System Structure & Program Execution 2

상태	완료
생성 일시	@2023년 6월 30일 오후 8:00
수업일	@2023년 6월 30일
유형	정리

이전 시간 복습…

CPU는 pc(프로그램 카운터)가 가리키고 있는 메모리 주소에서 인스트럭션을 읽어 실행하는 역할만 하는 것임

(메모리에 올라와 있는 애들만 읽을 수 있다는 뜻임) + 인터럽트 들어왔는지 확인하는 것도 함

인터럽트가 들어오면 CPU를 누가 쓰던 간 상관 없이 제어권이 운영체제에 넘어감 그러면 운영체제는 인터럽트 벡터에 가서 해당 인터럽트 처리에 맞는 것을 찾아 처리함

동기식 입출력과 비동기식 입출력

동기식 입출력 (Synchronous I/O)

동기식 입출력은 입출력 요청 후 입출력 작업이 완료된 후에야 제어가 사용자 프로그램에 넘어가는 것을 말한다.

예를 들어, 프로그램이 디스크에서 어떤 정보를 읽어 오라는 요청을 했을 때 디스크 입출력이 완료되기까지는 어느 정도의 시간이 소요될 것이다. 이때 2가지 방식으로 동기식 입출력을 구현할 수 있다.

[구현 방법 1]

- 입출력이 끝날 때까지 인터럽트를 기다렸다가, 끝나면 사용자 프로그램에게 CPU의 제어권을 넘긴다.
- 매 시점 하나의 입출력만 일어날 수 있다.

- CPU를 낭비한다는 문제가 있다.

[구현 방법 2]

- 입출력이 완료될 때까지 해당 프로그램(프로세스)에서 CPU의 제어권을 빼앗는다.
- 입출력 처리를 기다리는 줄에 그 프로그램을 세운다.
- 다른 프로그램에게 CPU의 제어권을 넘긴다.
- 동기식 입출력은 보통 이 방법으로 구현한다.

즉 입출력을 요구하는 프로세스의 CPU 제어권을 빼앗아 다른 입출력을 요구하는 프로세스에 CPU를 던져주고

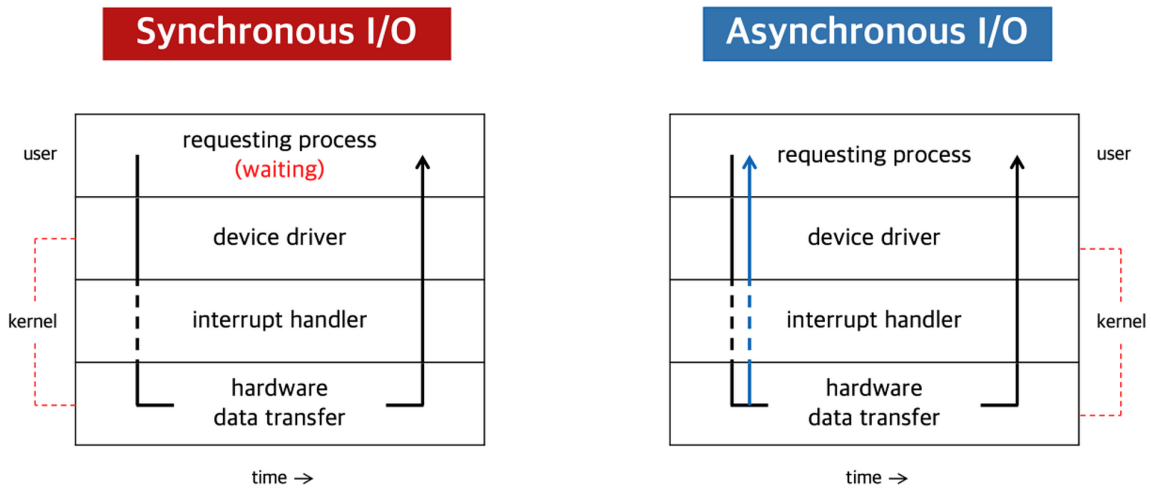
또 이 프로세스에 CPU 제어권을 빼앗아 다른 입출력을 요구하는 프로세스에 CPU를 던져주는 것을 반복한다면

