

리눅스 커널(운영체제) 강의노트 [1]



aeharvlee

Nov 18, 2018 · 25 min read

커널을 공부하는 마음가짐

리눅스 커널(**Linux Kernel**)을 한 사람이 전부 아는 것은 불가능하다. 커널 관련 두터운 원서를 75 ~ 80권 정도는 읽어야 “아, 한 번씩은 훑어봤다”라고 말할 수 있을 정도다. IBM과 같은 대형회사에서도 리눅스를 다루는 사람만 250명 정도가 있다고 한다. 250명의 사람이 방대한 커널에서 각자 분야를 맡아서 일을 처리한다.

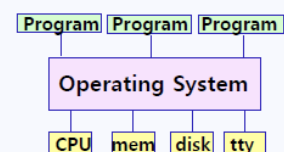
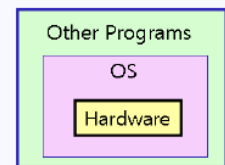
따라서 “모든 걸 다 알아야지!”라는 마음가짐 보다는 “커널과 운영체제가 어떤 식으로 동작하는지 개략적으로 이해해보자”라는 수준의 마음가짐을 가지는 것이 좋겠다.

1. 운영체제란?

운영체제(**Operating System**)란 하드웨어 자원들(**cpu, memory, disk, tty**)을 관리하고 프로그램들을 지원(**support**)해주는 것이다. (우측에 있는 그림을 살펴보면, Operating System의 아래에 하드웨어들이 있고 그 위로는 프로그램들이 있다.)

What is an Operating System?

- A program that acts as an **intermediary** between a **user** of a computer and the computer **hardware**.
- Operating system goals:
 - **Execute** user **programs**
 - Make solving user problems **easier**.
 - Make the computer system **convenient** to use.
 - Use hardware in an **efficient** manner.



- “Performance”
 - **Throughput** --- jobs / sec. system oriented
 - **Utilization** --- % of time busy system oriented
 - **Response time** --- sec / job user oriented
 - user oriented index conflicts with system oriented indices

다르게 표현하면, 하드웨어를 감추고 겉으로 다른 프로그램들을 지원해준다고 생각할 수 있다. 하드웨어를 감춘다는 건, 프로그램을 사용하는 사람이 편하게 쓸 수 있게 각종 기반 작업을 지원한다는 것으로 이해할 수 있다.

Note: 일반적으로 우리는 파워포인트나 워드를 쓸 때 프로그램이 **cpu**와 **memory**와 어떻게 소통하는지 등에 대해서 따로 신경쓰지 않는다. 이는 다 운영체제 덕분이다.

1.1 프로그램이란?

코딩을 해봤다면 `main()` 함수의 존재에 대해서 알 것이다. 프로그램이란 `main()` 함수를 포함하여 다른 다양한 기능들을 하는 함수들이 모인 존재라고 생각할 수 있다. 함수들이 적혀 있는 소스코드 파일을 컴파일 하면 프로그램이 된다는 것을 우리는 알고 있다.

`*.c` 파일을 컴파일해서 `a.out` 혹은 `*.exe` 등이 생성되고 우리는 이것을 프로그램이라 부른다.

여기서 한 가지 생각해볼 점이 있다. 왜 대부분의 프로그램은 분리되어 있는가?에 대한 점이다. Microsoft사를 예로 들어보자. Microsoft사는 Word, PowerPoint 등 많은 프로그램을 보유하고 있다. Office관련 프로그램을 통틀어 우리는 Microsoft Office라고 부르기도 하는데, 왜 Microsoft사에서는 왜 하나의 **Office** 프로그램이 아니라 여러 프로그램 (**Word, Powerpoint**)으로 분할해놨을까?

사업적인 목적도 있을 수 있겠지만 본질적으로는 하나의 커다란 프로그램으로 운영할 경우 발생하는 비효율성 때문이다. 거대한 프로그램은 실행할 때 부팅 시간도 오래 걸리고 메모리 사용에 있어서도 심각한 비효율성을 초래한다. 이런 여러가지 불편한 점이 있기에 여러가지 프로그램으로 분할해 놓은 것이다.

위와 같은 이유로 리눅스 운영체제 또한 **Kernel, Shell, Utility** 등 여러가지 프로그램으로 나뉘어져 있다.

Note: *는 임의의 문자를 뜻한다. 어떤 문자(패턴)와도 매칭될 수 있다. 와일드카드(wildcard)라고도 불리는 이 기호는 정규식(Regular Expression)을 공부한다면 제일 처음으로 배운다. 이 챕터에서 설명하는 `*.c`라고 적혀 있는 부분

은 `helloWorld.c`, `goodByeWorld.c` 와 같이 나타낼 수도 있다. 또한 `a.out` 의 경우 유닉스 계열 운영체제에서 사용되는 실행파일과 목적파일의 형식이고 `.exe` 의 경우 윈도우 운영체제에서 사용되는 실행파일 확장자다. 참고로 `a.out` 이라는 이름은 어셈블러 출력(assembly output)을 줄인 말이다.

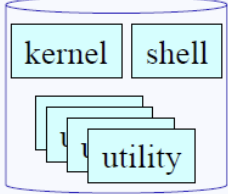
1.2 커널(Kernel)이란?


Kernel은 본질적으로 프로그램이다. 우리가 흔히 아는 `main()` 으로 시작하는 프로그램 말이다.

하지만 다른 모든 프로그램과는 다르게 커널만이 가지고 있는 특별한 점이 있다. 그것은 바로 **‘Memory Resident’**라는 점이다. 메모리에 항상 상주해 있는 것이 바로 커널이다.

Terminology

- *kernel*
 - memory resident part of OS, just plain C program.
- *utility*
 - **command**, disk resident part of OS (loaded on demand),
 - “program”(next page) → under /bin
- *shell*
 - A special utility. It’s mission is Job Control
 - reads keyboard input & execute command (interpreter)
 - UNIX interface to user. Shell prompt & command.
- *file*
 - “named collection of information”
 - “sequence of bytes”, no other restrictions (eg record, block)
 - I/O devices are treated as files (*special files*, try `ls /dev`)
- *standard file*
 - *standard output*: screen
 - *standard error*: error message
 - *standard input*: keyboard





NEAOSS MC2.0 2-2-1-I v1.0
CC-BY 2.0 KR. © Korea OSS Promotion Forum, NIPA

5

커널이 아닌 다른 프로그램들은 메모리에 있어도 되고 없어도 된다. **‘Disk Resident’**라고 표현한다. 필요할 때마다 그 때 그 때 메모리에 로딩해서 사용하면 된다는 의미다.

커널은 **‘Memory Resident’**특징을 제외하곤 아주 평범한 **C program**이다. 커널을 제외한 다른 프로그램들을 우리는 **Utility**라고 하는데 위에서 언급했듯 **disk resident**하다. 항상 현 주소가 **disk**라는 의미다. 사용자가 필요할 때 요청을 하면 그 때 메모리에 올라오는 (로딩되는) 것이다. 그런 의미에서 **Utility**를 우리는 **Command**라고도 칭한다. **Utility**와 **Command**를 동의어로 생각하고 공부해보자.

1.3 셸(Shell)이란?

우리 디스크에는 수십 수백개의 프로그램들이 존재한다. 이 프로그램들이 언제 메모리에 로딩되고 언제 메모리에서 해제되는지 누가 관리해줄까? 관리해주는 프로그램이 꼭 필요하지 않을까?

위와 같은 필요에 의해 탄생한 것이 셸이다. 많은 프로그램들의 메모리 교통 정리를 해주는 역할을 한다. 유틸리티 중 하나로 셸의 1차적인 임무는 **'Job control'**이다. Utility, Command, Job을 동의어로 생각하고 공부해보자.

1.4 파일(file)이란? (유닉스에 한정함)

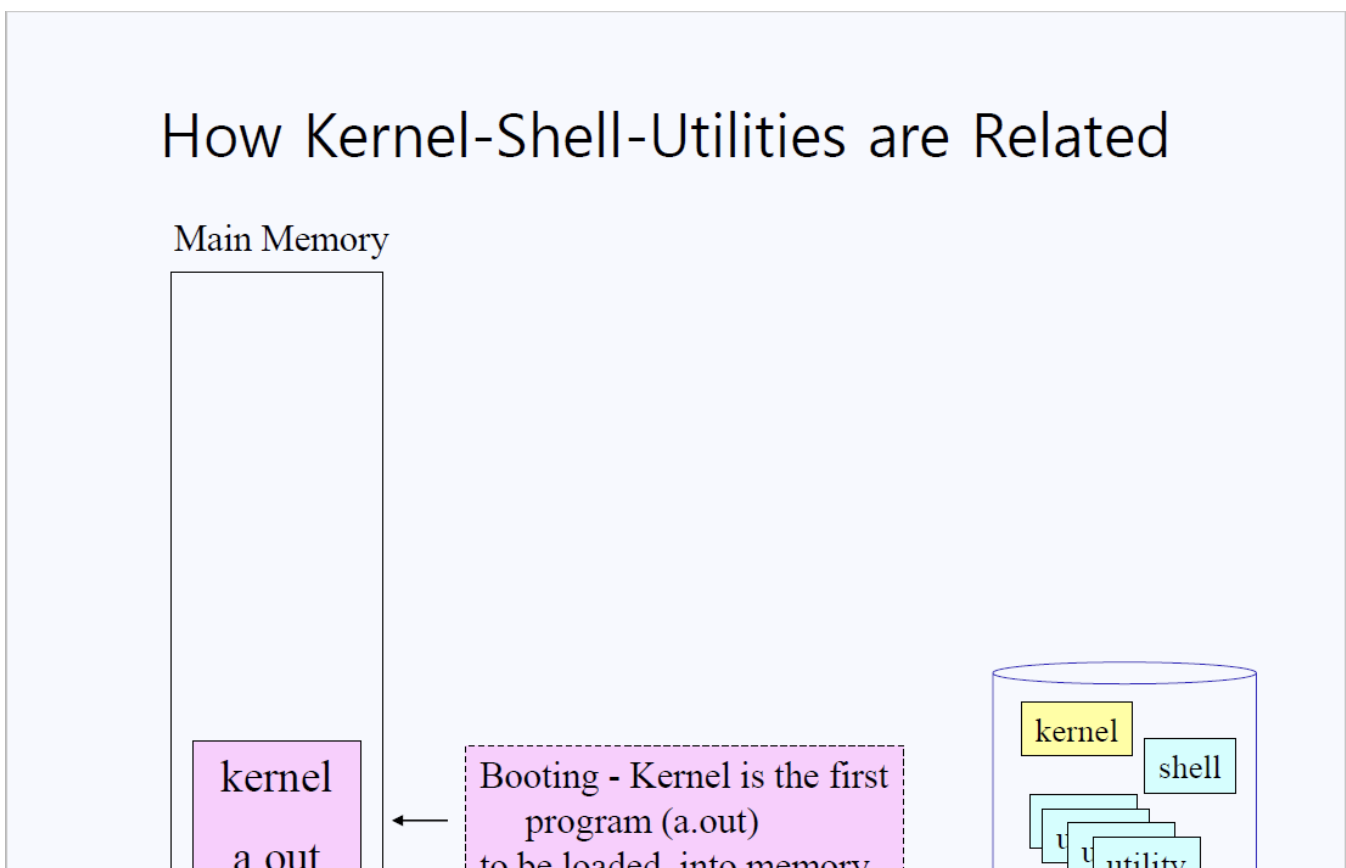
유닉스(Unix)에서 파일은 **sequence of bytes**를 의미한다. 말 그대로 바이트들의 배열이란 뜻이다. 모든 함수, 명령어들은 결국 기계어로 해석하면 0과 1의 나열에 불과하다.

