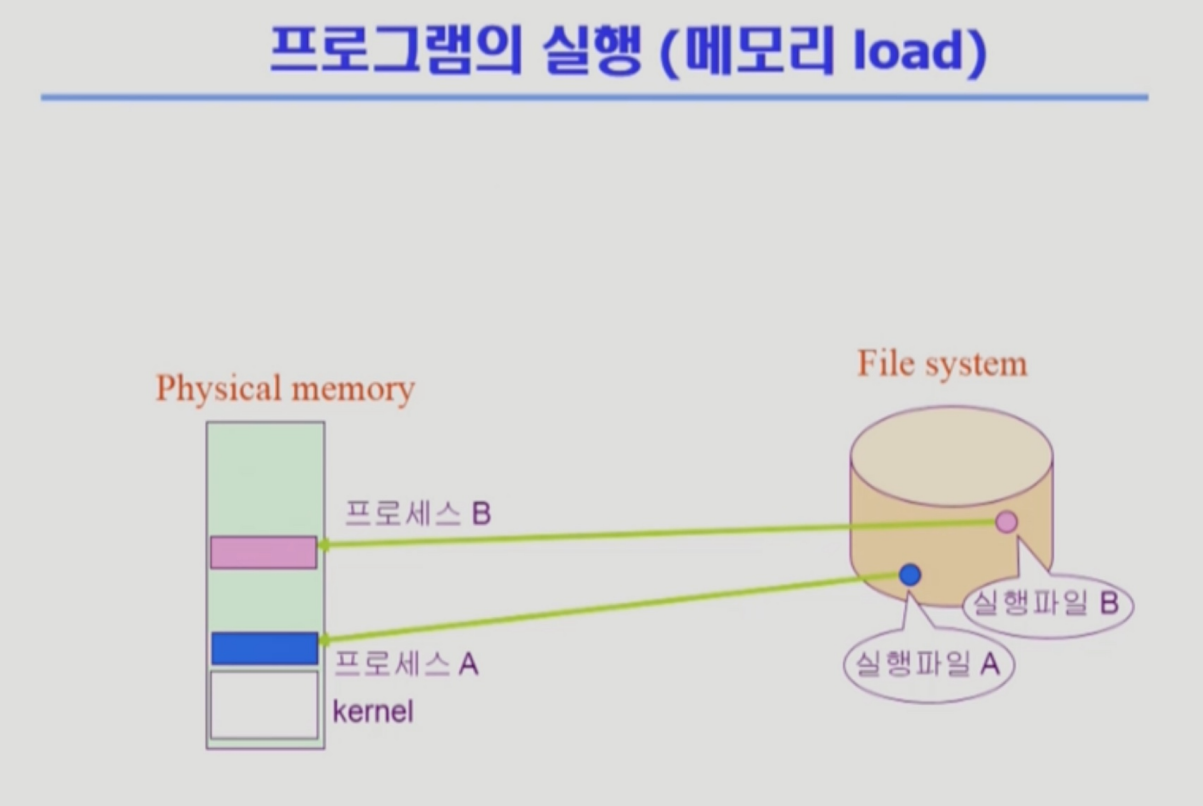
CPU에서 직접 접근할 수 있는 메모리 스토리지 매체를 프라이머리

CPU에서 직접 접근하지 못하는 메모리를 세컨더리 라고 부름

하드디스크는 바이트 단위로 접근이 안됨

보통 캐싱은 재사용을 목적으로 함

→ 프로그램이 어떻게 컴퓨터에서 실행이 되는가

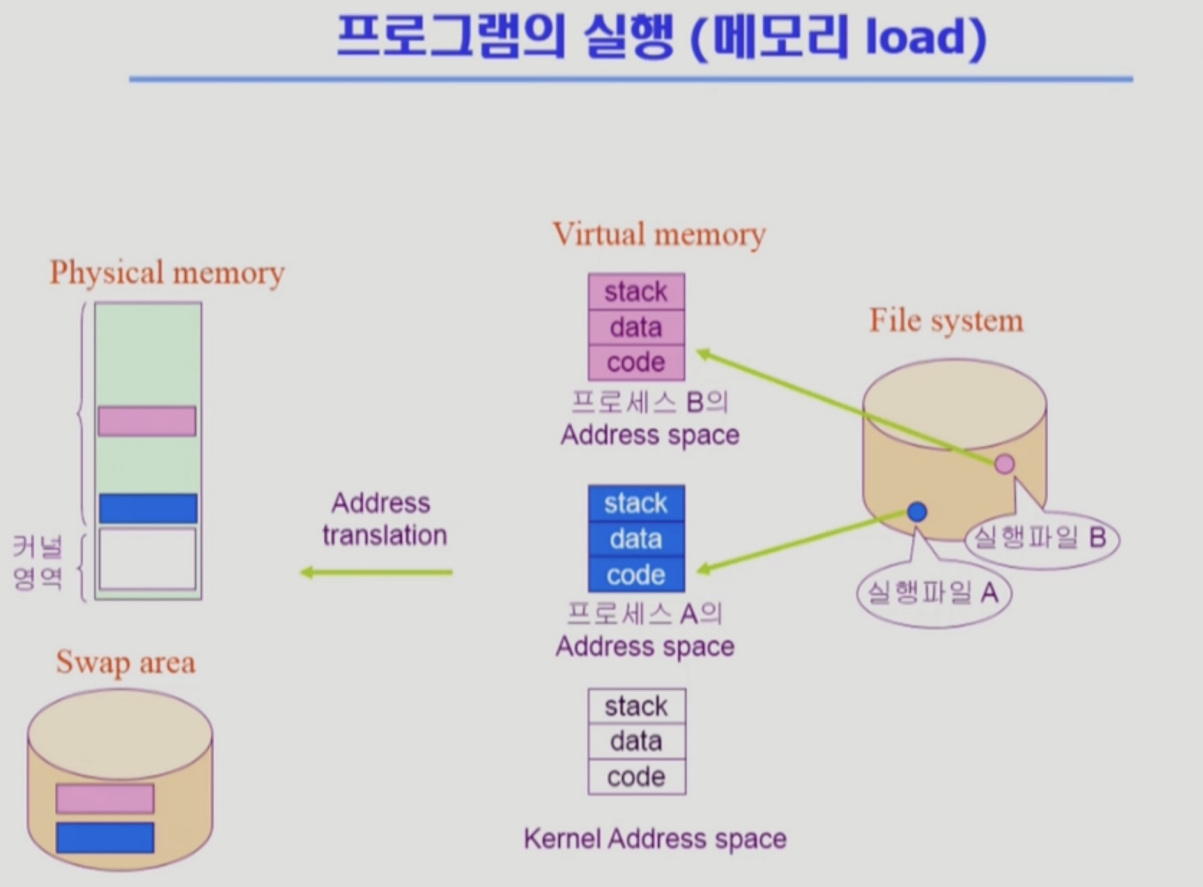


보통 프로그램이라는 것은 실행파일 형태로 하드디스크에 저장이 되어있다.

파일시스템에 파일형태로 저장이 되어있고 실행을 시키게 되면 메모리로 올라가서 프로세스가 된다.

그러면서 실행이 되는 것이다.

정확하게는 물리적인 메모리에 바로 올라가는 것이 아니라 중간에 한단계를 거치게 되는데 그게 바로 버추얼메모리 가상메모리라는 단계이다.



어떤 프로그램을 실행시키게 되면 그 프로그램에 address space 우리말로 주소공간이 형성이 되는데 메모리 주소공간이다. 0번지부터 시작하는 그 프로그램만의 독자적인 주소공간이 생기게 된다.

예를들어, A 라는 프로그램을 실행시키면 프로그램 A의 주소공간이 0번지부터 만들어지고 B라는 프로그램을 실행시키면 0번지부터 시작하는 자기자신만의 독자적인 주소공간이 만들어진다. 이러한 주소공간은 각 프로그램마다 만들어지는 주소공간을 code, data, stack 이런 영역으로 구성이 된다.

코드는 프로그램 기계어 코드 CPU에서 실행할 기계어 코드를 담고있고 데이터는 변수라던지 전역변수같은 프로그램을 사용하는 자료구조를 담고있다.

스택은 코드가 함수구조로 되어있기 때문에 함수를 호출하거나 리턴할때 데이터를 쌓았다가 꺼내가는 이런 용도로 사용하는 게 스택영역

모든 프로그램이 이런 독자적인 주소공간을 가지고 있는데 이것을 물리적인 메모리에 올려서 실행을 시키는 것이다.