CPU가 놀지 않고 일을 할 수 있다는 장점 및 I/O 장치들도 여러 개가 동시에 일을 할 수 있다. 그러다 I/O 장치에서 작업이 끝나면 I/O 장치 컨트롤러가 CPU에 인터럽트를 거는 것임 그래서 보통 이 방법을 많이 사용함

비동기식 입출력 (Asynchronous I/O)

비동기식 입출력은 입출력이 시작된 후, 입출력 작업이 끝나기를 기다리지 않고 CPU 제어권을 사용자 프로그램에게 즉시 넘기는 것을 말한다.

동기식 입출력 vs 비동기식 입출력



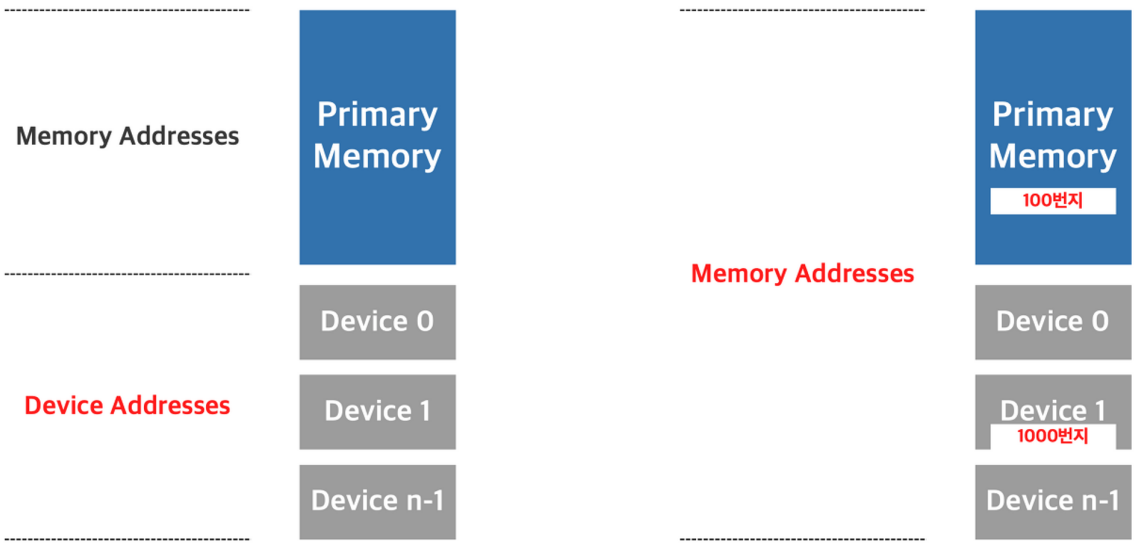
위 그림은 동기식 입출력과 비동기식 입출력의 비교해 보여 준다. 사용자가 입출력 요청을 하면 동기식 입출력에서는 먼저 운영체제의 커널로 CPU의 제어권이 넘어와서 입출력 처리와 관련된 커널의 코드가 수행된다. 이때 입출력을 호출한 프로세스의 상태를 Blocked 상태로 바꾸어 입출력이 완료될 때까지 CPU를 할당받지 못하도록 한다. 입출력이 완료되면 I/O 컨트롤러가 CPU에게 인터럽트를 발생시켜 입출력이 완료되었음을 알려주고, Blocked 상태인 해당 프로세스에게 CPU를 할당받을 수 있는 권한을 준다. 권한을 준다는 이야기는 CPU를 할당 받을 수 있는 줄에 서게 될 수 있다는 뜻이다.