특히 유닉스, 리눅스에서는 **I/O device**도 file로 취급한다. 당장 이해가 안되겠지만, 입출력 기계들(하드 디스크, USB, 키보드 등)을 파일로 취급한다고 알고 있으면 된다.

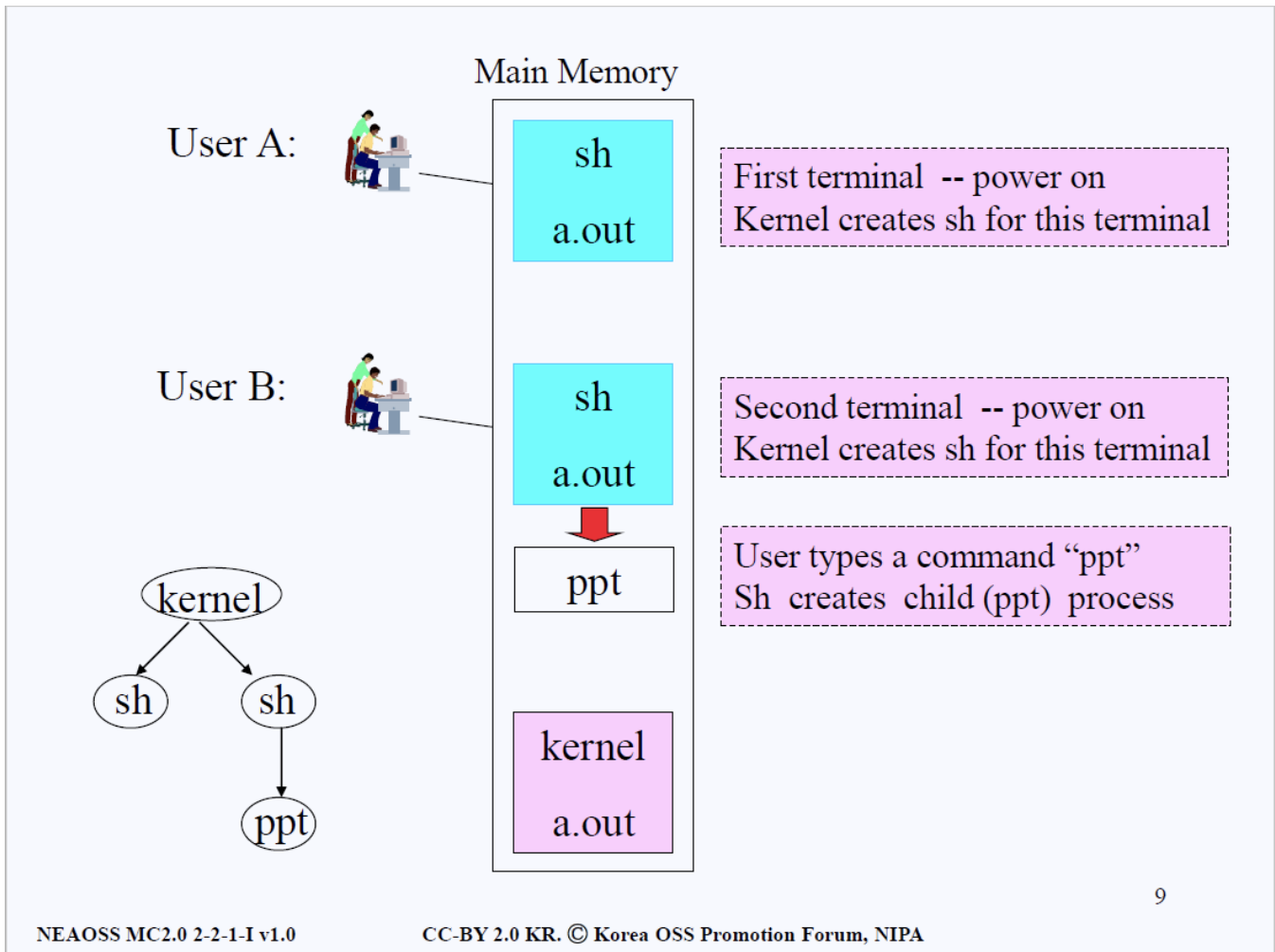
Note: 유닉스, 리눅스에서 입출력 기계들은 `/dev/hd0`, `/dev/hd1`, `/dev/tty2` 등 파일로서 다뤄지고 입출력 기계와 1:1로 대응된다.

1.5 커널과 셸, 그리고 유틸리티의 관계

아래의 이미지를 살펴보면, 좌측에 메모리 그리고 우측에 디스크가 있다. 커널과 셸, 그리고 유틸리티들이 디스크와 메모리에서 어떻게 작동하는지 살펴보자.



맨 처음 시스템을 부팅하면 제일 먼저 메인 메모리에 Kernel(a.out)이 올라온다. 커널 실행파일이 메모리에 로딩된다는 말이다. 리눅스는 멀티 유저 시스템으로 하나의 시스템에 다양한 유저가 들어온다는 사실을 상기하고 아래의 그림을 살펴보자.



유저가 터미널에 전원을 키면 그 터미널 위에서 셸(shell) 이란 프로그램이 메인 메모리에 올라온다. 그 후 셸은 유저가 키보드로 커맨드를 입력하기를 기다린다. 유저가 커맨드를 입력하면, 셸은 커맨드에 대응하는 유틸리티를 디스크로부터 가져와서 실행시킨다. 각 유저로부터 전원이 들어올 때마다 이 3개의 프로그램(커널, 셸, 유틸리티)의 관계가 형성됨을 알 수 있다.

그런데 키보드로 커맨드를 입력하길 기다리고 그에 대응하는 유틸리티를 디스크로부터 가져와 실행시킨다는 것이 무슨 말일까? 이 말의 의미를 이해하기 위해서는 리눅스가 멀티유저 시스템이라는 점과 **CLI(Command Line Interface)**라는 점을 상기해야 한다. 더 자세한 설명은 계속해서 나오니 따라가보자.

위 내용들을 다시 한 번 정리하자면 이렇다. 커널은 운영체제이며 항상 메모리에 상주해 있다. 나머지 프로그램들은 전부 유틸리티이며 디스크에 상주한다.

유틸리티는 항상 디스크에 있다가 필요할 때마다 메모리에 올라오고 사용하지 않을 때는 다시 내려간다. 유틸리티를 우리는 커맨드라고도 칭하며 커맨드들의 교통정리를 하는 것이 셸의 역할이다.

셸한테 우리가 ppt라고 커맨드를 입력하면 셸은 child process로 ppt를 생성한다. child process도 추후에 나오는 개념이니 일단 넘어가보자.

1.6 터미널은 뭐고 콘솔은 뭘까?

이 파트는 KLDP의 게시물을 참조하여 작성하였다.

콘솔은 전통적으로 보면 계기판과 입력장치의 모음과 비슷한 것들의 집합으로 컴퓨터를 조작하기 위한 조작부라고 생각하면 된다.

옛날 대형컴퓨터는 serial(rs232, 422, 485) 를 이용하여 터미널이라는 장치와 연결하여 조작하였고, 아직도 몇몇 은행에서는 이러한 장비들을 볼 수 있다. 터미널이라는 장비는 CRT와 키보드로 구성되어 있으며 Teraterm, 넷텀, 하이퍼터미널과 같은 프로그램들은 모두 터미널 에뮬레이터라고 불렸다. 이들은 일반 PC를 터미널 대용으로 사용할 수 있도록 해주는 역할을 한다. 터미널 을 통해 콘솔 과 통신한다 라고 생각해보자.

이해를 쉽게하기 위해 터미널은 콘솔의 부분집합이라고 생각하는 것도 좋겠다. 아래 커널 컴파일의 도움말에 기술된 내용을 보면, 리눅스에서 콘솔과 터미널은 거의 동일한 의미로 사용되고 있다고 볼 수 있기 때문이다.

If you say Y here, you will get support for terminal devices with display and keyboard devices. These are called “virtual” because you can run several virtual terminals (also called virtual consoles) on one physical terminal.

2. 운영체제 비교, 리눅스 vs 윈도우

2.1 자원의 소모에 대하여

운영체제에는 종류가 굉장히 많지만 가장 대표적이라고 일컬어지는 리눅스와 윈도우를 한 번 비교해보려 한다. 윈도우는 개인 컴퓨터(Personal Computer)에 사용되는 운영체제다. 개인 컴퓨터에 쓰인다는 것은 나홀로 사용자라는 것, 즉 싱글유저 시스템이라는 의미다. 리눅스가 멀티유저 시스템으로 만들어진 것과는 상반된다.

싱글유저 시스템일 때는 보안문제에 대해서 신경을 별로 안 써도 되지만, 멀티유저 시스템일 때는 보안에 특히 신경을 써야한다. 이유는 간단하다. 멀티유저 시스템은 그 안에 내 파일 뿐만 아니라 다른 사람의 파일 또한 존재하기 때문이다. 내가 다른 사람의 파일을 멋대로 읽고 쓸 수 있다면? 즉 삭제하고 변경할 수 있다면? 이라는 물음을 던지면 쉽게 이해가 갈 것이다.

또 하나의 이슈는 메모리 관리다. 멀티유저 시스템에서는 한정된 자원을 모두가 효율적으로 이용해야 하기 때문에 메모리 관리가 상당히 중요하다. 반면 윈도우와 같은 싱글유저 시스템에서는 멀티유저 시스템만큼 메모리 관리에 신경을 덜 써도 된다.

싱글유저 시스템: 내 돈 내고 구입한 컴퓨터(장비)인데, 내가 아니면 누가 써? 나만 쓸거야!

위의 한 문장이 싱글유저 시스템의 기본 철학이다. 그래서 PC(Personal Computer)는 일찌감치 윈도우 운영체제와 함께 결합된다. 윈도우는 GUI(Graphic User Interface)를 채택하고 있기 때문이다. 일반 사용자 입장에서 화면에 일일이 키보드로 명령어를 입력하는 것보다 마우스 몇 번 클릭해서 프로그램을 실행하고 관리하는 것이 훨씬 편하기 때문이다.