그런데 물리적인 메모리는 커널은 처음에 컴퓨터를 켜서 부팅을 하면 커널은 메모리에 항상 상주해서 올라가 있지만 이러한 사용자 프로그램들을 실행을 시키면 주소공간이 생겼다가 프로그램을 종료 시키면 사라지는 것이다.

프로그램을 실행시켰을때 마다 만들어지는 주소공간을 물리적인 메모리에 통째로 다 올려놓는 것이 아니다.

메모리 공간이 낭비되기때문에 그래서 당장 필요한 부분에 해당하는 코드만 올려놓고 그렇지 않은 것은 올려놓지 않는다. 그래야 메모리가 낭비되지 않는다. 그래서 사용이 안되면 메모리에서 쫓아내게 된다. 경우에 따라서 지워버리면 되지 않고 보관하고 있어야 하는 것이 있을 것이다. 주소공간 중에서 당장 필요한 것은 이렇게 물리적인 메모리에 올려놓고 그렇지 않은 부분은 디스크에 내려놓게 된다.

디스크에서 swap area에 내려놓게 된다. 즉 프로그램마다 만들어진 독자적인 주소공간은 사실 머릿속에만 있는 주소공간이다. 실제로 연속적으로 어디에 할당되는 것이 아니라 쪼개져서 어떤부분은 여기 어떤부분은 swap memory에 내려와있고 그렇다. 그래서 virtual memory 라고 부르는 것이다. 메인 메모리의 연장부분으로 하드디스크를 사용하는 것을 swapping swap 목적으로 사용한다고 하고 ,

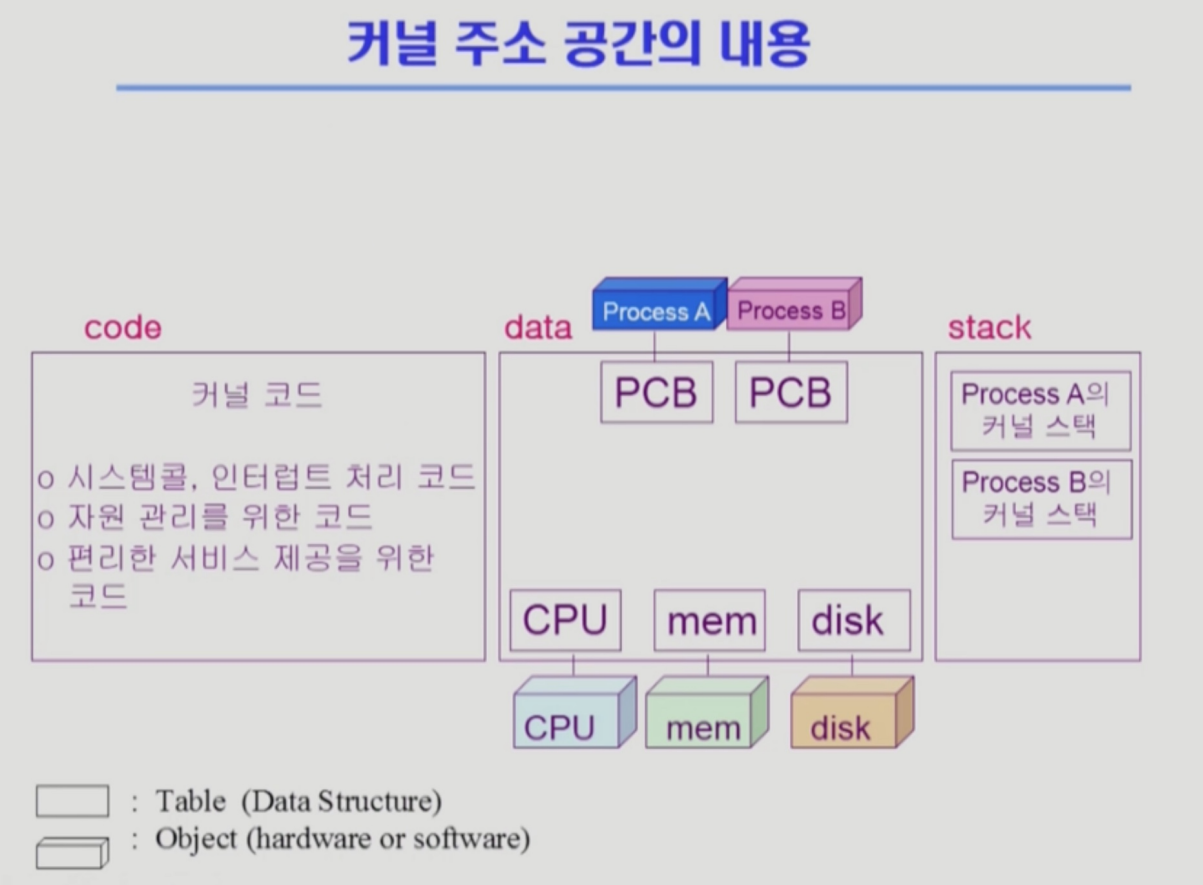
virtual memory 라는 것은 각 프로그램마다 독자적으로 가지고 있는 이 메모리 주소공간을 virtual memory라고 부르는 것