반면 비동기식 입출력에서는 CPU의 제어권이 입출력을 요청한 프로세스에게 곧바로 다시 주어지며, 입출력 연산이 완료되는 것과 무관하게 처리 가능한 작업부터 처리한다.

한편 두 방식 모두 입출력 연산이 완료되면 인터럽트를 통해 CPU에게 알려준다.

서로 다른 입출력 명령어

I/O는 일반적인 I/O 방식과 Memory Mapped I/O 방식이 있다.



좌측이 일반적인 I/O고, 우측이 Memory Mapped I/O에 해당한다.

일반적인 I/O

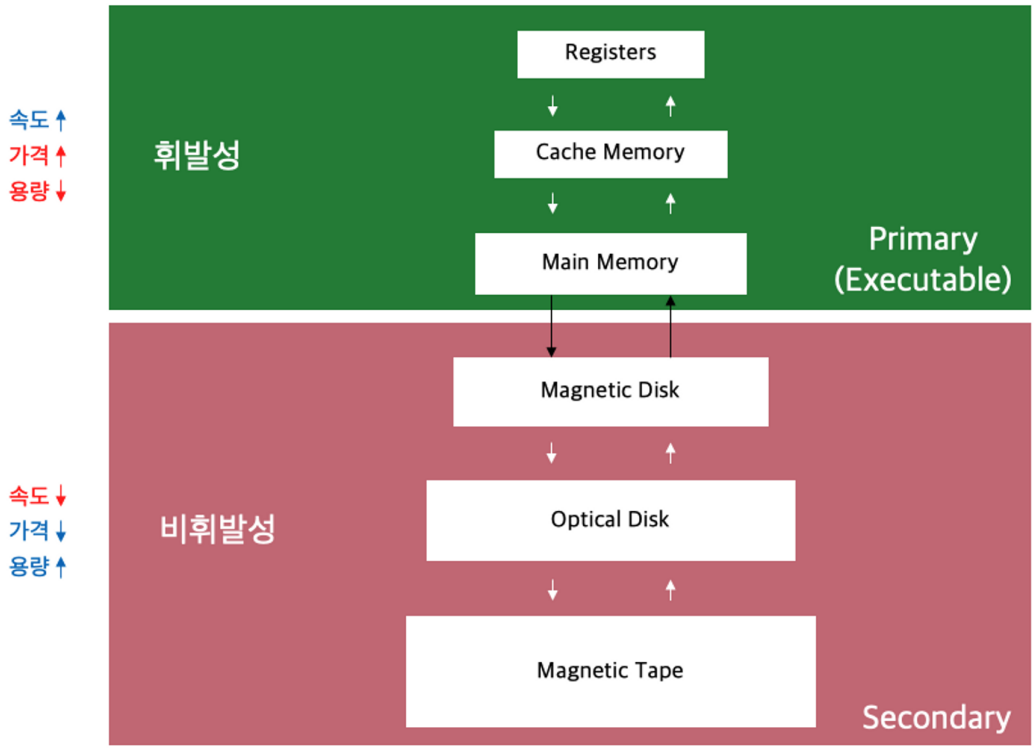
메모리에 접근하는 명령과 각 I/O 장치를 접근해야 하는 명령(인스트럭션)이 구분되어 있다.

Memory Mapped I/O

I/O 장치도 메모리 주소에 연장 주소를 붙여 접근할 수 있다.

예를 들어 100번지에 접근하는 것은 일반 메모리에 접근하는 것이지만, 1000번지에 해당하는 메모리에 접근하는 명령은 사실 I/O를 하는 명령임을 뜻한다.

저장 장치 계층 구조



위로 갈수록 속도가 빠르지만, 단위 공간 당 가격이 비싸고 용량이 적다. 반면 아래로 갈수록 단위 공간 당 가격이 싸고 용량이 많지만, 속도가 느려진다.

Primary Storage vs Secondary Storage

CPU는 바이트 단위로 접근 가능한 매체이어야 접근이 가능하다.