반면 리눅스, 유닉스 등의 멀티유저 시스템은 **CLI(Command Line Interface)**를 채택한다. 윈도우에서는 내가 사용할 수 있는 유틸리티(프로그램, 커맨드와 동의어)가 아이콘으로 보기 좋게 화면에 표시되는데, 리눅스는 **man**이라는 명령어로 내가 사용할 수 있는 커맨드가 무엇인지 알아내거나 사전에 알고 있어야 원활한 사용이 가능하다.

Why protection is necessary in Linux

Multi-user vs Single-user System

- | Multi-user | vs | Single-user System |
|---|----|---|
| <ul style="list-style-type: none"> • Linux • Protection - Yes • Resource – minimize <ul style="list-style-type: none"> – Text mode (CUI)* – Total silent • eg <i>vi</i> | | <ul style="list-style-type: none"> • Windows • Protection - Little • Resource – use them all <ul style="list-style-type: none"> – Window, GUI – Shows everything <ul style="list-style-type: none"> • History • State • Command • Option • eg <i>word</i> |

* CUI: Character User Interface

* GUI: Graphical User Interface

11

NEAOSS MC2.0 2-2-3-I v1.0

CC-BY 2.0 KR. © Korea OSS Promotion Forum, NIPA

윈도우는 사용자에게 편리한 인터페이스를 제공하는 대신, 동일 작업 대비 훨씬 많은 자원을 요구한다. 리눅스는 사용자에게 다소 불편한 인터페이스를 제공하는 대신, 화면에 표시되는 Char(1 단위당 1바이트)만큼의 자원을 쓰는 등 효율적인 자원 사용을 가능케 한다.

2.2 보안 이슈에 대하여

윈도우와는 다르게 리눅스에서는 보안이 매우 중요하다. 위에서 언급한 상황을 다시 생각해보자. 만일 한 프로세스가 다른 프로세스의 정보를 함부로 I/O 하려고 하면 어떨까? 즉, 다른 프로세스의 파일을 삭제하거나 내용을 바꿀 수 있다면? 그것만큼 끔찍한 일도 없을 것이다. 어제 힘들게 작성한 보고서를 자신의 가장 친한 친구가 실수로 삭제했다고만 생각해도 화가 치미는데, 모르는 사람이 그런다면... 상상하기도 싫다.

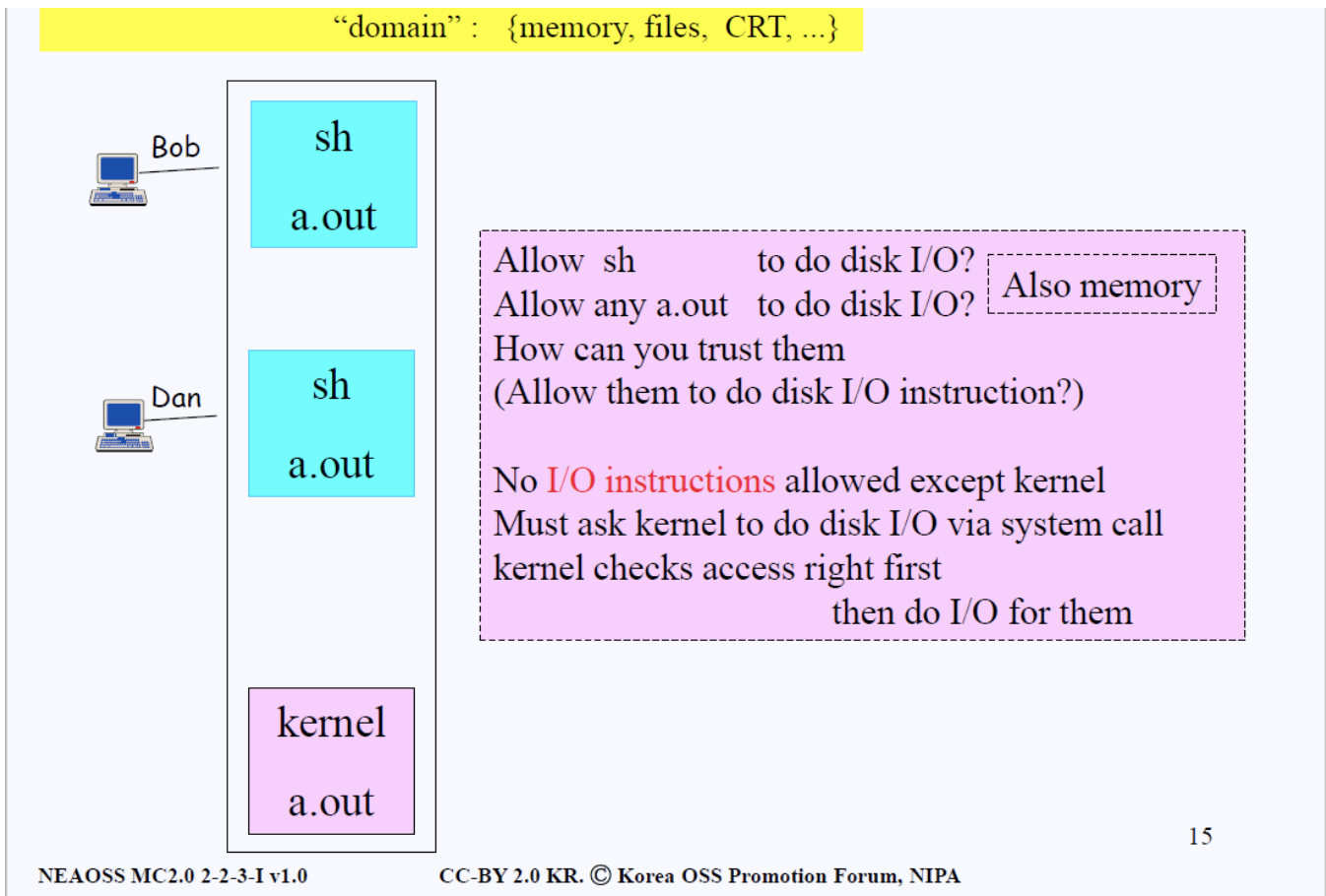
이러한 보안문제를 해결하기 위해 리눅스에서는 사전방지(Prevent)하기 위해 많은 노력을 하게된다. 사후처리가 아닌 사전방지(Prevent)하는 이유 또한 간단하다. 내 파일이 삭제된 이후에 복구하는 뻔짓보다는 사전에 그런 행위를 못하게 막는 것이 훨씬 효율적이고 철학적으로도 옳다.

우리가 시스템을 설계하는 입장이라고 생각해보자. 알다시피 한 시스템에서 CPU는 모든 프로그램이 공통적으로 사용하는 공용자원이다. 메모리에 존재하는 프로그램은 여러가지고 하나의 프로그램만이 한 번의 순간에 CPU를 온전히 차지한다.

CPU는 한 순간에 하나의 연산밖에 못한다. 즉, 한 순간에 CPU를 사용하는 건 오직 하나의 프로그램 뿐이다. 여기서 순간이란 건, 기술력에 따라서 달라지는데 현재는 1초에 40억 번의 연산이 가능한 CPU들이 등장하고 있으니, 최근 기술로서 한 순간을 해석해보자면 나노세컨드 정도는 될 것으로 생각된다.

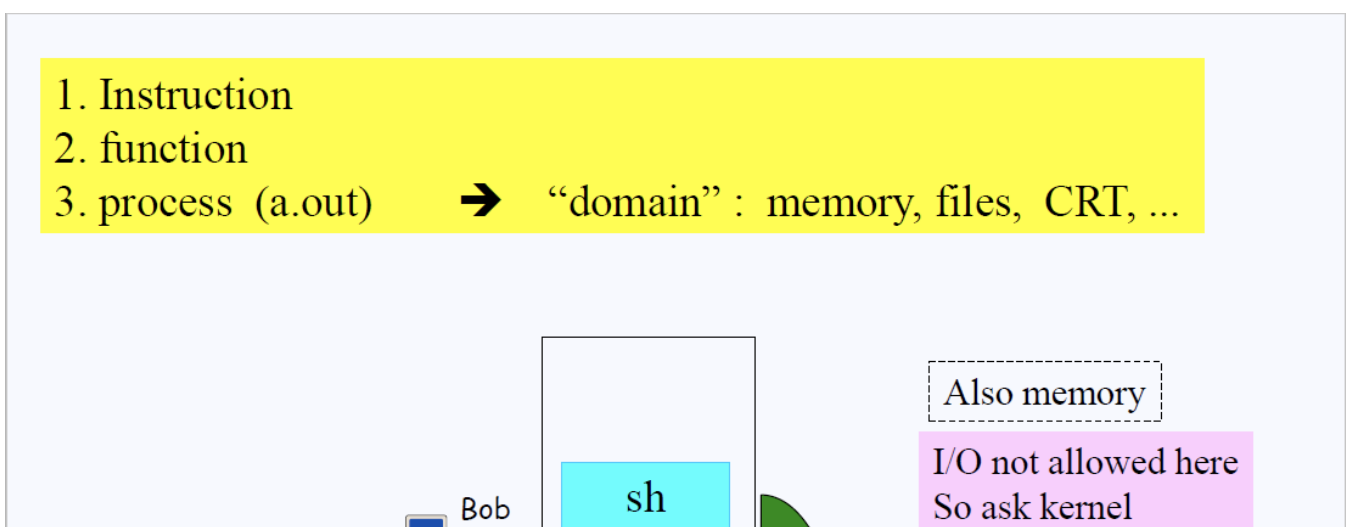
위의 내용들을 바탕으로 아래의 그림을 살펴보자.

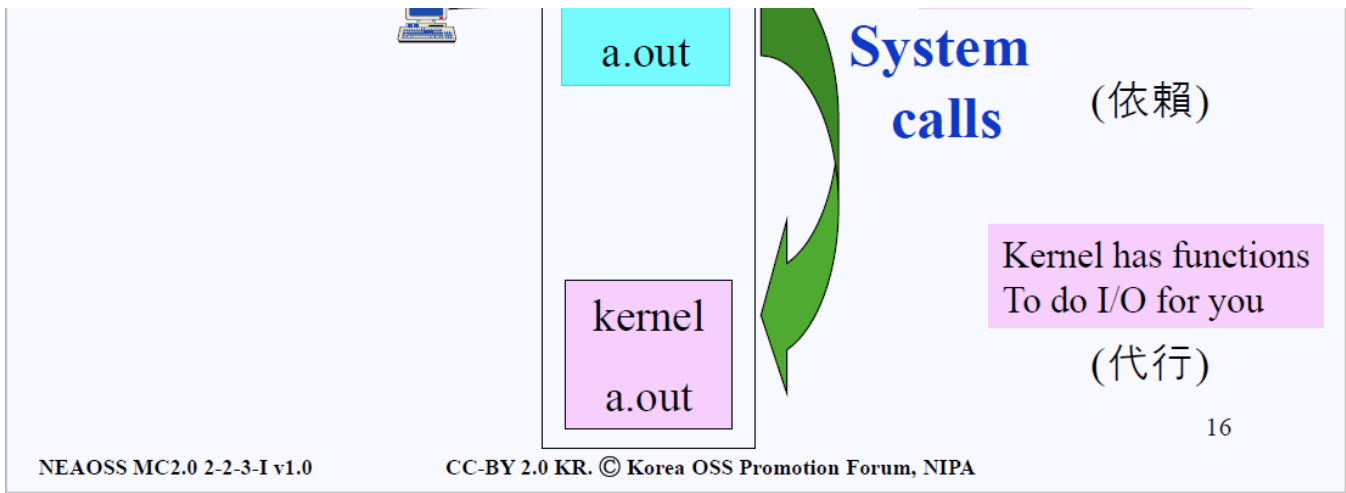
Note: 1초에 40억번의 연산이 가능하다는 말은 1초에 CPU는 40억번의 순간을 함축하고 있다는 뜻이다. CPU는 1초에 40억번의 순간이 존재하고 그 순간마다 연산(모든 프로그램을 관리)하기 때문에 우리가 워드, 게임, 피피티, 인터넷 익스플로러, 구글 크롬 등을 한꺼번에 켜놓고 작업을 해도 끊김 없이 동시에 처리가 되는 것처럼 보인다.



