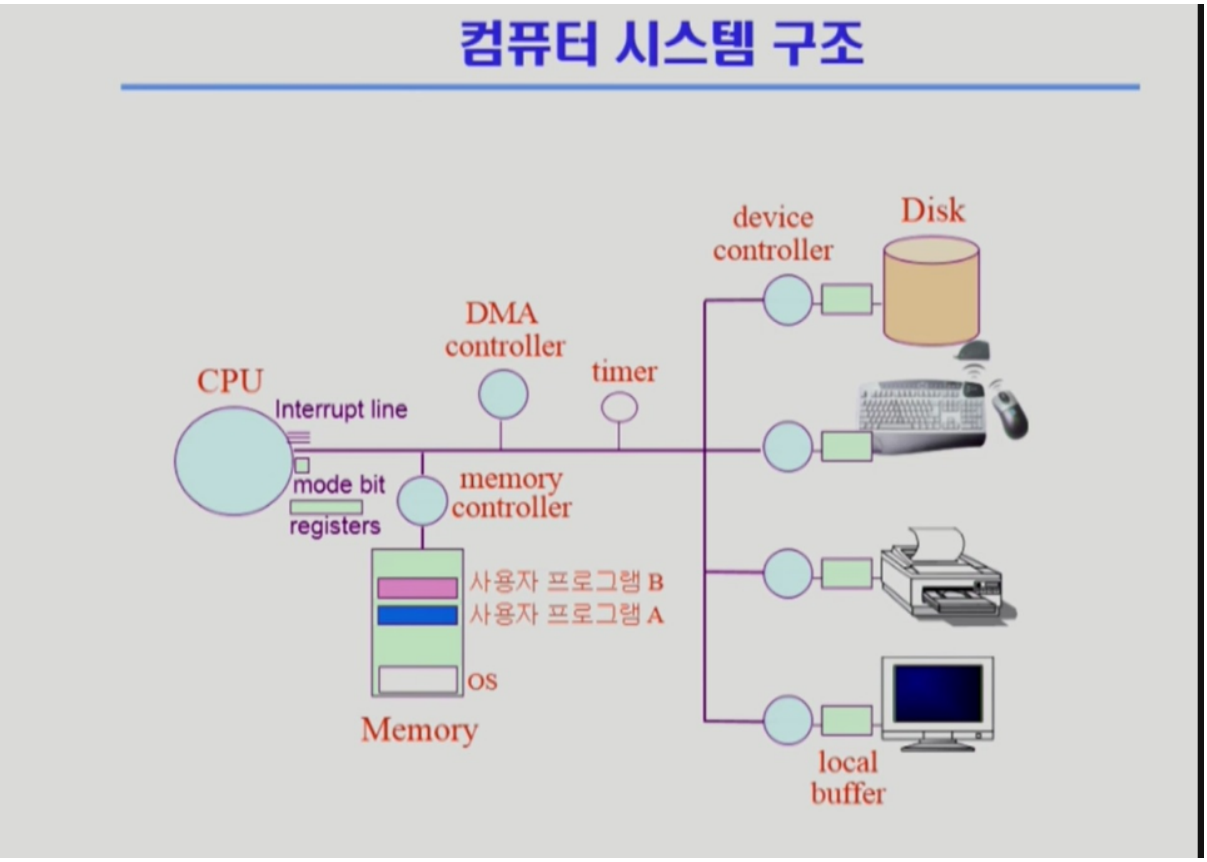
2. System Structure & Program Execution

▼ 목차

1. 컴퓨터 시스템 구조



레지스터 :
메모리보다 빠른 임시적인 저장 공간

모드 빗 :
CPU에서 실행되는 것이 운영체제인지 사용자 프로그램인지 구분해줌

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

mode bit을 통해 하드웨어적으로 두 가지 모드의 operation 지원

1 유저 모드 : 사용자 프로그램 실행
0 커널 모드 : OS 코드 수행

- 보안을 해칠 수 있는 중요한 명령어는 커널 모드에서만 수행 가능하도록 함
- 인터럽트나 익셉션이 발생하게 되면 하드웨어가 mode bit 을 0으로 바꿈
- 사용자 프로그램에게 CPU를 넘기기 전에 mode bit을 1로 세팅