[Primary Storage]

CPU에서 직접 접근할 수 있는 매체를 말하며, Executable (실행 가능하다)라고 부른다. 즉, 해당 저장소는 바이트 단위로 CPU 접근이 가능하므로 Primary라고 부르며 전원이 꺼지면 데이터가 사라지는 휘발성 특징을 갖는다.

[Secondary Storage]

CPU가 직접 접근하지 못하는 매체를 말한다. 하드 디스크의 단위는 섹터 단위이므로 CPU가 접근하지 못해서 Secondary라고 부른다. 해당 저장소는 전원이 꺼져도 데이터가 보존되는 비휘발성 특징을 갖는다.

캐시 메모리

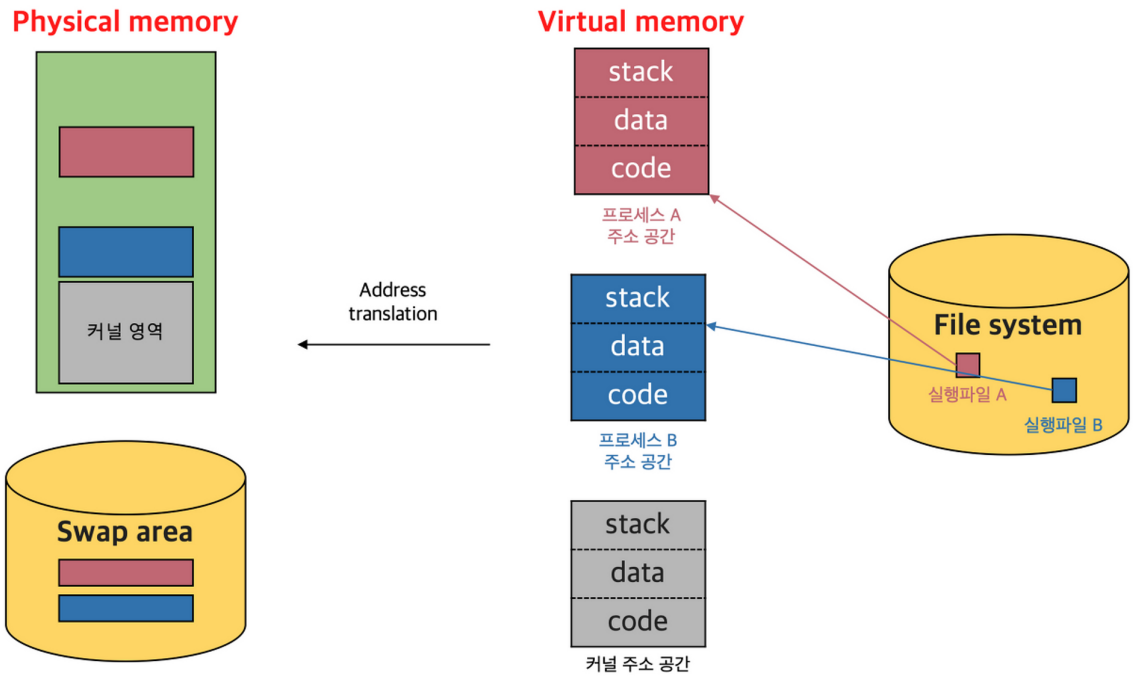
캐시 메모리는 CPU와 메인 메모리의 속도 차이를 완충하기 위해 존재한다. 용량이 적기 때문에 모든 것을 담아 둘 수는 없으나, 빈번히 사용되는 필요한 정보를 선별적으로 저장하여 재사용하는 캐싱 기법을 통해 시스템의 성능을 높일 수 있다.

caching: copying information into faster storage system

재사용의 목적이 있음. 처음에 읽을 때는 어쩔 수 없이 밑에서 위로 한번 가져다 놔야 함, 하지만 일단 한번 읽어 놓으면 같은 것을 두 번 요청할 때는 아래로 내려갈 필요가 없음. 시간 단축, 재사용성 ⬆️

프로그램이 어떻게 실행되는가?