swap memory 로 사용하는 하드디스크는 전원이 나가면 의미없는 데이터이다. 이유는 전원이 나가면 프로세스는 종료가 된다. 메모리에 있는 내용도 사라지기때문에 swap area에있는 정보도 사실 의미가 없어진다.

그다음에 각 프로그램마다 0번지부터 시작하는 주소공간이 있는데 이 물리적인 메모리도 0번지부터 시작하는 주소이다.

메모리 주소변환을 해주는 계층이 있다. 운영체제가 할 수 있는건 아니고 하드웨어의 지원을 받아서 하드웨어 장치가 따로 있다.

운영체제 커널도 사실 하나의 프로그램이기 때문에 코드 데이터 스택 이런 식의 주소공간으로 구성이 되어있다.



커널에는 어떤 코드들이 있느냐

자원을 효율적으로 관리하는 일을 하기때문에 그것하고 관련된 코드가 운영체제에 있을 것이고, 사용자에게 편리한 인터페이스를 제공하는게 운영체제의 역할 관련된 코드들이 있을 것이다. 또 다르게 설명하면 운영체제라는 것은 언제 CPU를 어떻게 되느냐면 인터럽트가 들어오면 CPU를 얻게 된다. 각각의 interrupt마다 무슨일을 처리해야하는지 운영체제 커널의 커널코드의 함수 형태로 구현이 되어있을 것이다. 시스템콜이나, 하드웨어가 발생시키는 interrupt에 대해서 무슨일을 해야할 지 커널의 코드부분에 들어있게 된다.

커널의 데이터부분에는 운영체제에 사용하는 자료구조들이 정의가 되어있다.

CPU, 메모리 , 디스크 와 같은 하드웨어들을 직접 관리하고 통제한다.

하드웨어 종류마다 자료구조를 하나씩 만들어서 관리를 하고 있다.

운영체제는 또 프로세스들을 관리함

현제 실행중인 프로그램들을 관리해야하기 때문에 각 프로그램들이 독자적인 주소공간을 가지고 있지만 이러한 것들은 관리하기 위한 그런 자료구조가 필요하다. CPU를 얼마나 썼는

지 메모리를 다음에 얼마나 누구에게 주어야할지 각 프로그램마다 관리하고 있는 자료구조가 필요하다. 그것을 우리가 PCB 라고 부른다 프로세스 컨트롤 블럭