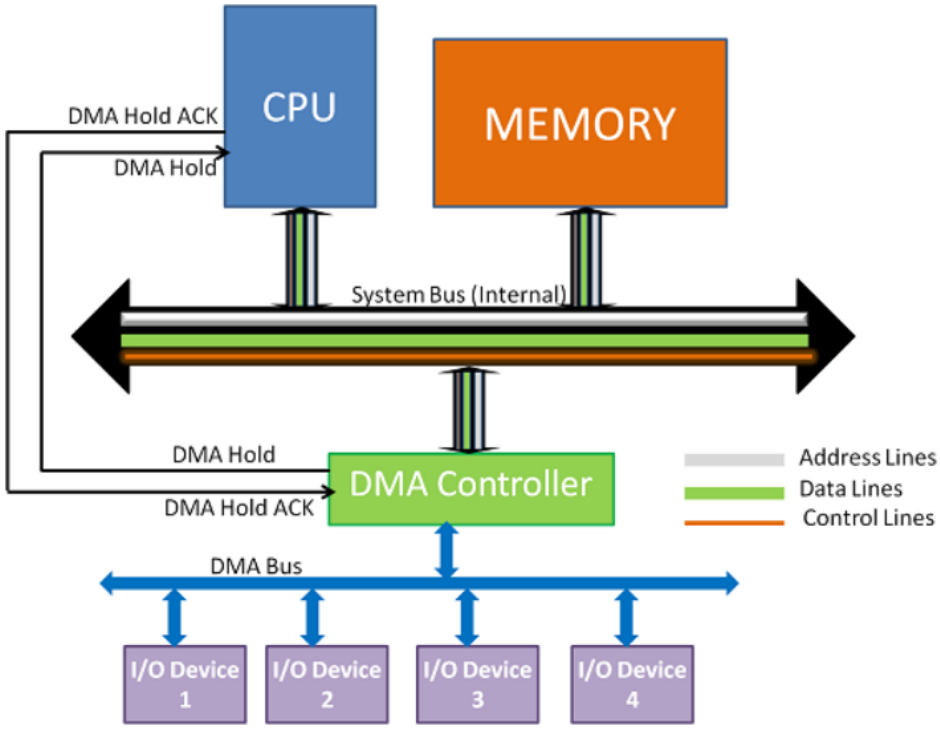
인터럽트 :
인터럽트는 현재 실행 중인 프로세스의 흐름을 중단하고 우선순위가 더 높은 작업에 대한 처리를 허용하는 메커니즘

인터럽트 라인 :
하드웨어적으로 구현되어 있으며, 각 인터럽트는 고유한 신호를 전송하여 중앙 처리 장치에게 특정 인터럽트 유형 및 우선순위를 알려줍니다. CPU는 인터럽트를 감지하고, 현재 실행 중인 작업을 중단하고 해당 인터럽트를 처리하는 인터럽트 서비스 루틴으로 이동하여 요청된 작업을 수행합니다.

DMA 컨트롤러 (Direct Memory Access) :
CPU가 IO 장치들로 너무 많은 인터럽트를 받게 하지 않기 위해 IO 장치들에서 요청이 들어온다면 CPU가 직접 그 내용들을 메모리에 카피하지 않고 DMA가 그 역할을 수행해준다.
카피 작업이 모두 완료되었으면 CPU에게 인터럽트를 한 번만 걸어준다.

빠른 입출력 장치를 메모리에 가까운 속도로 처리하기 위해 사용
CPU 중재 없이 device controller가 device의 buffer storage 내용을 메모리에 block 단위로 직접 전송

바이트 단위가 아니라 block 단위로 인터럽트를 발생시킴
→ 블록의 크기는 시스템에 따라 다르지만, 일반적으로 여러 섹터로 구성됩니다. 섹터는 디스크의 물리적인 단위로서 일반적으로 512바이트에서 4킬로바이트 크기



티이머 :

특정 프로그램이 CPU를 독점하는 것을 막기 위한 장치이자 시분할이 가능하게 해줌

할당된 시간이 지나게 되면 타이머 인터럽트가 발생. 예시로 만약 한 프로그램이 무한 루프에 빠졌을 경우 timer가 없다면 제어할 수 없게 됨.

디바이스 컨트롤러 :

IO 장치들을 관리하는 일종의 작은 CPU

---

2. 동기식 입출력과 비동기식 입출력

동기식 입출력

운영체제(커널)에 I/O 요청이 들어오면 입출력 작업이 완료된 후에야 CPU 제어권이 유저 프로그램에 넘어감

주로 입출력 처리에 의해 생성된 결과를 사용해야 하는 경우 활용

ex) 프로그램에서 필요한 데이터를 읽어와서 처리해야 되는 경우 => 동기식처리

구현 방법 1

I/O가 끝날 때까지 프로세스가 CPU를 가지고 있는 방법 (CPU 낭비)

1. I/O가 끝날 때까지 CPU를 낭비시킴
2. 매시점 하나의 I/O만 일어날 수 있음

구현 방법 2

I/O가 완료될 때까지 다른 프로그램에게 CPU를 주는 방법
작업중에 오래걸리는 작업은 뒤로 미뤄두고 먼저 끝나는 작업부터 처리하는 방법이다.