CPU가 Bob의 터미널과 연결된 셀에게 양도되었다고 생각하자. 즉 시스템이 Bob의 터미널과 연결된 셀에 CPU를 넘겨준 것이다. 셀도 하나의 프로그램이므로, 셀에 버그가 생겼다고 가정해보자. 본래 셀은 Bob이 접근 가능한 디스크 영역, 메모리 영역에만 데이터를 읽고 써야하는데, 디스크의 Sector Address를 잘못 계산해서 Dan의 영역에 데이터를 썼다고 가정해보자.

즉 Dan이 저장해 놓은 파일들이 다 엉망이 됐다는 뜻이다. 시스템은 이미 CPU를 Bob과 연결되어 있는 셀에게 양도를 해줬고... 이 셀은 잘못을 저질렀다. 해당 프로그램이 오류로 엉뚱한 곳에 데이터를 읽거나 쓰는 상황이 발생했는데 어떻게 이러한 상황을 예방할 수 있을까?



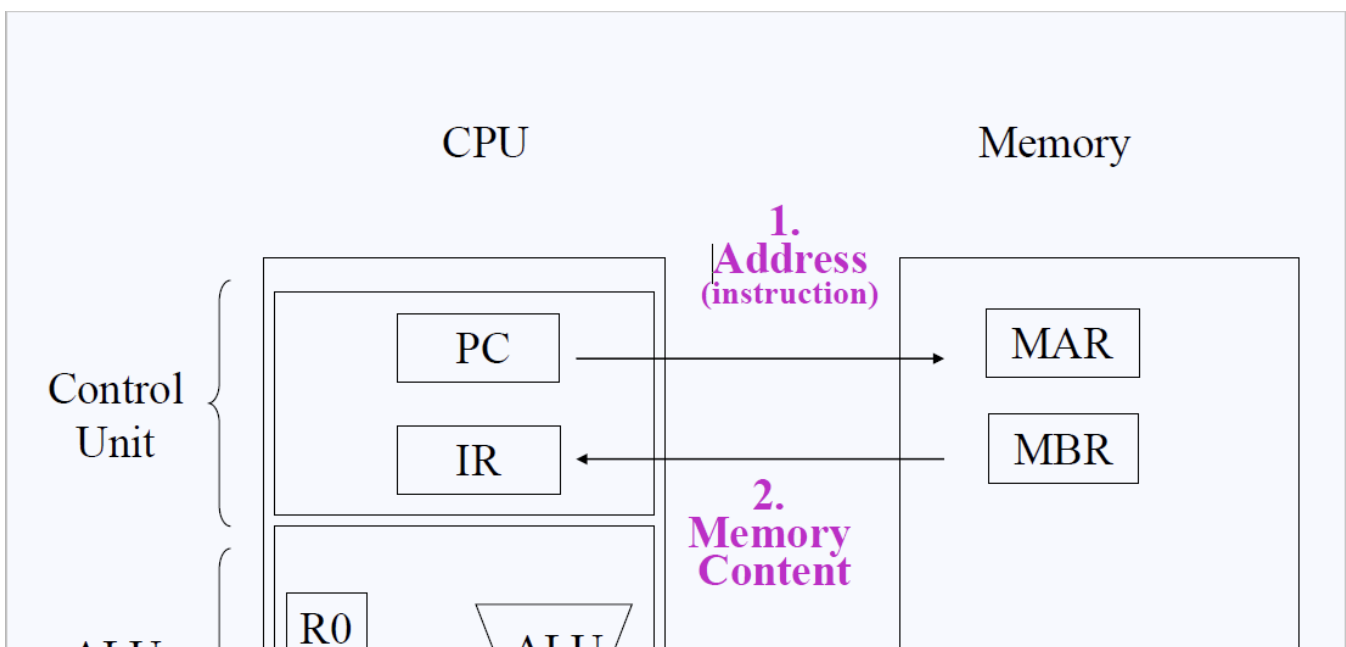


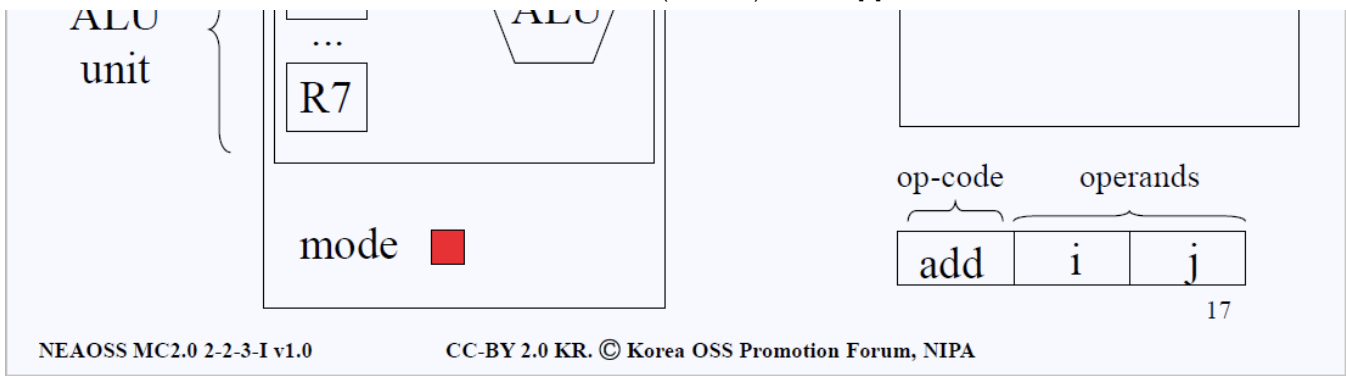
위의 그림에도 써져 있지만 리눅스 시스템 설계자가 고안한 해답은 바로 바로 **"I/O instruction 하지마!"**다. 만약 셸을 포함한 다른 일반 프로그램이 I/O를 수행하는 순간 CPU 사용권한을 바로 뺏겨버린다.

셸(sh)입장에서는 한편으론 억울할 수 있다."나는 **CPU**를 사용할 수 있고, 내 역할은 **I/O**를 하는건데 그럼 뭘 하라는 거야?"라고 셸은 반박한다. 자신의 임무를 수행하려 했더니 억울하게 CPU를 뺏겨버리니 말이다.

그래서 리눅스 시스템 설계자는"그럼 니가 **I/O** 할 때 커널에게 부탁을 해. **I/O** 하게 해달라고."라고 셸에게 말한다. 즉, 현재 리눅스 시스템의 작동은 **I/O instruction**을 할 때는 커널이 갖고 있는 **function**에 부탁을 하는 방식으로 되어있다. 이 부탁하는 과정을 **System call**이라 한다.

커널은 해당 입출력 명령이 합법적인 것인지 검사한 후에 I/O를 대신 진행해준다. 이런 특별한 구조를 구축하기 위해 윈도우나 개인 컴퓨터에는 없는 개념을 하드웨어에 도입한다. 아래의 그림을 살펴보자.

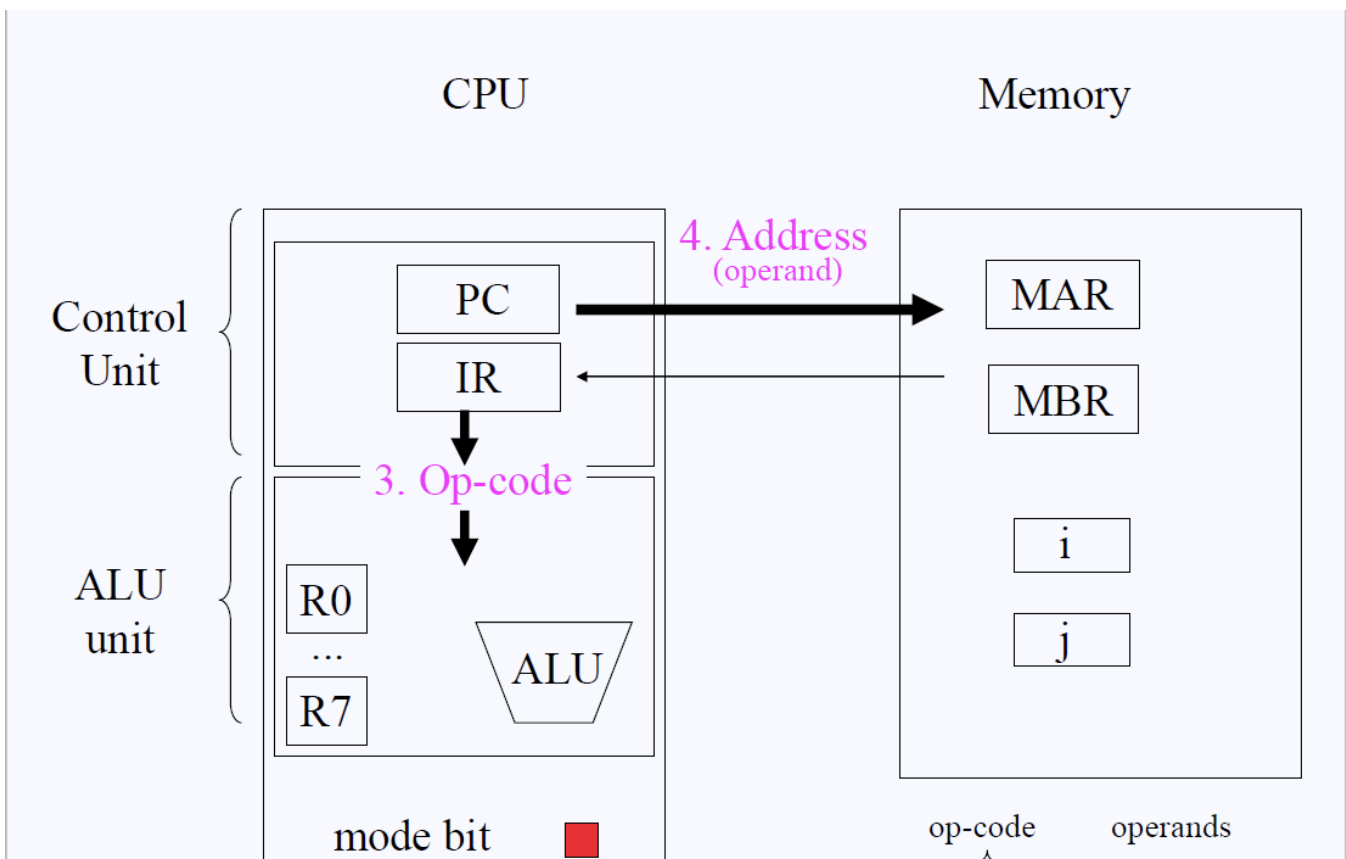




바로 **CPU**에 하나의 **Binary Bit**를 도입한 것이다. 위 그림에 빨간색 네모칸이 바로 그 bit(비트)다. 이 bit을 우리는 **mode bit**라 칭한다. 비트라는 건 알다시피 2개의 데이터(0과 1)만을 저장할 수 있다. 0은 유저모드를 의미하고 1은 커널모드를 뜻한다. 우선 우리가 메모리와 CPU에 대해서 알고 있는 개념을 다시 한 번 정리해보자. 그림과 함께 설명을 보자.