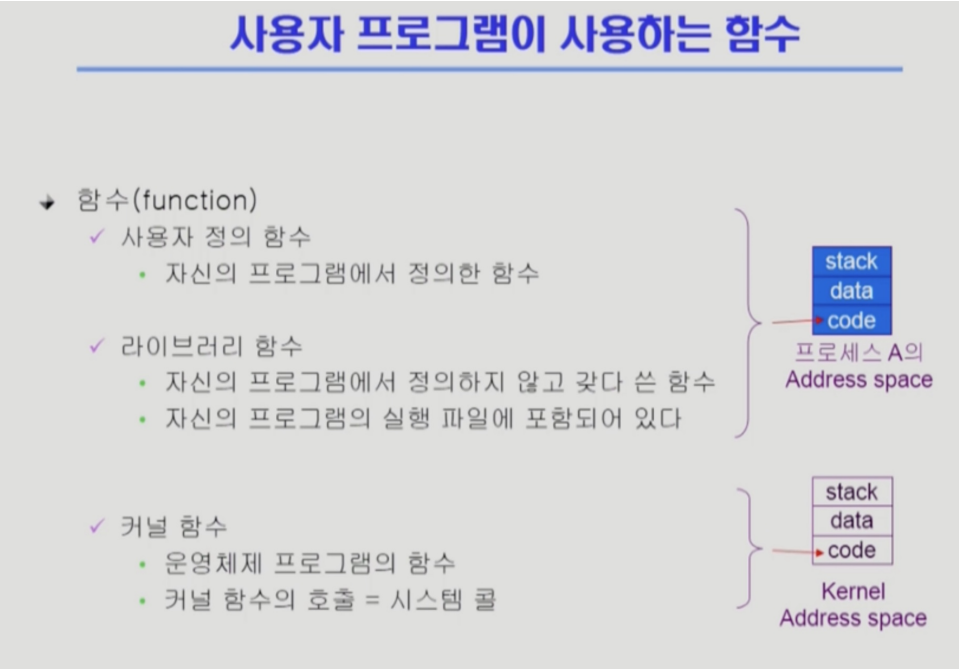
시스템안에 프로그램이 하나 돌아가면 그 프로그램을 관리하기 위한 그러한 자료구조가 운영체제 커널에 하나씩 만들어지는데 그것이 PCB 라는 것 프로세스마다 만들어짐

운영체제도 함수구조로 코드가 짜여져 있기 때문에 함수를 호출하거나 리턴할때 스택영역을 사용해야한다.

그래서 커널스택이라는 것이 있는데 운영체제코드는 여러 프로그램들이 사용자들의 요청에 따라서 불러서 사용할 수가 있다. 사용자 프로그램들이 운영체제 커널의 코드를 불러서 실행하기때문에 여기서 함수호출을 하게 되면 커널스택을 쓸때 어떤 사용자 프로그램이 커널의 코드를 실행 중인 가에 따라서 사용자 프로그램마다 커널 스택을 따로 두고 있다.

프로세스마다 커널스택을 별도로 그려놓은 이유이다.

사용자 프로그램이 사용하는 함수



`함수(function)`

- 사용자 정의 함수

  : 자신의 프로그램에서 정의한 함수

- 라이브러리 함수

: 자신의 프로그램에서 정의하지 않고 갖다 쓴 함수

: 자신의 프로그램의 실행 파일에 포함되어 있다.