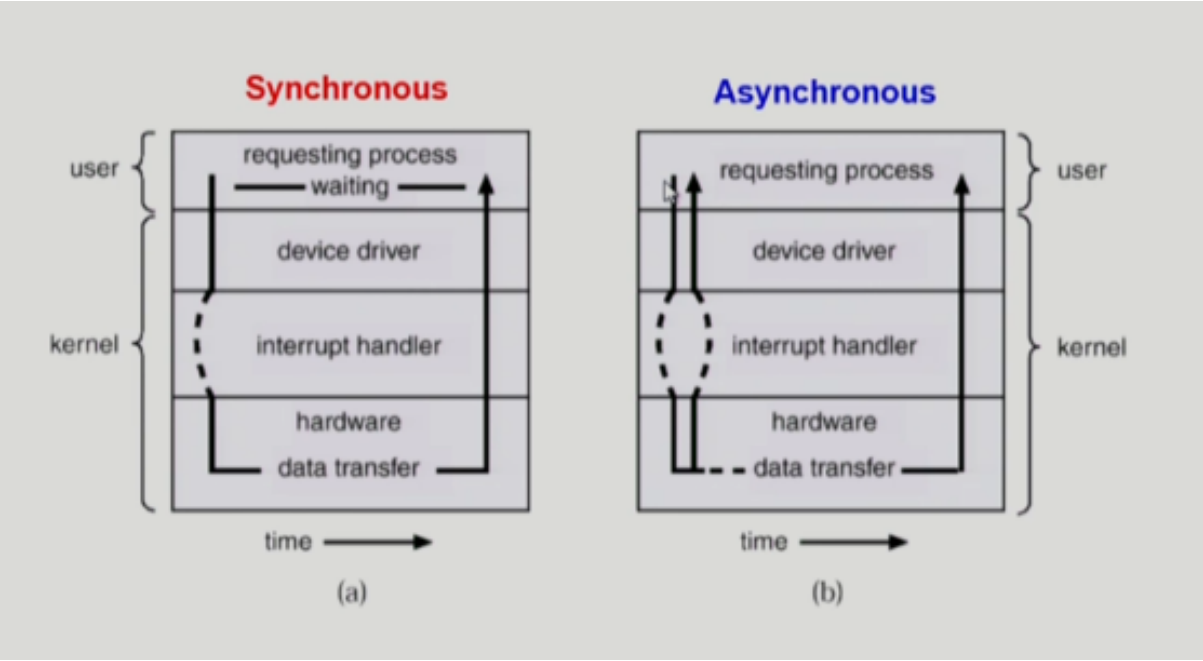
1. I/O가 완료될 때까지 해당 프로그램에게서 CPU를 빼앗음
2. I/O 처리를 기다리는 줄에 그 프로그램을 줄 세움
3. 다른 프로그램에게 CPU를 줌

여기서 오래 걸리는 작업을 뒤로 미뤄둔다는 의미는 아무것도 안하면서 뒤로 미루는게 아니다.
예를들어 A버퍼에는 물이 꽉차있고, B버퍼는 물이 빈 상태에서 이제 차기 시작한다.
그럼 A버퍼를 먼저 처리하면서 B버퍼에는 물이 차고있을거다. 그러면 A버퍼를 처리 한 후에 B버퍼를 처리한다는 소리다.

비동기식 입출력

운영체제(커널)에 I/O 요청만 해놓고 사용자 프로그램이 CPU 제어권을 즉시 돌려받아 다른 작업을 수행함.
입출력 작업이 끝날때까지 기다리지 않음. (비순차적)