CPU의 Control Unit 파트에 **PC**는 **Program Counter**라고 하는 특수 레지스터다. **PC**에서는 이번에 수행해야 할 **instruction(명령)**의 주소를 메모리로 보낸다. **MAR**은 **Memory Address Register**이며 **Address Bus**라고도 이해할 수 있는데, **MAR**이 읽어 들인 주소에 해당하는 데이터를 **MBR(Memory Buffer Register)**에 담아 **IR(Instruction Register)**에 보낸다.

IR로 들어온 **Instruction(명령)**은 위 그림의 우측 하단과 같은 구조를 지닌다. **op-code**에는 수행해야 할 명령이 적혀있고 그 옆에는 **operands**라고 하는 명령 인수들이 적혀 있다. **i**와 **j** 주소에 있는 값을 더하라는 의미 정도로 해석할 수 있다.

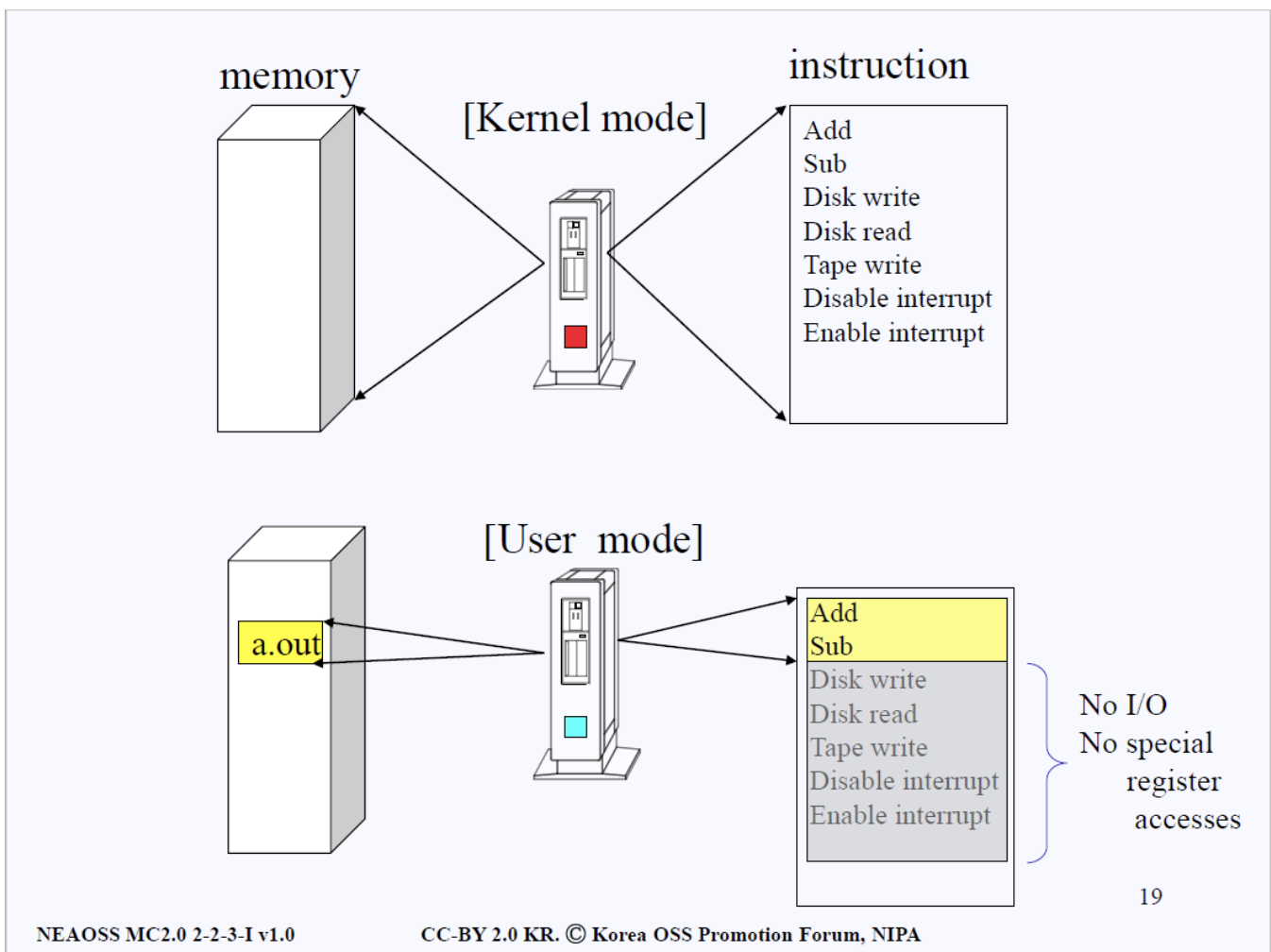




Op-code를 읽어들이며 명령어를 처리하는데, **i**와 **j**에 해당하는 메모리 주소에 담겨 있는 값을 알아내기 위해 또 다시 메모리에 접근해서 연산을 진행한다.

2.2.1 mode bit(모드 비트) 추가설명

CPU에는 한 바이너리 비트인 모드 비트라는 개념이 존재하고, 그것은 **0** 또는 **1**의 값을 갖는다. 커널 모드에서 동작하느냐 유저 모드에서 동작하느냐를 정해주는 이 바이너리 비트를 우리는 일단 **0**이면 유저모드, **1**이면 커널모드로 동작한다고 생각해 보자.




만약 모드 비트가 커널모드(**1**)로 되어 있다면 CPU는 어떠한 영역의 메모리라도 접근할 수 있다. 커널모드가 아닌 유저모드로(**0**)로 되어 있다면 모든 메모리에 접근(Access)은 불가능하고 자신의 Address Space만 접근가능하다.

또한 커널모드(**1**)일 경우에는 모든 **instruction**이나 **op-code**를 수행(execute)할 수 있지만, 유저모드(**0**)라면 **I/O instruction**이나 **special register accesses**와 관련된

instruction은 금지된다. 즉 유저모드에서는 타인에게 큰 영향을 줄 수 있는 **instruction**은 모두 거부되는 것이다.

Note: **special register accesses**란 스택 포인터(**SP**)나 프로그램 카운터(**PC**)와 같은 특별한 레지스터에 대해 값을 읽는 등의 행위를 말한다.

CPU "mode bit"



CPU mode	memory	op-code
kernel (全體 權限)	<i>access <u>any</u> (all)</i>	<i>execute <u>any</u></i>
user (部分 權限)	<i>mine only (local)</i>	<i>restricted *</i>

* privileged op-code (特殊 命令語?)

- No I/O instruction
- No special-register access
- any thing that can harm others (惡影響) 20

NEAOSS MC2.0 2-2-3-I v1.0

CC-BY 2.0 KR. © Korea OSS Promotion Forum, NIPA

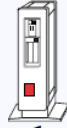
2.3 커널모드와 유저모드

인천 공항 보안 검색대를 생각해 보자. 일반 사람들이라면 보안 검색대를 반드시 통과해야겠지만, 자신이 대통령이나 국가원수에 해당하는 중책이고 매우 바쁜일이 있다면 보안 검색대를 거치지 않고 바로 공항을 나설 수 있다.

CPU는 항상 메모리에서 **address**를 메모리에 건넨다. 원하는 **instruction**이 있다면 해당 **instruction**의 주소를 Program Counter가 메모리로 보낸다. 이 과정이 CPU가 메모리에 “이 **instruction**을 수행해야 하니 메모리에 관련된 코드를 나에게 보내다오”라고 말하는 과정이며, **instruction**이 오면 그걸 실행(**execute**)하는 과정에서 또 관련 매개변수 (**operands**)에 관련된 주소를 보내고 관련 정보를 받아온다.

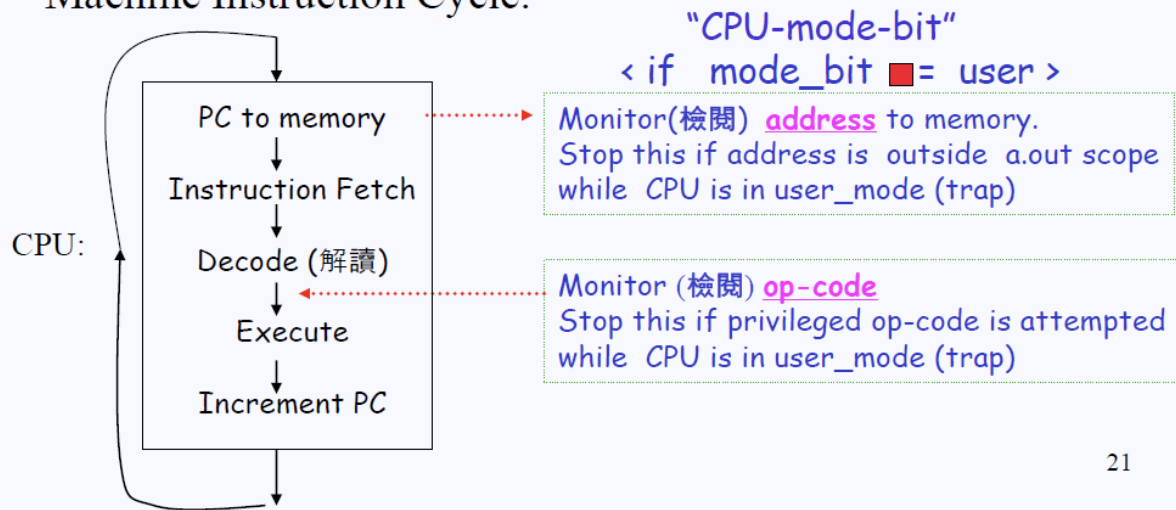
이처럼 CPU는 계속 메모리에게 address를 보내는 작업을 진행하는 것인데, CPU가 메모리로 주소를 보낼 때는 그 당시 모드 비트가 어떤 것으로 되어 있느냐가 정말 중요하다. 먼저 아래의 그림을 살펴보자.