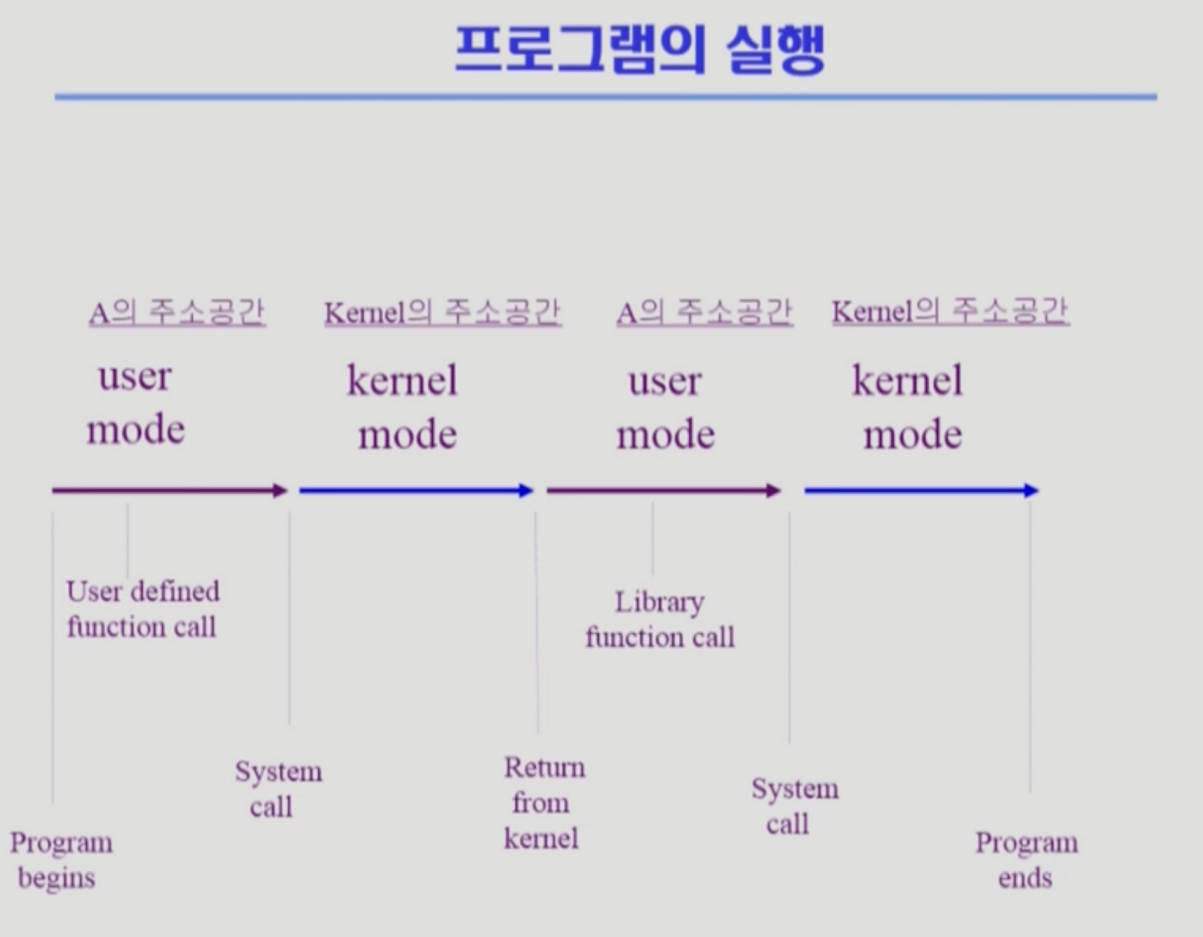
- 커널 함수

: 운영체제 프로그램의 함수

: 커널 함수의 호출 = 시스템 콜

을 통해서 interrupt line 을 셋팅해서 CPU 제어권이 커널로 넘어오게 해서 커널함수를 실행하게 함

: 영역 자체가 다르기때문에 점프가 불가능하다



컴퓨터 시스템의 일반적인 구조

CPU , 메모리 - 컴퓨터 / 다른말로는 호스트

컴퓨터에 붙어서 데이터를 컴퓨터 안에 넣는 인풋이나 또는 호스트에서 처리된 결과를 내보내는 아웃풋을 하는 디바이스들

디바이스들을 전담하기 위한 컨트롤러들이 붙어있고 컨트롤러들은 이러한 아이오디바이스를 통해서 데이터를 읽거나 쓰거나 하면서 작업 공간인 로컬 버퍼에다가 데이터를 저장함

CPU 뭔가를 알려주고 싶을때 컨트롤러가 인터럽트라는 것을 걸어서 뭔가를 전달할 일이있다는것을 알려주게 되고

CPU 는 매 순간 메모리 어딘가에 올라와있는 기계어를 처리해야한다.

우리말로 기계어 CPU Instruction 0과 1의 조합으로 된 기계어 집합에 따라서 어떠한 기계어들이 있는지는 다르다.

CPU는 매순간 메모리에 어떤 위치에 있는 기계어를 하나씩 읽어와서 실행을 하게 된다.

메모리 어디있는 기계있는 기계어를 읽어오는가

CPU안에 있는 레지스터 안에서 메모리 주소를 가지고 있는 레지스터가 있다.

프로그램카운터 레지스터라는 것

그것이 가리키고 있는 메모리 위치에서 인스트럭션을 하나 읽어와서 CPU가 하나 실행하고 그다음에 메모리를 가리키고 있던 프로그램카운터라는 레지스터는 다음주소를 가리키게 됨

아까 인스트럭션이 보통 4바이트. CPU에서 하나 실행하게 되면 다음 인스트럭션 주소인 즉 메모리 다음 인스트럭션을 실행할 위치인 프로그램카운터가 4가 증가한다. 특별한 일이 없으면 CPU는 항상 메모리 다음에 있는 인스트럭션 그다음 인스트럭션 을 순차적으로 실행을 하게 됨

프로그램이 항상 순차적이지는 않음 함수호출이나 또는 제어구조에서 if 문을 만족안할때는 뭔가 다른메모리주소를 점프해서 인스트럭션을 수행해야함

보통 기계어 집합중에서 쩜프하는 인스트럭션이 있다. 메모리에 항상 그 다음 위치에 있는 인스트럭션을 순차적으로 실행하는 것이 아니고 점프하는 인스트럭션을 만나면 메모리 위치를 점프해서 좀 덤 멀리있는 인스트럭션을 수행할 수 있고 이런 싞으로 정의가 되어있다.