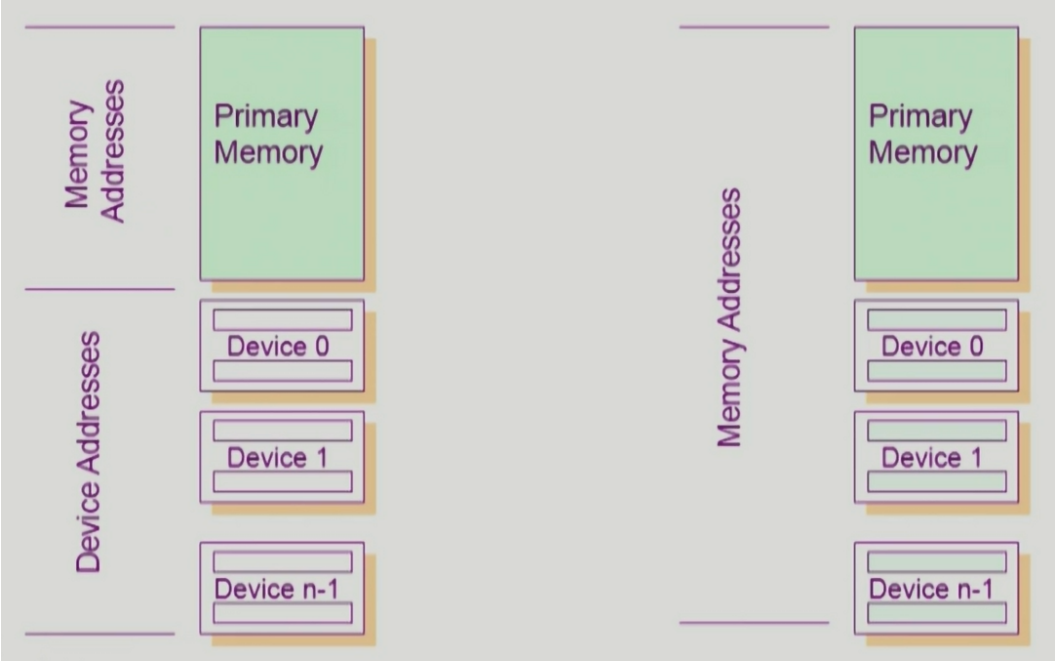
ex) 프로그램에서 필요한 데이터 없이 글을 쓰는 과정 -> 비동기식 처리



(동기식, 비동기식) 입출력 모두 I/O의 완료는 인터럽트로 알려준다.

---

3. I/O 처리 방법 2가지



(좌) **일반적인 I/O 처리 방법**
메모리상에 I/O 처리를 위한 별도의 instruction이 존재. 해당 instruction을 실행하여 I/O 처리
일반적으로 입출력 장치와 메모리 간에 데이터를 주고받기 위해 별도의 명령어나 포트를 사용

(우) **Memory Mapped I/O**
I/O 처리를 위한 instruction을 메모리주소의 연장선상에 놓는 방법

Memory mapped IO

메모리 맵드 I/O는 입출력 장치를 컴퓨터의 주소 공간에 매핑하는 방식을 의미합니다.
즉, 입출력 장치가 일반적인 메모리 주소와 동일한 주소 공간에 위치하게 됩니다. 이렇게 함으로써 입출력 장치는 일반적인 메모리에 저장된 데이터와 마찬가지로 CPU에 의해 접근될 수 있습니다.

1. 간편한 접근: 입출력 장치에 접근하기 위해 별도의 명령어나 포트를 알아야 하는 번거로움이 없습니다. 메모리 주소를 사용하여 입출력 장치에 접근할 수 있습니다.
2. 통일된 인터페이스: 입출력 장치와 메모리 간에 인터페이스가 통일되어 있으므로, 입출력 장치도 메모리처럼 취급할 수 있습니다. 따라서 소프트웨어 개발과 유지보수가 용이해집니다.