• CPU_mode_bit :



- SW need HW's help to "prevents" illegal action
- one HW bit ■ in CPU (usually part of PSW)
- machine instruction (SW) can read/write this
- Access to mode_bit ■ is privileged op-code

Machine Instruction Cycle:



위의 그림은 빨간색 모드 비트가 유저모드였을 경우를 상정하고 설명을 한다. 앞에서 인천 공항 검색대로 예시를 들었었는데, 운영체제에 빗대서 다시 정리해보자. 먼저 첫 번째 보안 검색(address)을 진행한다.

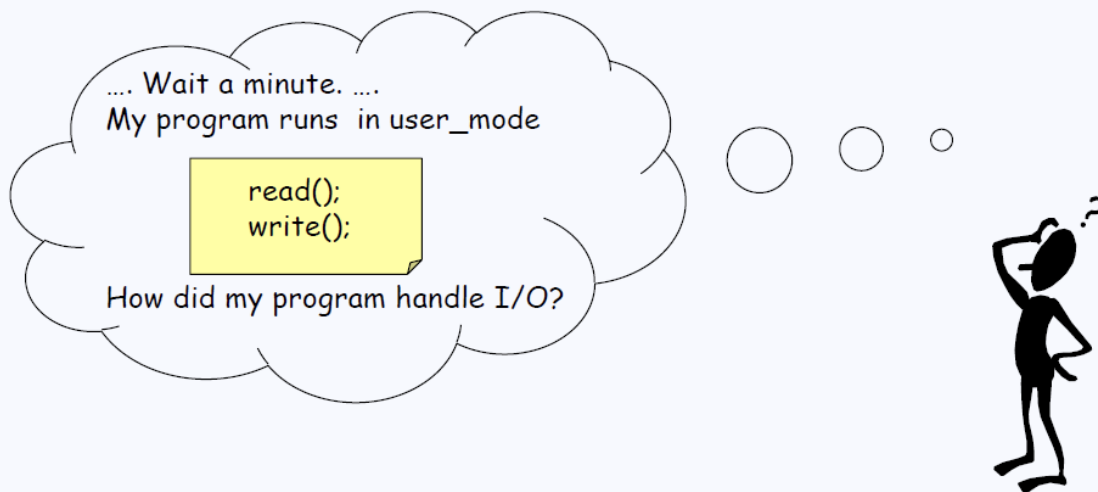
CPU와 메모리 사이에는 **MMU(Memory Management Unit)**이라는 하드웨어가 존재한다. 이 MMU의 역할은 CPU로부터 메모리로 가는 **address** 정보를 조사하는 것이다. 즉, CPU가 메모리에게 넘기는 address 정보가 올바른지 판명하는 것이다. “CPU야. 너가 지금 보낸 주소가 너가 할 수 접근있는 메모리 범위를 벗어나지는 않았니?”라고 말이다.

첫 번째 검사를 무사히 통과하면 **Instruction Fetch**, 즉 instruction을 가져온다. 위에서 우리는 instruction의 구조가 op-code, operands로 구성된다는 것을 확인했는데, 바로 이 **op-code**를 보고 이것이 덧셈이구나, 뺄셈이구나, 곱셈이구나 등을 확인하게 된다. 이게 바로 2번째 검사다.

명령어(op-code)를 확인해 봤는데, 만약 이것이 **privileged op-code(I/O와 같은 중요한 역할을 하는 실행)**을 시도하려 한다면 그 순간 바로 **CPU**를 뺏겨버린다.

이렇게 작동을 한다면 위에서 언급한 보안이슈를 만족할 수 있다. **illegal access**가 사전에 예방(prevent)이 되고 차단이 될 수 있다.

- When my program runs, CPU mode bit = **user_mode**
 - cannot execute I/O instruction
 - cannot access special registers
 - cannot access memory outside current a.out
- When OS kernel runs, CPU mode bit = **kernel_mode**
 - no restriction at all



22

NEAOSS MC2.0 2-2-3-I v1.0

CC-BY 2.0 KR. © Korea OSS Promotion Forum, NIPA

정리하자면 모드 비트가 유저모드일 때, CPU가 접근하는 메모리 주소가 실행 중인 프로그램의 범위 밖이거나 I/O instruction 등의 금지된 실행을 하려고 한다면 CPU를 운영체제로부터 박탈당한다. 반대로 커널모드였을 경우는 위에서 언급한 검증 절차를 전혀 밟지 않아도 된다.

커널모드는 어떠한 메모리 영역도 접근 가능하며 어떠한 연산도 시행할 수 있는 특권을 가지고 있다. 오직 커널만이.

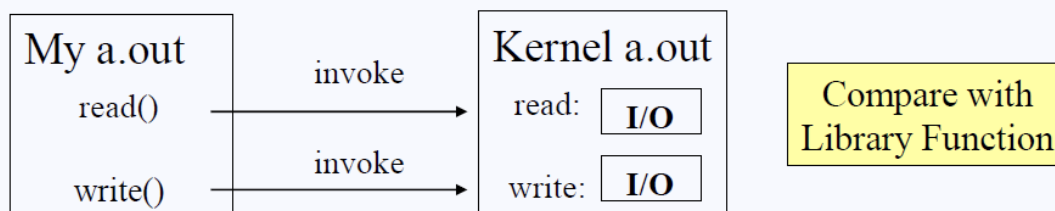
3. printf("Hello World!")의 진실

그러면 이제 우리가 한 가지 궁금한 점이 있다. 우리는 프로그램을 만들 때 소스코드에 입출력과 관련된 함수를 작성한다. `printf()` 를 사용하거나 `get()` 등 디스크에 접근해서 값을 읽어오거나 화면에 문자를 출력하는 함수를 사용한다. 우리가 작성한 코드와 프로그램

은 유저모드에서 아무런 제약없이 사용할 수 있었는데, 왜 I/O가 금지되었다고 말을 하는 것일까?

정답은 “소스코드에서만 그렇게 보인다”이다. 소스코드에서는 개발자가 입출력을 관리하는 것처럼 보이지만, 소스코드를 컴파일 한 후에 바이너리 파일을 열어보면 I/O와 관련된 instruction은 전혀 존재하지 않는다.

- **Source:** (has I/O statement)
`"read next byte from disk file X into my variable Y"`
- **Binary:** (compiler changes I/O statement to following)
`"prepare all parameters (for disk read)"`
`"execute chmodk instruction"`
`/* After compile, I/O statements (eg read, write)`
`does not appear in my binary a.out`
`I/O statements are only found in kernel`
`In order to perform I/O,`
`my a.out has to call kernel I/O functions at run time`
`This is all done by (compiler, OS and HW) */`



23

NEAOSS MC2.0 2-2-3-I v1.0

CC-BY 2.0 KR. © Korea OSS Promotion Forum, NIPA

위에서 언급했듯이 I/O를 하고 싶으면 커널이 가지고 있는 **function**을 호출하는 방법밖에는 없다. 즉 커널에 부탁을 해야한다는 말이다. 그 행위를 우리는 위에서 시스템 콜 (**System call**)이라고 했었다.

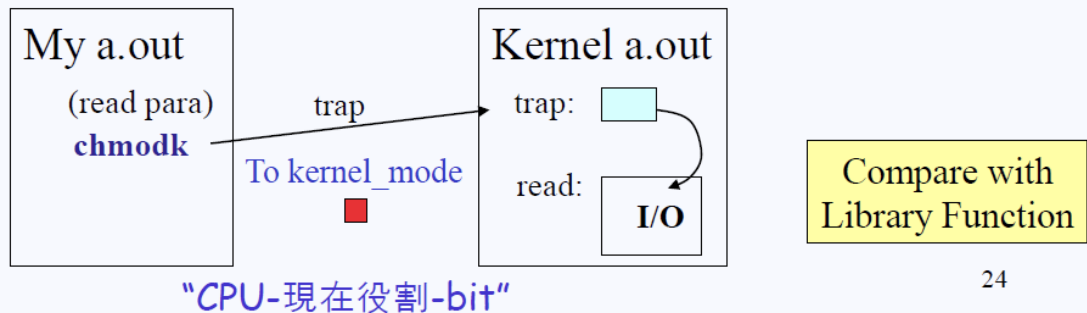
그런데 의문점이 하나 있다. 함수를 호출한다는 건, 특정 기능을 호출한다는 건 내 프로그램 안에 있는 함수를 호출하는 개념인 것인데, 어떻게 다른 프로그램(커널)의 함수를 호출하지?

그 원리는 우리가 입출력 함수를 담은 소스코드를 컴파일 했을 때, 해당 부분(입출력 등)이 등장하면 **'change CPU protection mode to Kernel'** 명령어를 수행하는 것이다. CPU의 모드비트를 바꾸는 것이다.

소스파일에 **privileged instruction**이 등장하면 바이너리파일에서 **chmodk**로 변환된다는 건 이제 이해할 수 있을 것이다. 그렇다면 이제 하드웨어가 이 명령어를 어떻게 실행시켜가는지 이해해보자.

3.1 chmodk를 실행했을 때 하드웨어에서 벌어지는 일들

- At run time (part I: **hardware**)
 - my program executes “**chmodk**”
 - this is privileged instruction
 - CPU cannot continue (in user_mode)
 - HW trap
 - HW saves CPU state vector (including return address)
 - HW sets CPU_mode_bit ■ <== kernel mode
 - HW jumps to trap handling routine (in kernel a.out)



24

소스파일에 입출력 파트에 컴파일러가 컴파일 시에 **chmodk**를 넣어둔 후 발생하는 일은 첫 번째로 유저로부터 CPU를 뺏는다. 더 이상 유저모드에서 실행(run)할 수 없게 만드는 것이다. 그걸 우린 “**trap**에 걸린다”라고 한다.

트랩은 인터럽트랑 비슷한 개념으로 글 최하단 부분에 자세하게 설명을 적어놓았다.

