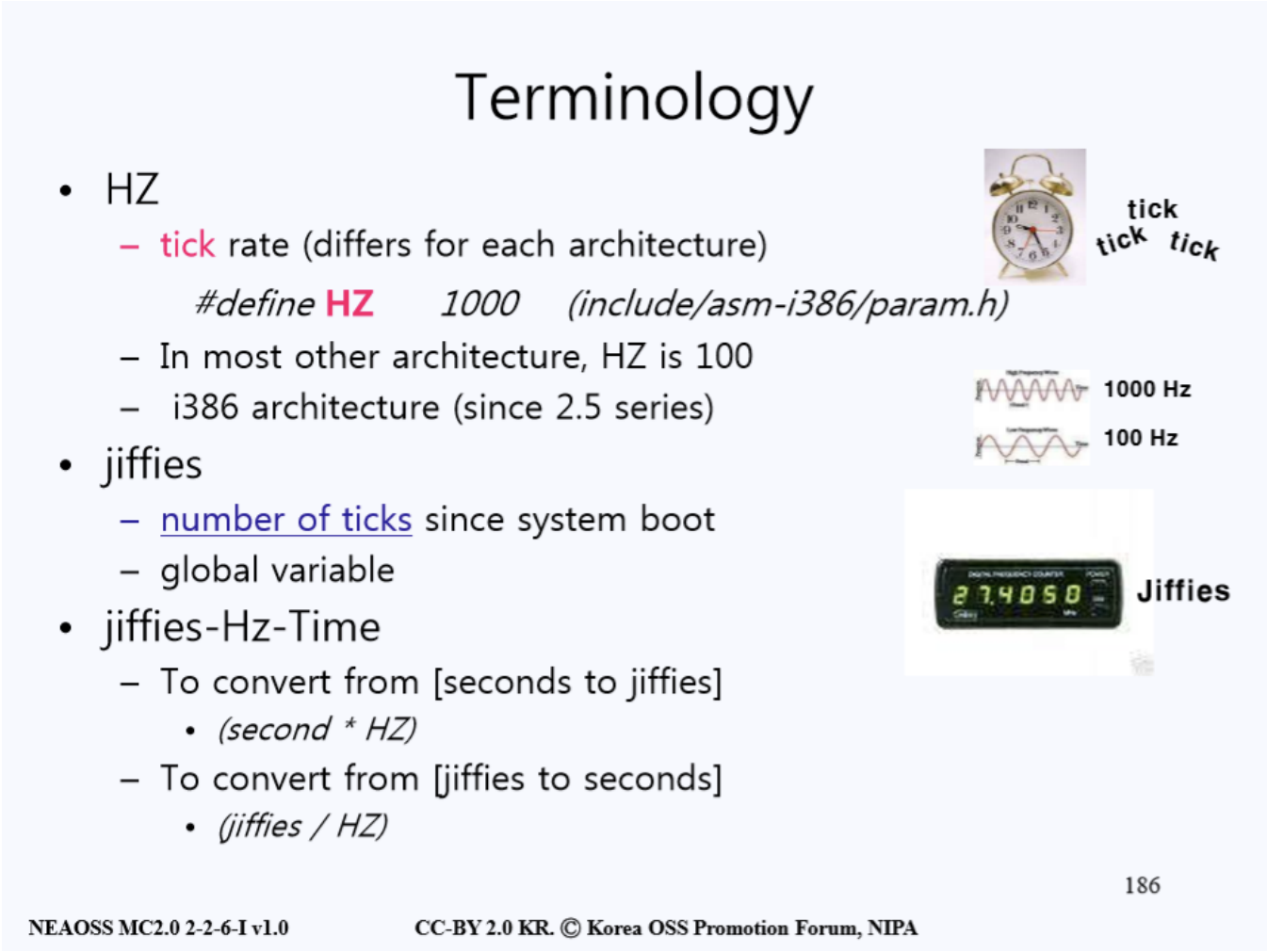
리눅스 커널(운영체제) 강의노트[6]

Seung Joo Choi (Bookstore3)
Nov 19, 2018 · 11 min read

지난 5강에선 **Timeslice** 라는 **CPU**에게 주어지는 사용시간과 **CPU**의 사용 순서를 관리하는 커널 스케쥴링에 대하여 공부를 했다. 이번 시간에는 그 시간의 단위에 대한 공부와 여러 개의 인터럽트가 일어났을 때의 관리방법에 대하여 공부를 할 것이다.

1. 타이머와 시간 관리

우리는 시계가 돌아갈때 나는 소리를 째깍째깍 거린다고 표현을 하며 이는 영어로 Tick Tack이라고 표현이 된다. 이때 일초에 1000번 째깍거리면 1000 헤르츠(Hertz, HZ)라고 하고 이는 1 밀리세컨드(1 Millisecond)가 된다. 이러한 표현들은 물리학에서 사용하는 표현들이고 `#define HZ 1000` 이라는 표현을 사용하면 **1초에 1000번 인터럽트**가 걸리는 설정으로 된다. 대부분의 경우에는 `100` 을 걸어 놓는다.