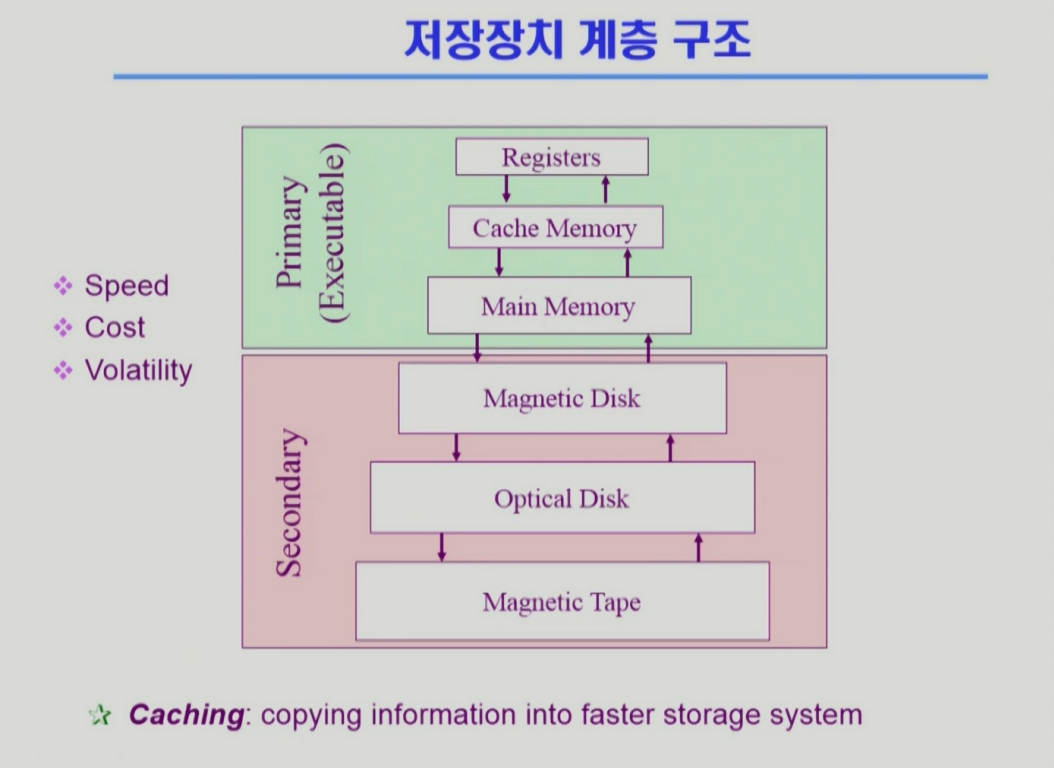
3. 메모리 기반 기능 활용 : 입출력 장치가 메모리로 간주되므로, 메모리의 다양한 기능을 입출력 장치에 활용할 수 있습니다. 예를 들어, DMA(Direct Memory Access)를 사용하여 입출력 데이터 전송을 가속화할 수 있습니다.

메모리 맵드 I/O를 사용하면 CPU는 입출력 장치를 다른 메모리 위치와 구별할 필요 없이 일관된 방식으로 접근할 수 있습니다. 입출력 장치는 일반적인 메모리 주소를 사용하여 데이터를 전송하고, CPU는 해당 주소를 읽거나 쓰는 것으로 입출력을 수행할 수 있습니다. 이를 통해 프로그래밍이 단순화되고 입출력 장치를 다루는 코드가 더욱 간결해집니다.

메모리 맵드 I/O는 입출력 장치를 다루는 소프트웨어 코드를 단순화하고 통일성을 부여하는 장점

---

4. 저장장치의 계층 구조



상위로 갈수록 처리 속도가 빠르고, 가격이 비싸고 용량이 작다는 특징이 있다.

primary 매체
CPU에서 직접 접근할 수 있다. CPU가 byte 단위로 읽고 실행할 수 있는 경우를 말한다.
휘발성 매체이기 때문에 전원이 꺼지면 저장된 데이터가 모두 지워진다.

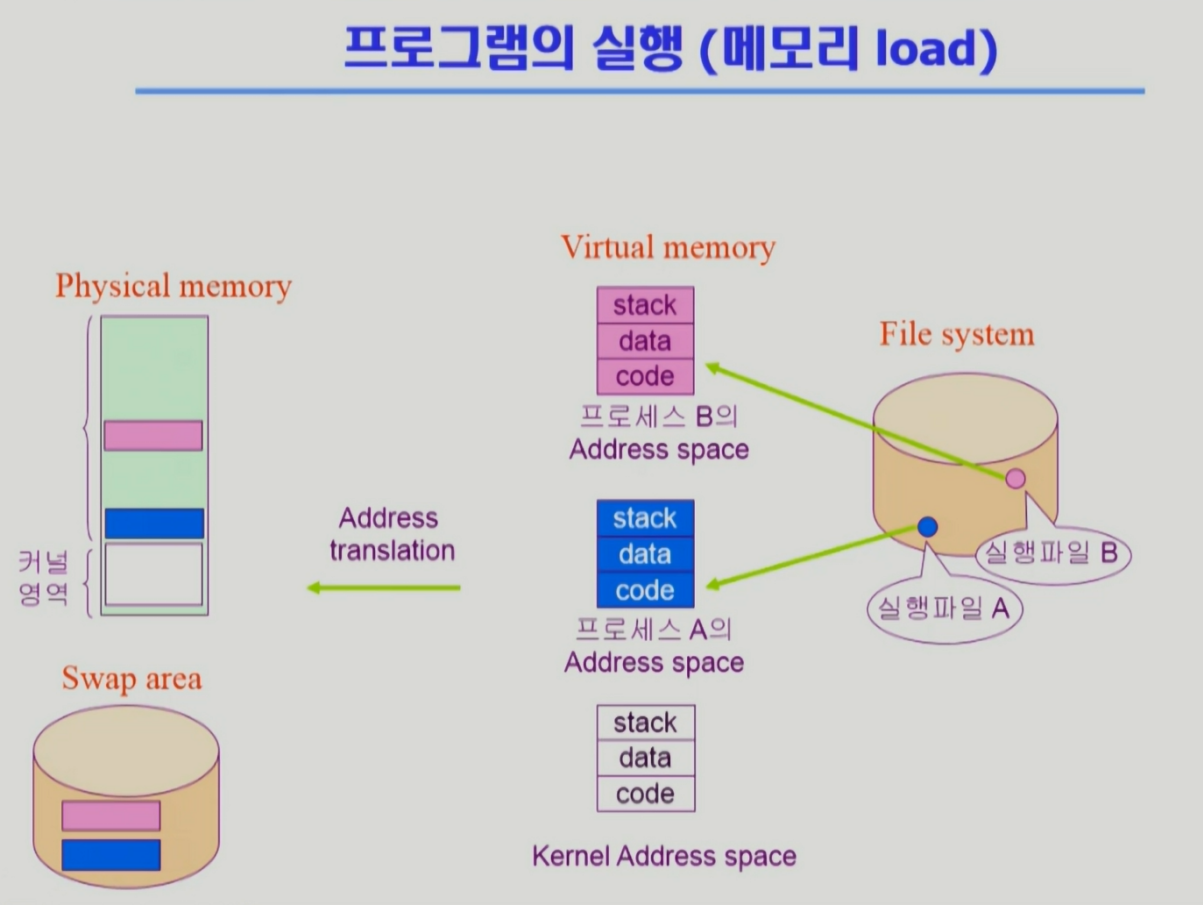
secondary 매체
CPU에서 직접 접근하여 실행할 수 없다.
비휘발성 매체여서 전원이 꺼져도 데이터가 남아있다.