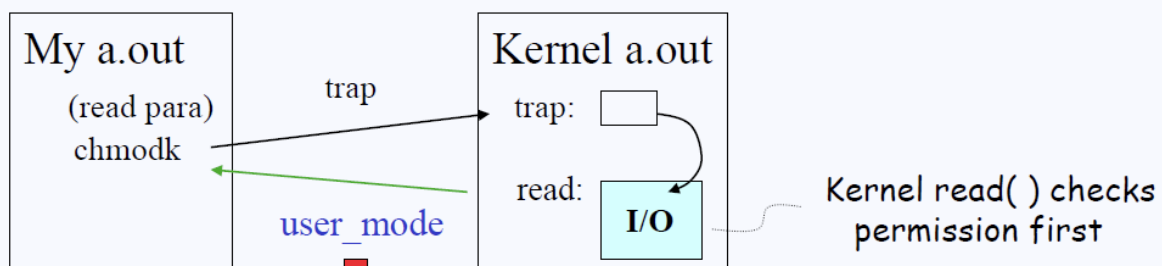
트랩에 걸린 후에 트랩핸들러 루틴(트랩을 처리하는 루틴)으로 진입하는데, 해당 루틴은 커널 안에서 처리된다. **chmodk**명령어를 처리하기 직전에 우리는 시스템 콜과 관련된 parameter를 사전에 약속된 곳에 기록을 해둔다. 왜냐하면 이후 트랩이 명령어를 처리할 때 유저가 어떤 처리(write, read, open, close)를 하려 했는지를 알아야 하기 때문이다.

I/O와 관련된 디스크와 관련 섹터에 "A프로그램에 B작업을 요청합니다."라는 관련 parameter를 적어두는 것이다. 그럼 트랩핸들러가 그 정보를 확인한 후, "아 너는 I/O를 하고 싶고 그 중에서도 read를 하고 싶구나"처럼 확인을 하는 것이다.

이제 유저가 뭘 처리하고 싶어하는지 알았으니 그냥 진행하면 되는 것일까? 아니다. 유저가 해당 디스크나 메모리 영역에 권한(read, write, execute 중 하나)이 있는지도 확인해야 한다.

Note: 커널 **function**은 라이브러리 function과 다르다. 라이브러리에 있는 함수를 우리가 호출할 때는, 해당 코드가 그대로 우리 소스코드 안에 Copy & Paste 되는 반면 커널의 함수를 호출한다는 건 커널에게 부탁을 하고 커널이 해당 함수를 수행해주는 개념이다. 커널이 메모리에 항상 상주해 있어야 하는 이유 중 하나이기도 하다. 유저가 어떤 함수를 요구할지 모르기 때문에 항상 대기해야 하는 것이다.

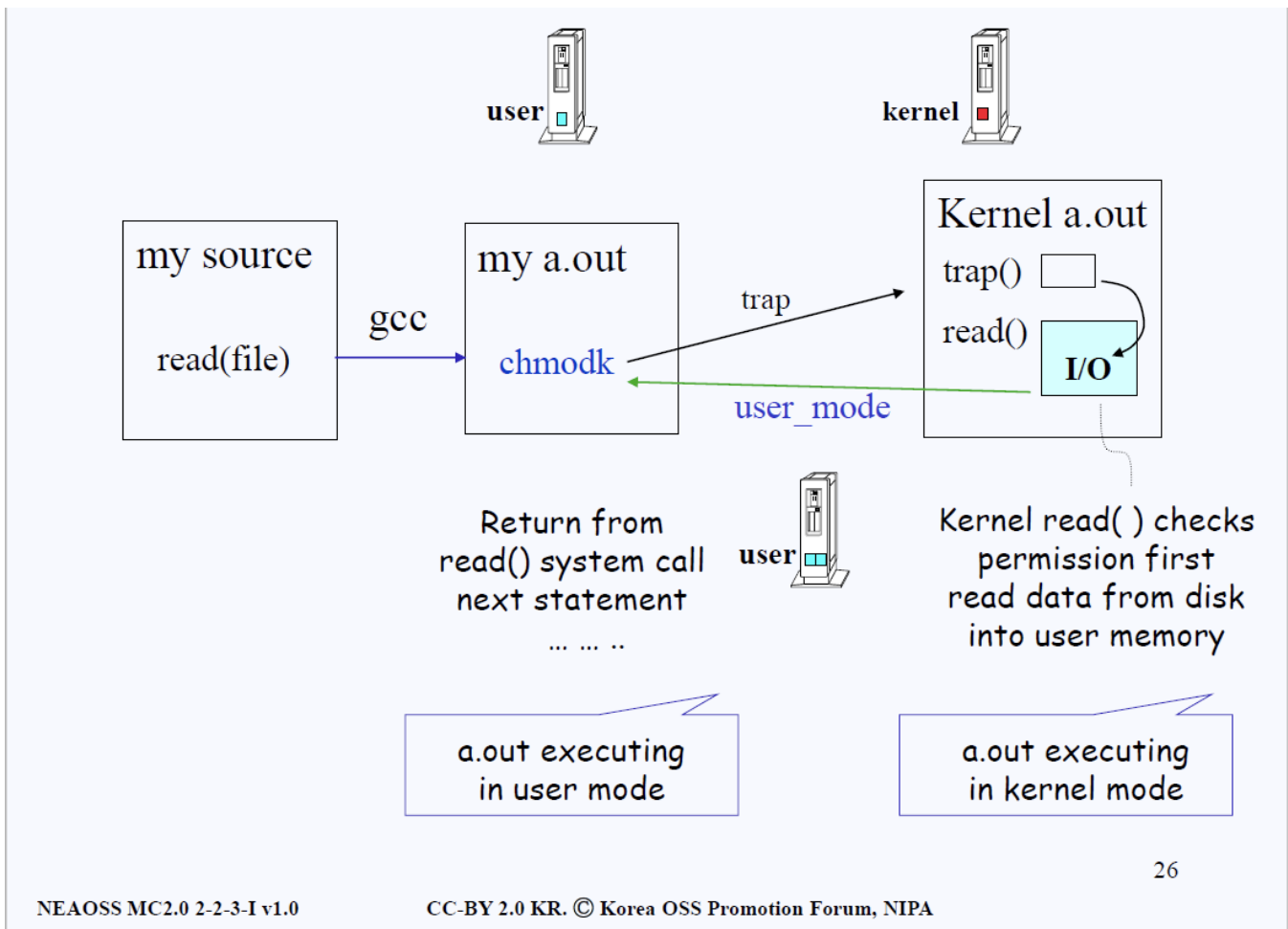
- Run time (part I: **software**)
 - Now trap handler (in kernel a.out) starts
 - inspects what caused trap
 - system call, divide by zero, memory bound,?
 - invoke appropriate kernel function (system call)
 - All done? CPU_mode ■ <== user_mode
 - restore state vector
 - return to interrupted location in "my a.out"



"CPU-現在役割-bit"

25

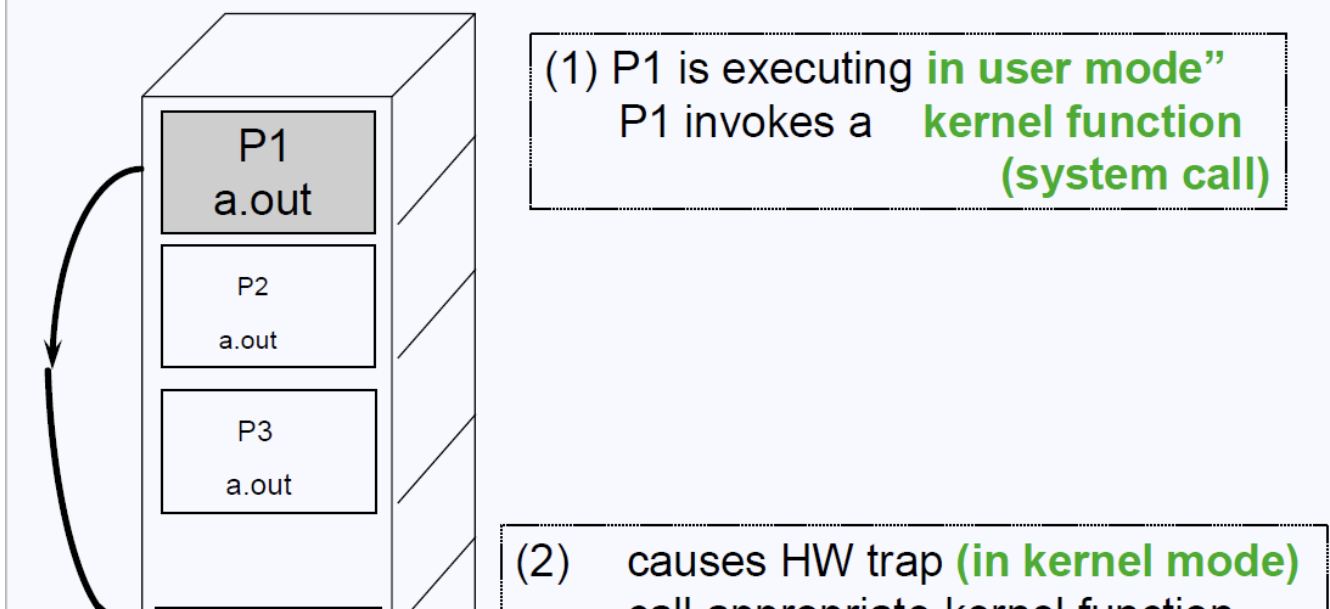
위의 검증 과정들을 거친 후 read/write 등의 작업이 끝나면 트랩으로 돌아가고, 해당 트랩에서 다시 유저모드로 return된다. 그 때 비로소 유저는 자신이 처리한 작업이 제대로 완료됐는지를 확인할 수가 있다.



26

과정을 한 번 더 간략하게 도식화한 것이 위 그림이다. 소스파일을 컴파일해서 바이너리파일에 **chmodk**를 끼 넣고, 그로 인해 트랩이 발생한다. 커널 안의 트랩 핸들러는 적절한 검증절차를 거친 후에 유저모드에서 요구했던 작업을 진행한다. 그 후 다시 유저모드로 돌아온다.