프로그램의 실행 (메모리 load)



프로그램은 File System에 실행 파일 형태로 저장되어 있고, 이를 실행하면 메모리에 올라가 프로세스가 된다. 정확히 말하면, 물리적인 메모리에 프로그램이 바로 올라가는 것이 아니라 가상 메모리 단계를 추가로 거친다. 이때 독자적인 메모리 주소 공간이 형성되는데, 이 공간에는 Code, Data, Stack 영역이 있다.

Code

CPU에서 실행할 기계어 코드를 저장한다.

Data

전역 변수 등 프로그램이 사용하는 데이터를 저장한다.

Stack

함수가 호출될 때 호출된 함수의 수행을 마치고 복귀할 주소 및 데이터를 임시로 저장한다.

그 다음 가상 메모리(실제로 물리적인 메모리에 올라갈때는 쪼개지기 때문임 물리적 메모리/스왑영역으로)에서 물리적인 메모리로 프로그램이 올라가는데, 메모리 낭비를 방지하기 위해 프로그램 중 당장 실행에 필요한 부분만 올라가고, 그렇지 않은 부분은 디스크 중 메모리의 연장 공간으로 사용되는 스왑 영역에 내려 놓는다. 즉, 주소 공간을 쪼개서 어떤 부분은 메모리에 있고, 어떤 부분은 스왑 영역에 있게 된다.

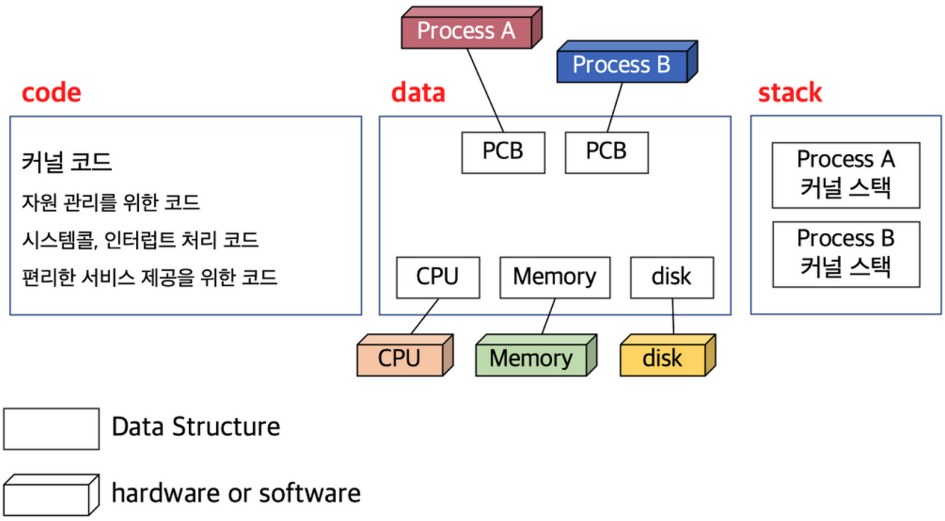
(즉 모든 프로그램이 이렇게 독자적인 공간(가상메모리로 분류된)을 가지고 있는데 이것을 물리 메모리에 올려 실행시키는 것인 필요한 부분만)

+운영체제 또한 하나의 프로그램이기 때문에 가상 메모리 단계에서 코드, 데이터, 스택 영역으로 나뉘게 됨

Disk 활용

위 그림에서 스왑 영역과 File System 두 디스크가 존재하는데, 스왑 영역은 메모리의 연장 공간으로서 휘발성 특징을 갖고 있고(전원 나가면 의미 없는 데이터임 메모리에 올라온 데이터와 그냥 사라지기 때문에), File System은 비휘발성 용도로 사용한다.