Caching(캐싱)은 secondary 매체로부터 primary 매체인 Cache memory에 데이터를 복사해오는 것을 말한다. 데이터의 재사용을 목적으로 한다.

---

5. 메모리 주소

프로그램 실행



- 프로그램은 '실행파일' 형태로 하드 디스크의 파일 시스템에 저장되어 있다.
- 실행파일을 실행시키면 프로그램이 메모리상에 올라가서 '프로세스'가 되는데,
- 이때 메모리에 즉시 올라가는게 아니라 가상 메모리 단계를 거친다.

5-1. Virtual memory 주소 (Address Space)

프로그램을 실행하는 시점에 해당 프로그램만의 독자적인 Address space(메모리상의 주소 공간)를 생성하는 것을 말한다.

이 Address space는 stack, data, code 영역으로 구분된다.

- `stack` — 함수를 호출하거나 리턴할때 사용하는 영역

- `data` — 변수 등 프로그램이 사용하는 데이터들을 담고 있는 영역
- `code` — CPU에서 실행할 기계어 코드를 담고 있는 영역

Address space를 통째로 메모리에 올리는 것이 아니라, 당장 필요한 부분만을 메모리에 올린다.
또한 커널(운영체제)영역이 항상 메모리상에 상주하는것과는 달리
Address space는 프로그램 실행이 끝나면 메모리상에서 지워진다.

당장 필요하지 않는 부분은 디스크의 Swap area에 보관해둔다.

그렇기 때문에 프로그램마다 만들어지는 독자적인 주소 공간은 가상에만 존재하는 것이지 실제로 어디에 연속적으로 존재하지 않고 쪼개져서 존재하게 된다

> Swap area는 메모리의 연장선상에 위치하는 디스크 공간으로 메모리 용량을 보조하는 역할을 한다.
> 메모리와 마찬가지로 휘발성이다. 반면 디스크의 file system은 비휘발성 공간이다.

Address transition은 가상 메모리 상의 주소를 주 메모리 상의 주소로 변환하는 것을 말하며, 하드디스크의 특정 장치가 이 역할을 담당하고 있다.