Alternating User mode & Kernel mode





27

NEAOSS MC2.0 2-2-3-I v1.0

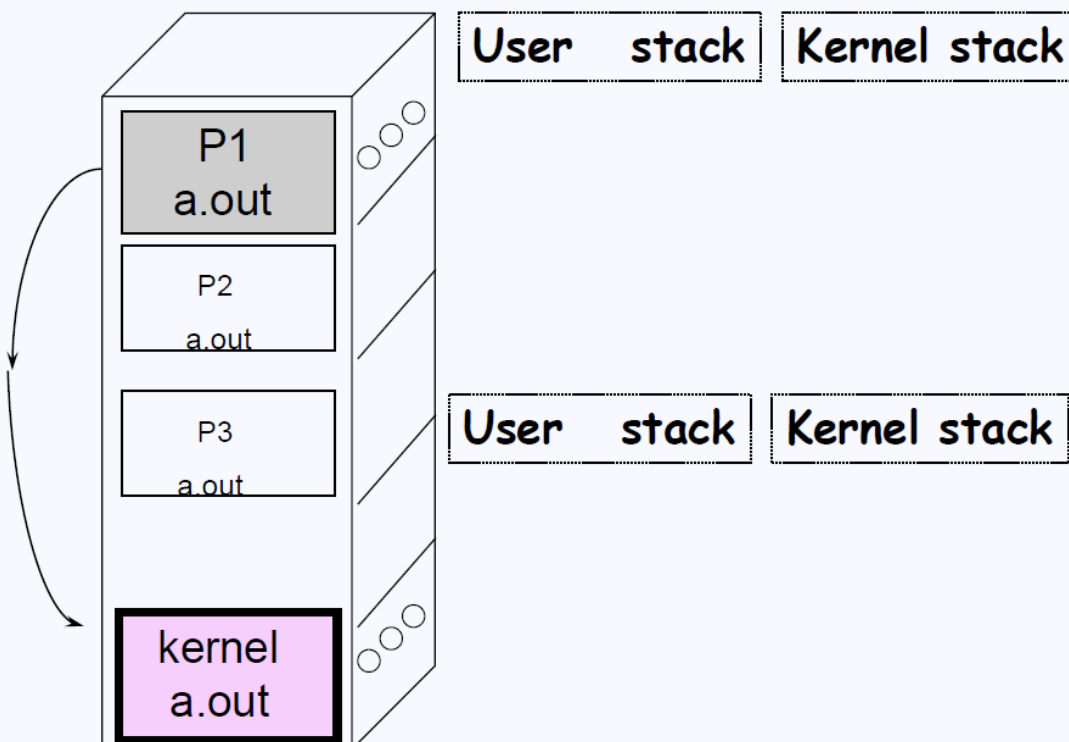
CC-BY 2.0 KR. © Korea OSS Promotion Forum, NIPA

커널모드와 유저모드 사이의 모드가 바뀌는 과정을 계속해서 반복한다. 프로그램 내에 더 이상 커널에 요구할 것이 없을 때까지. 모든 프로그램은 다 유저모드로 **run**하다가 커널 모드로 **run**하는 걸 반복한다.

자, 그런데 **run**을 한다는 건 유저 모드 혹은 커널 모드 안에 있는 **function**을 사용한다는 것인데 **function**들을 계속 **call** 했다가 **return**하고 **call** 했다가 돌아오고 하는 과정을 반복한다는 것이다.

function에는 보통 **local variable**(지역 변수)들이 있기 마련이고, 해당 지역 변수들이 어디 저장되는지 생각해보자. 일단 이 지역변수들은 메모리에 언제부터 언제까지 존재할까? 그렇다. 함수가 호출되고 리턴되기까지 존재한다. 따라서 미리 메모리에 담아두는 비효율적인 방법보다는 임시적으로 메모리에 담아뒀다가 삭제하기 용이한 자료구조를 택해야 하는데, 그것이 바로 **스택(Stack)**이다.

Every program needs two stacks



28

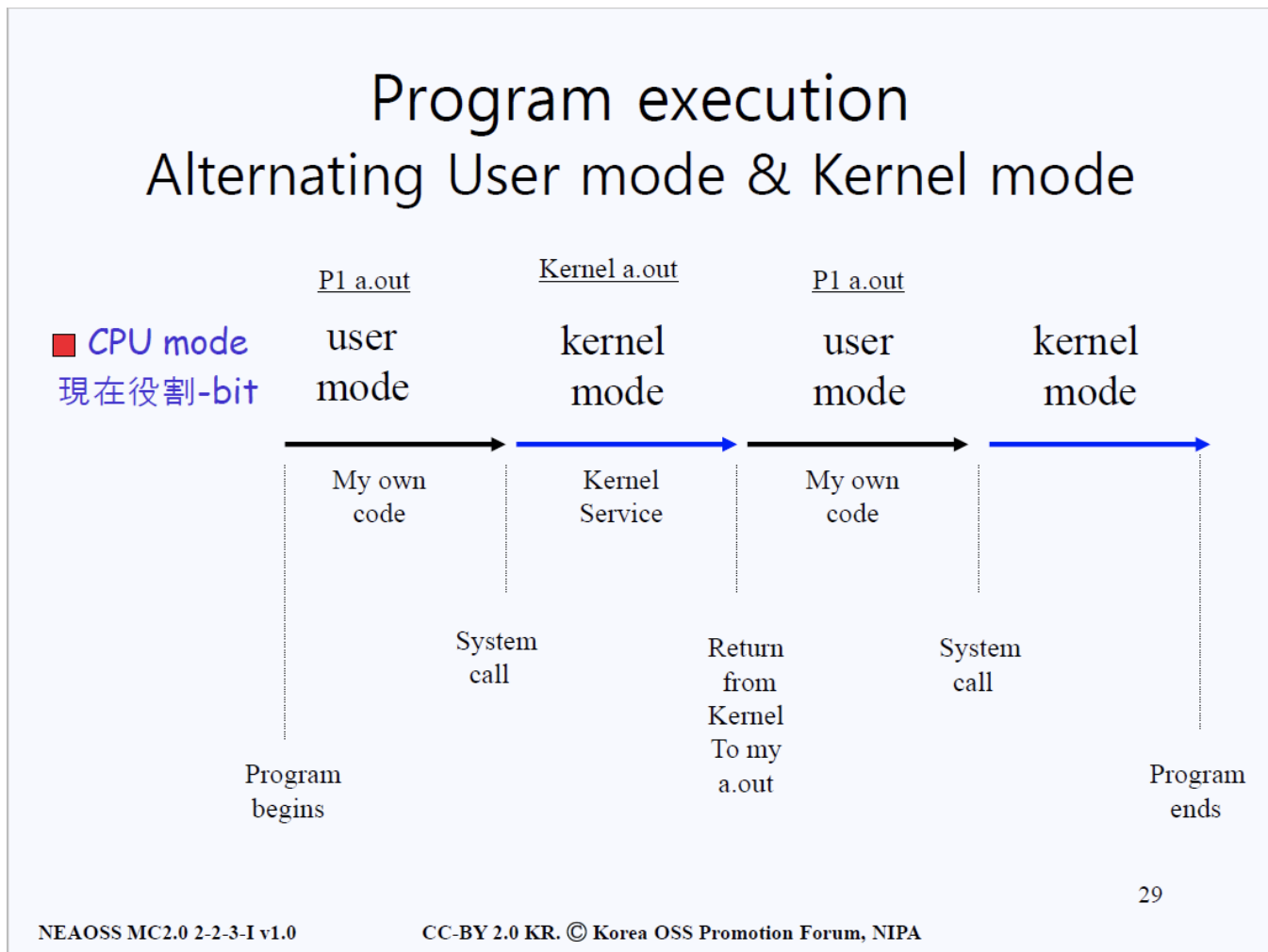
NEAOSS MC2.0 2-2-3-I v1.0

CC-BY 2.0 KR. © Korea OSS Promotion Forum, NIPA

function이 호출되면 해당 **function**의 **local variable**(지역변수)들이 스택에 **push**(삽입)된다. 지역변수 뿐만 아니라 함수가 끝나고 돌아갈 주소(**return address**)등도 함께 **push**(삽입)된다. 어떤 프로그램이나 유저모드와 커널모드를 오가기를 반복한다는 점을 우리는 알고 있다.

유저모드에서 유저만의 **function**들을 실행하고 리턴하기 위해서 유저모드에도 스택이 필요한 것이고, 커널모드도 마찬가지로 자신만의 **function**을 실행하고 리턴하기 때문에 스택이 필요하다.

4. 정리



결국 프로그램의 실행은 유저모드와 커널모드를 **Alternating**하는 것으로 볼 수 있다. 위 그림을 주욱 따라가면서 1장에서 했던 내용들을 다시 상기해보자. 유저모드가 자신만의 코드를 수행하다가 커널에게 부탁할 일이 생기면 **System call**을 한다.

그렇게 되면 커널모드로 진입하여 **Kernel a.out**이 진행되면서 커널이 일을 처리한 후 다시 유저모드로 돌아와서 작업을 진행한다. 이 과정은 프로그램이 종료할 때까지 반복된다.

마지막으로 인터페이스에 대한 언급으로 1장 강의노트를 마치고자 한다. 커맨드는 유틸리티의 또 다른 이름이고 유틸리티는 **disk resident program**이다(function이 아니다). 운영체제의 interface를 살펴보면, 커맨드는 키보드 **interface**고 **system call**이나 **library call**은 **function interface**이다.

리눅스 운영체제를 사용하게 된다면, **man** 명령어를 정말 많이 사용하게 될텐데 각종 명령어를 모를 때 `man <command>` 방식으로 원하는 명령에 대한 상세 정보를 확인할 수 있다. 명령어 우측 괄호 안에 있는 숫자가 해당 명령어가 커맨드, 시스템 콜, 라이브러리 함수인지를 분간해준다.

Manual

- Manual volumes

(1) commands

(2) system call

(3) library functions

command | sys call | library
(1) (2) (3)

CAT(1) User Commands
CAT(1)
NAME
cat - concatenate files and print on the standard output
SYNOPSIS
cat [OPTION] [FILE]...
DESCRIPTION
Concatenate FILE(s), or standard input, to standard output

39

5. 마치며

1장 강의노트를 마친다. 고건 교수님의 강의는 무료로 제공되고 있으니 운영체제를 공부한다면 꼭 한 번 들어봤으면 한다. 강의를 듣지 않아도 강의노트만으로 이해가 된다면 시간 절약을 위해 강의노트를 보고 공부해도 좋을 듯하다.

5.1 인터럽트와 트랩

인터럽트는 시스템 내에서 하드웨어가 생성한 흐름 변경이다. 인터럽트 원인을 처리하기 위해 인터럽트 처리기가 사용된다. 제어는 인터럽트된 컨텍스트 및 명령으로 리턴된다. 트랩은 소프트웨어가 생성한 인터럽트다. 장치 폴링의 필요성을 없애기 위해 인터럽트를 사용하여 I/O의 완료를 알릴 수 있다. 트랩을 사용하여 운영 체제 루틴을 호출하거나 산술 오류를 포착 할 수 있다.

인터럽트는 하드웨어 인터럽트이며 트랩은 소프트웨어 호출 인터럽트이다. 하드웨어 인터럽트 발생은 일반적으로 다른 하드웨어 인터럽트를 비활성화하지만 트랩에는 해당되지 않는다. 트랩이 제공 될 때까지 하드웨어 인터럽트를 허용하지 않으려면 명시 적으로 인터럽트 플래그를 지워야한다. 일반적으로 컴퓨터의 인터럽트 플래그는 트랩이 아닌 (하드웨어) 인터럽트에 영향을준다. 즉,이 플래그를 지우더라도 트랩을 방지 할 수는 없다. 트랩과 달리 인터럽트는 CPU의 이전 상태를 유지해야한다.

트랩은 일반적으로 소프트웨어 인터럽트 라고하는 특별한 종류의 인터럽트다. 인터럽트는 하드웨어 인터럽트(하드웨어 장치의 인터럽트)와 소프트웨어 인터럽트(트랩)를 모두 포괄하는보다 일반적인 용어다.