커널 주소 공간의 내용



Code

CPU, 메모리 등의 효율적으로 자원을 관리하기 위한 부분과 사용자에게 편리한 인터페이스를 제공하기 위한 부분이 주를 이루고 있다. 이 밖에도 커널의 코드는 시스템 콜 및 인터럽트를 처리하기 위한 부분을 포함한다.

(처리하기 위한 함수 같은 코드들이 들어있다고 봐야함)

Data

각종 자원을 관리하기 위한 자료 구조가 저장된다. CPU나 메모리와 같은 하드웨어 자원을 관리하기 위한 자료 구조뿐 아니라 프로세스를 관리하기 위한 자료 구조(PCB라고 함)도 커널의 데이터 영역에 유지된다.

커널의 데이터 영역 내에는 각 프로세스의 상태, CPU 사용 정보, 메모리 사용 정보 등을 유지하기 위한 PCB를 두고 있다.

Stack

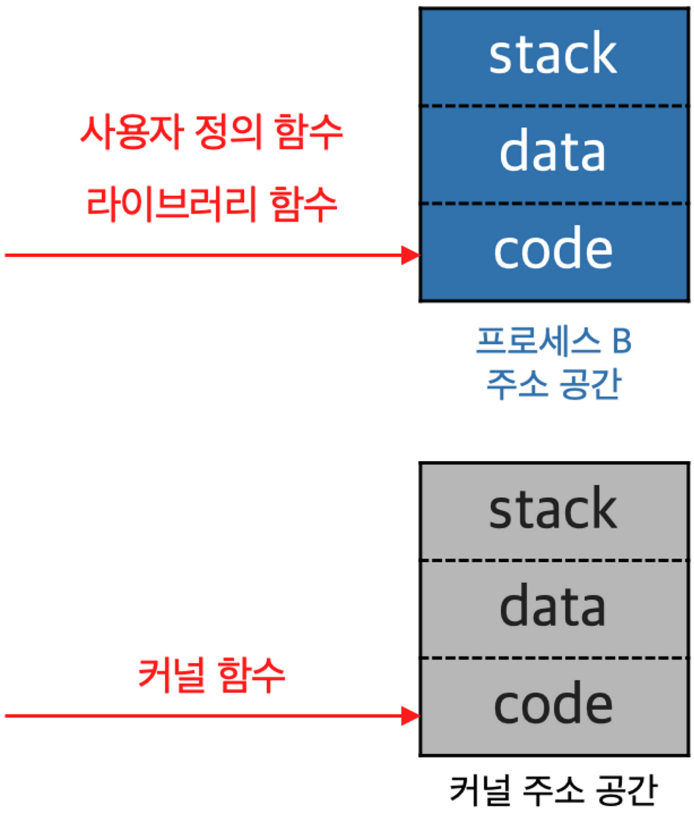
프로그램의 스택 영역과 마찬가지로 함수 호출 시의 복귀 주소를 저장하기 위한 용도로 사용된다. 하지만 커널의 스택은 일반 사용자 프로그램의 스택과 달리 현재 수행 중인 프로세스마다 별도의 스택을 두어 관리한다.

이는 일반 사용자 프로그램이 자기 주소 영역 내부의 함수를 호출하면 자신의 스택에 복귀 주소를 저장하지만, 프로세스가 특권 명령을 수행하려고 커널에 정의된 시스템 콜을 호출하고 시스템 콜 내부에서 다른 함수를 호출하는 경우에는 복귀 주소가 커널 내의 주소가 되므로 커널의 스택 영역에 저장되어야 하기 때문이다.

즉, 프로그램이 자기 자신의 코드 내에서 함수 호출 및 복귀 주소를 유지하기 위해서는 자기 주소 공간 내의 스택을 사용하고, 시스템 콜이나 인터럽트 등으로 운영 체제의 코드가 실행

되는 중에 함수 호출이 발생할 경우 커널 스택을 사용한다.

사용자 프로그램이 사용하는 함수



사용자 정의 함수

자신의 프로그램에서 정의한 함수를 말한다.