5-2. Phsical memory 주소

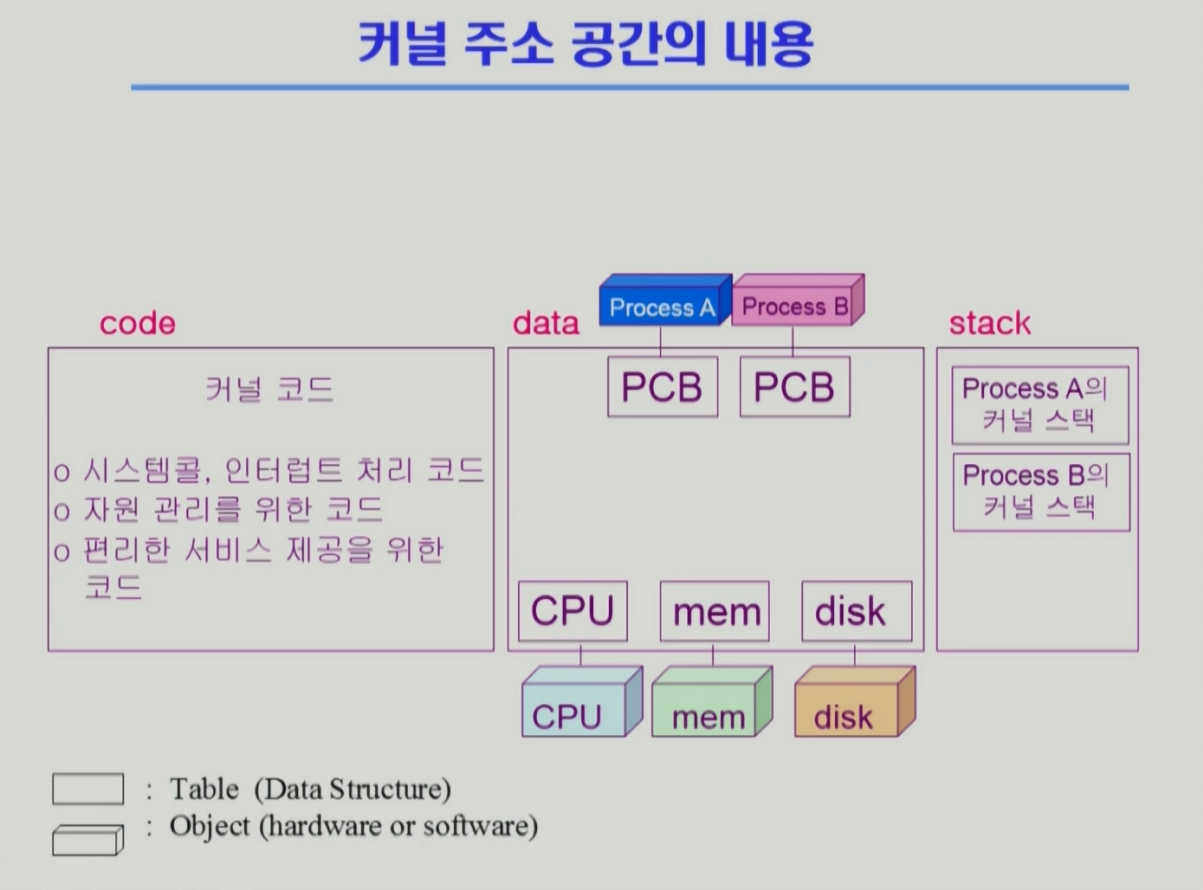
Address Space 중에서 당장 필요한 부분이 물리적 메모리에 올라감
→ 당장 필요하지 않은 것은 swap area로 보내짐

가상 메모리 값이 물리적 메모리로 올라가기 위해서 Address transition (주소 변환) 과정을 거친다
가상 메모리 주소 → 변환 → 물리적 주소

5-3. Kernal Address Space (커널 주소 공간)

커널

메모리에 상주하는 운영체제의 부분
좁은 의미의 운영체제. 즉 커널은 메모리에 상주하는 부분으로 운영체제의 핵심 역할을 함



- `code` — CPU 들이 수행할 기계어들이 모인 부분. 실행 파일에서의 코드들이 올라온다
  시스템콜, 인터럽트 처리 코드와 자원 관리를 위한 코드들이 올라옴
  인터럽트가 들어왔을 때 CPU 제어권이 운영체제에게 주어지기 때문에 각각의 인터럽트를 어떻게 처리해야 하는지를 명시해놓은 코드가 있다.
- `data` — 전역 변수, 정적 변수, 상수 등의 데이터가 저장된다.
  변수 등 프로그램이 사용하는 데이터들을 담고 있는 영역
  현재 수행 중인 프로세스의 상태를 저장하는 자료구조인 PCB가 저장
  또한 각각의 프로그램을 관리하기 위한 자료구조(Process Control Block) 역시 포함한다.
  → PCB _ 운영체제가 프로세스를 관리하기 위해 사용하는 데이터 구조
- `stack` — 함수 호출 및 임시 데이터 저장에 사용된다.
  커널 내에서 함수 호출이 발생할 때마다 해당 함수의 로컬 변수 및 함수 호출 정보가 스택에 저장
  운영체제의 커널 코드는 여러 사용자 프로그램의 요청에 따라 실행되기 때문에 사용자 프로그램마다 별도의 커널 스택을 가지고 있다.