시스템이 켜진(부팅 된) 이후에 몇 번 tick을 했는지 기록한 것을 우리는 **Jiffies**라고 표현을 한다. 이는 전역 변수이며 카운터의 역할을 한다. 이러한 `Jiffies`를 설정된 `HZ`로 나누면 몇 초가 흘렀는지 알 수 있다.

> 예시: Jiffies가 24000이고 HZ가 100으로 설정되어 있었다면 24000/100 즉 시스템이 부팅된 이후 240초가 흘렀다는 것을 알 수 있다.

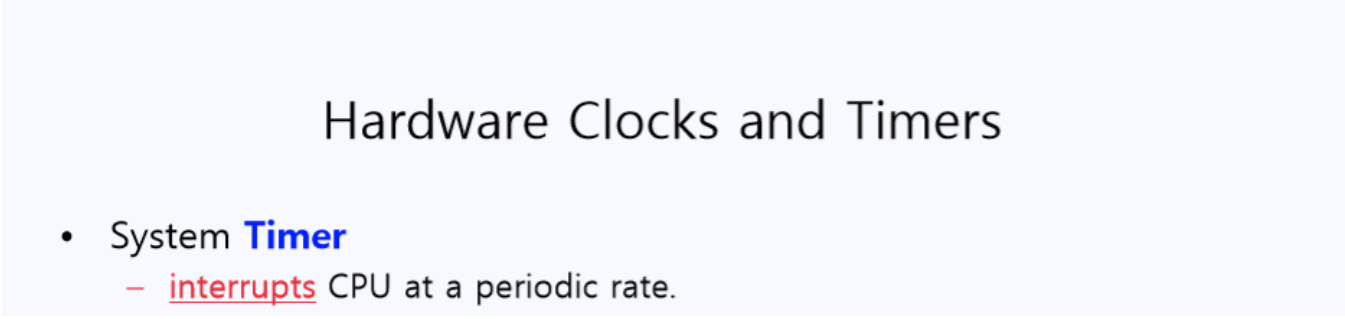
위에서 우리는 `HZ`의 단위로 인터럽트를 걸기위해 이러한 시간 개념을 도입했다는 것을 살펴봤. 그렇다면 **왜 1초에 100번씩이나 인터럽트가 걸려야 하는 것 일까. 왜 I/O 인터럽트처럼 어떤 입력이나 할 일이 생겼을 때만 인터럽트를 하면 되는 것 아닐까**라는 의문을 품을 수 있을 것이다.

시스템에 시간 단위를 도입한 이유는 **먼저 특정 시간마다 반복이 필요한 일들을 처리하려면 시스템이시간의 개념을 알아야 하기 때문이다.** 사실 가장 중요한 이유는 **스케쥴링**에 필요하다는 점이다. 프로세스들은 CPU를 사용할 수 있는 시간인 **Timeslice**를 배정받게 된다.

이때 얼마만큼의 시간이 지났는지 파악해 다음 작업에 `CPU`를 넘겨주는 등의 역할을 할때 시간의 단위를 사용하게 된다. 즉, 하나의 작업이 다 끝날때까지 다음 작업이 기다리는 것이 아닌 **`HZ`의 단위로 계속해서 프로세스들이 돌아가며 작업을 할 수 있게 해주기 위해 계속해서 특정 시간마다 인터럽트를 걸어주는 것 이다.**

> 첫번째 이유에 대한 예시: 만약 안구건조증이 있는 사람이라면 2시간 마다 하던 일을 멈추고 반복적으로 약을 넣어줘야 한다.
>
> 두번째 이유에 대한 예시: 학교에서 하루종일 한 과목만 할 수는 없기에 수업 시간 종소리를 통해 다음 수업을 진행한다.

그렇다면 이런 인터럽트의 횟수를 `100`에서 `1000`번을 하게 `HZ`를 설정하면 좋은 것일까? 꼭 그렇지만은 않다. 물론 인터럽트 하는 주기가 늘어났기에 반응을 해주는 횟수는 증가할 수 있다. **그러나 많은 인터럽트를 하면서 오버헤드가 증가하기 때문에 현재 쓰는 장치들은 대부분 100 HZ**라는 적절한 숫자로 설정해 준 것이다.



- It can be provided by
  - Programmable Interrupt Timer (PIT) on X86
    - kernel programs PIT on boot to drive timer interrupt at HZ
  - or APIC timer (see interrupt),
  - or processor's TSC(timestamp counter) register

- **Real-Time Clock** (RTC)
  - Nonvolatile device for storing the time
  - RTC continues even when the system is off (small battery)
  - On boot, kernel reads the RTC
  - Initialize the wall time (in struct timespec ***xtime***)

188

이러한 시스템의 시간은 크게 **Timer** 와 **Real-Time Clock(RTC)** 2가지로 나눌 수 있다. 먼저 **Timer**는 주기적으로 **CPU**에게 인터럽트를 거는 역할을 한다. 이러한 **Timer**는 프로그램적으로 인터럽트를 걸 수 있게 설정할 수도 있다.