CPU는 아주빠른 일꾼이라고 생각하면 됨

마음대로 할 수 있는건 아니고 프로그램카운터라는 CPU안에 레지스터가 가리키고 있는 메모리에서 인스트럭션을 읽어서 실행하는.. 늘 이것만 함..

CPU는 항상 프로그램카운터라는 레지스터가 가리키는 메모리 주소에서 인스트럭션을 하나 읽어서 실행하는 이 일만 하게됨

그런데 인스트러ㅓㄱ션을 하나 실행하고 나서 다음 인스트럭션을 실행하기 전에 하는일이 있음

그것은 인터럽트가 들어온게 있는지 체크하는것 만약 있다면 지금 프로그램카운터가 가리키고 있는 그 메모리 주소에서 인스트럭션을 읽어서 수앻하는게 아니라 잠시 지금것을 멈추고 인터러트가ㅏ 들어오면 CPU를 누가쓰고있었든 상관없이 CPU제어권이 운ㅇ영체제로 넘어가게 됨

운영체제는 인터럽트마다 무슨 왜 인터럽트가 걸렸는지 상황에 맞게 처리해야할 일들이 운영체제 안에 커널함수로 정의가 되어있음

인터럽트 라인별로 인터럽트 벡터 설명,,

CPU는 매번 프로그램카운터가 가리키고 있는 곳을 실행하게 되는데 인터럽트가 들어오면 운영체제한테 넘어감

CPU안에 모드빗이라는 것이 있어서 모드빗이 0이냐 1이냐에 따라서 모드빗이 0이면 CPU가 수행할 수 있는 기계어집합을 다 실행할 수 있고 1일때는 한정된 기계어들만 한정된 인스트럭션들만 실행할 수 있게 되어있음

운영체제가 CPU를 가지고 있ㅇ르때는 모드빗이 0이라서 모든것을 실행할 수 있음

사용자프로그램의 메모리영역을 본다거나 IO 디바이스를 접근한다거나 운영체제만 할 수 있는 모드빗이 0일때만 할 수 있는 특별한 인스트럭션으로 정의되어있고

1일때는 사용자 프로그램이 실행되는 CPU를 가지고 있을때이다. 한정된 인스트럭션만 실행하게되는데 뭔가는 못함 무엇을 못하냐면 사용자 프로그램을 100퍼센트 믿을수 없다 남의 프로그램의 메모리 주소를 보려고 할수도 있고 커널의 메모리주소에가서 이상한 것을 써놓는 것을 막고 사용자프로그램이 CPU를가지고있을때는 자기 영역만보고 일을 해야함

그것을 위해 모든 기계어를 다 실행하지는 못하도록 막아놓고 있음

또한가지 iO 디바이스를 접근하는 모든 인스트럭션은 모드빗이 0일때만 실행ㅎ될수있게 즉 운영체제ㅁ나 실행될수있게 막아놓음

사용자 프로그램이 CPU를가지고 인스트럭션을 사용하다가 IO작업을 해야겠다 그러면 본인이 직접 못함

운영체제에게 요청을 해야함

운영체제한테 요청을 하려면 프로그램 카운터 레지스터가 운영체제에 주소영역으로 점프를 해야하는데 그 쪽을 ㅗ점프하는 것음 1일때 못하기 때문에 사용자 프로그램이 운영체제에게 서비스를 요청할 때는 시스템 콜잉라는것을 한다.

운영체제에있는 함수를 사용자 프로그램이 요청하는 것이다.

시스템 콜은 어떻게하느냐 바로 점프하는것은 불가능 의도적으로 인터럽트라인을 세팅함

CPU는 하던일을 멈추고 제어권이 사용자 프로그램에서 운영체제로 넘어감

IO를 대신해달라 운영체제에게 서비스를 대신 해달라고 요청을 하게 됨