---

6. 사용자 프로그램이 사용하는 함수

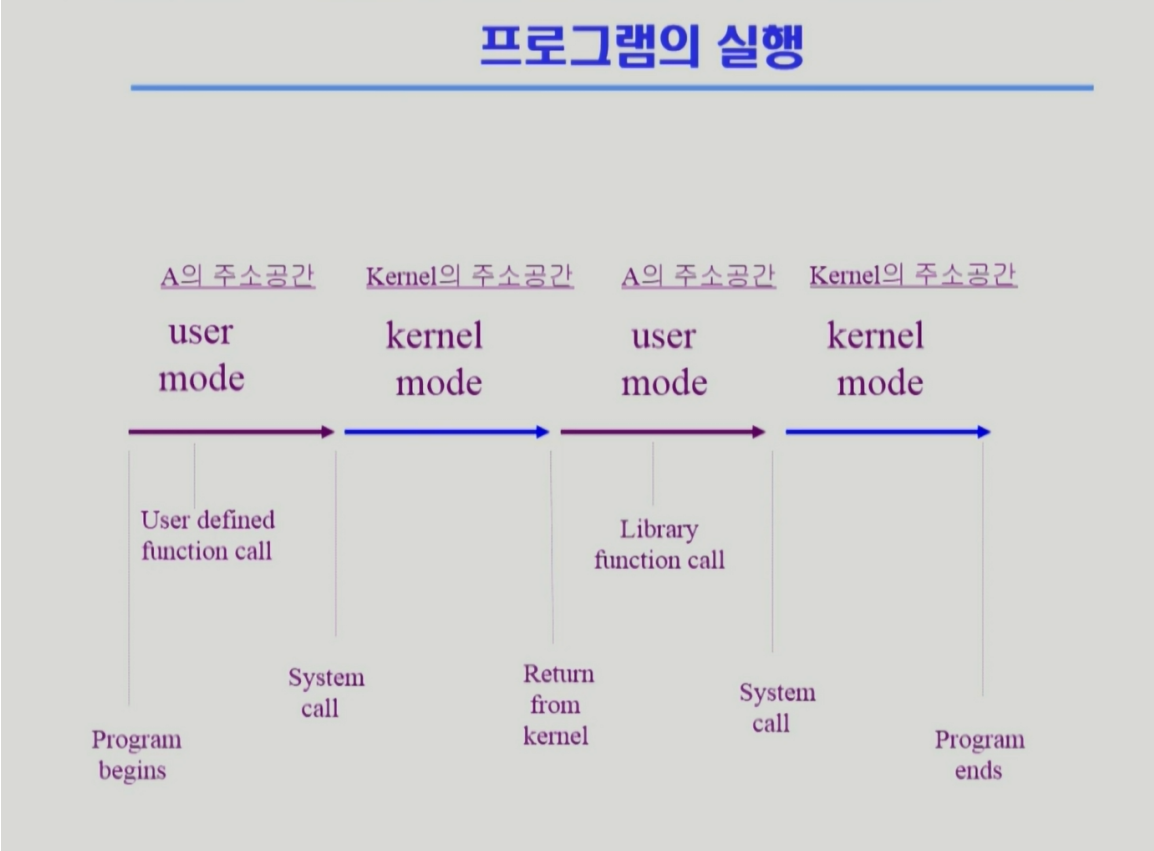
- 함수
  - **사용자 정의 함수** : 내가 프로그램에 정의한 함수
  - **라이브러리 함수** : 다른사람이 만들어 놓은 함수이지만 내 프로그램의 실행 파일에 포함되어 있는 함수
  - **커널 함수** : 운영체제 프로그램의 함수, 커널 함수의 호출 = System call

사용자 정의 함수든 라이브러리 함수든 컴파일해서 실행파일을 만들게 되면 내 프로그램 안에 들어있는 함수이기 때문에 언제든 자유롭게 실행할 수 있다.

반면 커널함수의 경우 내 프로그램의 함수가 아니라 커널코드에 포함된 함수이기 때문에 시스템 콜을 통해서 CPU 제어권을 넘겨야만 실행이 가능하다.

---

7. 프로그램 실행 단계



하나의 프로그램이 실행되는 동안 운영체제는 user mode ↔ kernel mode 전환을 반복한다.

프로그램이 CPU 제어권을 가지고 있으면 user mode라고 한다.
이때 syetem call을 하게 되면 CPU 제어권이 커널로 넘어가면서 kernel mode로 전환하게 되고 커널의 주소공간에 있는 커널함수를 실행한다.