라이브러리 함수

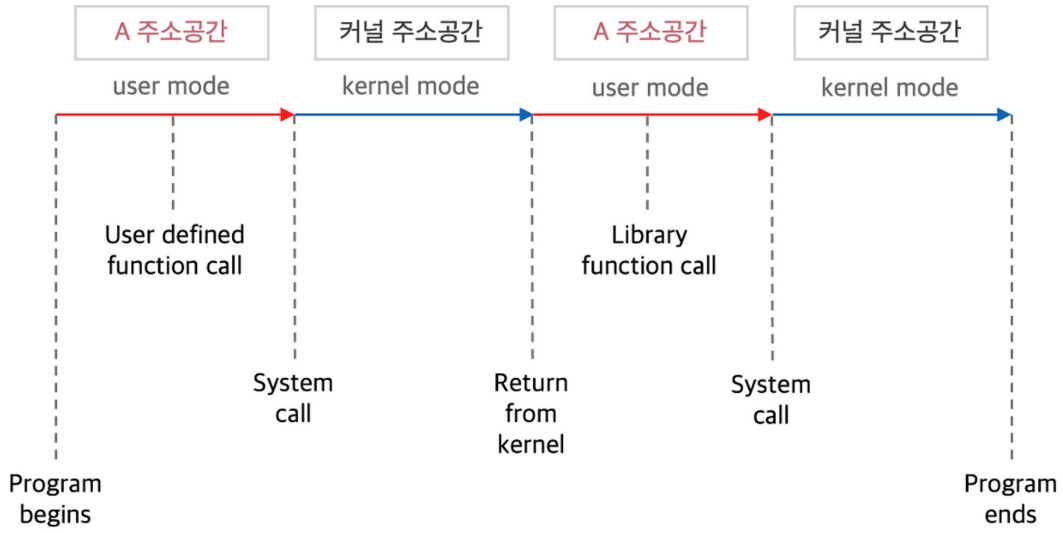
자신의 프로그램에서 정의하지 않고 가져다 사용한 함수를 말하며, 자신의 프로그램 실행 파일에 포함되어 있다.

커널 함수

운영 체제 프로그램의 함수를 말하며, 커널 함수의 호출은 곧 시스템 콜과 일맥상통한다. 당연하지만, 주소 점프를 할 수 없으므로 인터럽트 라인을 세팅하여 CPU에게 제어권을 넘겨

야 한다.

프로그램의 실행



위 그림은 사용자 프로그램 입장에서 본 실행 및 종료 과정이다.

프로세스 A가 CPU에서 실행되고 있다고 하면, 이는 자신의 주소 공간에 정의된 코드를 실행하는 것과 커널의 시스템 콜 함수를 실행하는 것으로 나누어 볼 수 있다. 전자를 사용자 모드에서의 실행 상태라고 하고, 후자를 커널 모드에서의 실행 상태라고 한다. 이때 한 가지 주의할 점은 비록 시스템 콜을 통해 실행되는 것이 프로세스 A의 코드가 아닌 운영 체제 커널의 코드이지만, 시스템 콜이 수행되는 동안 커널이 실행 상태에 있다고 하지 않고 프로세스 A가 커널 모드에서 실행 상태에 있다고 부른다. 프로세스 A 입장에서는 CPU를 운영 체제 커널에 빼앗긴 것으로 생각할 수 있지만, 사실 프로세스 A가 해야 할 일을 시스템 콜을 통해 커널이 대행하고 있는 것이기 때문이다.

프로그램은 즉 프로세스는 태어나서 죽을 때까지 유져모드, 커널모드를 계속 왔다 갔다가 죽는 것임(ex main 함수 끝나면) 현재 그림은 프로그램 A 관점에서만 보여지는 것임 CPU입장이라면 더 복잡함 인터럽트 오면 B프로그램 갔다가 커널 갔다가 등등