Real-Time Clock은 현실 세계의 시간을 표현하며 PC의 전원을 꺼놔도 보조 베터리를 통해 계속해서 현재 시간을 측정 한다. 후에 다시 PC를 켰을때 시스템은 **Real-Time Clock**의 시간을 보고 현재 시간을 표시한다. 그렇다면 이제 이런 타이머 인터럽트를 관리하는 핸들러의 실제 구현을 아래 그림과 함께 알아보자.

Timer Interrupt Handler

```
void do_timer(struct pt_regs *regs)
{       jiffies_64++;    /* increment jiffies */
        update_process_times(user_mode(regs));
        update_times();
}
```

1 if user mode
0 if kernel mode

```
void update_process_times(int user_tick)
{       struct task_struct *p = current;
        int cpu = smp_processor_id(), system = user_tick ^ 1;
        update_one_process(p, user_tick, system, cpu);
        run_local_timers();             /* marks a softirq */
        scheduler_tick(user_tick, system);
                /* decrements the currently running process's
                timeslice and sets need_resched if needed */
}
```

```
p->utime += user;
p->stime += system;
```

```
static inline void update_times(void)
{       unsigned long ticks;
        ticks = jiffies - wall_jiffies;
        if (ticks) {
                wall_jiffies += ticks;
```

```
            update_wall_time(ticks);
        }
        calc_load(ticks);                                    190
}
```

먼저 인터럽트가 걸리면 `dotimer(struct ptregs *regs)` 로 들어와 `Jiffies` 를 하나 증가시키게 된다. 이때 `Jiffies` 는 시스템이 켜진 이후 매번 카운트를 하는 역할을 함으로 `64비트` 라는 매우 큰 크기로 정해준다. 그 후
에 `update_process_times(user_mode(regs))` 를 통해 몇번이나 인터럽트가 걸렸는지 업데이트를 하는데 이때 인터럽트가 걸린 순간에 User 모드였는지 Kernel 모드였는지 같이 기록을 한다.

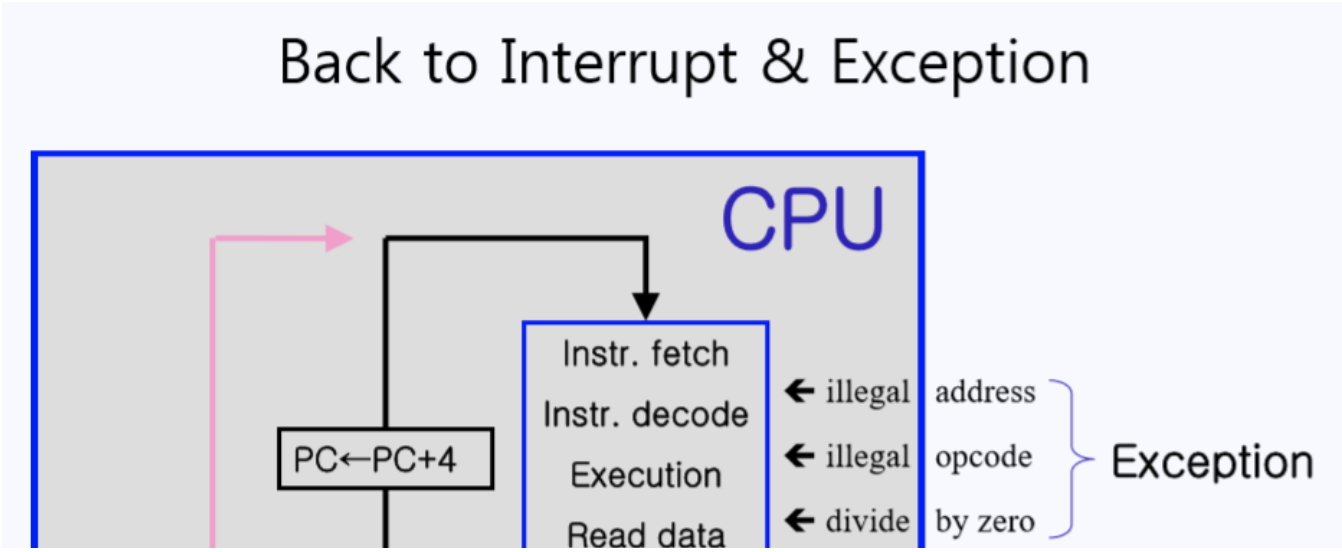
이러한 타이머는 프로그램으로 특정 시간으로 설정 할 수도 있으며 지연 시킬 수도 있다는 점 정도만 알아두고 이런 타이머에 의해 걸리는 인터럽트에 대해 자세히 알아보자

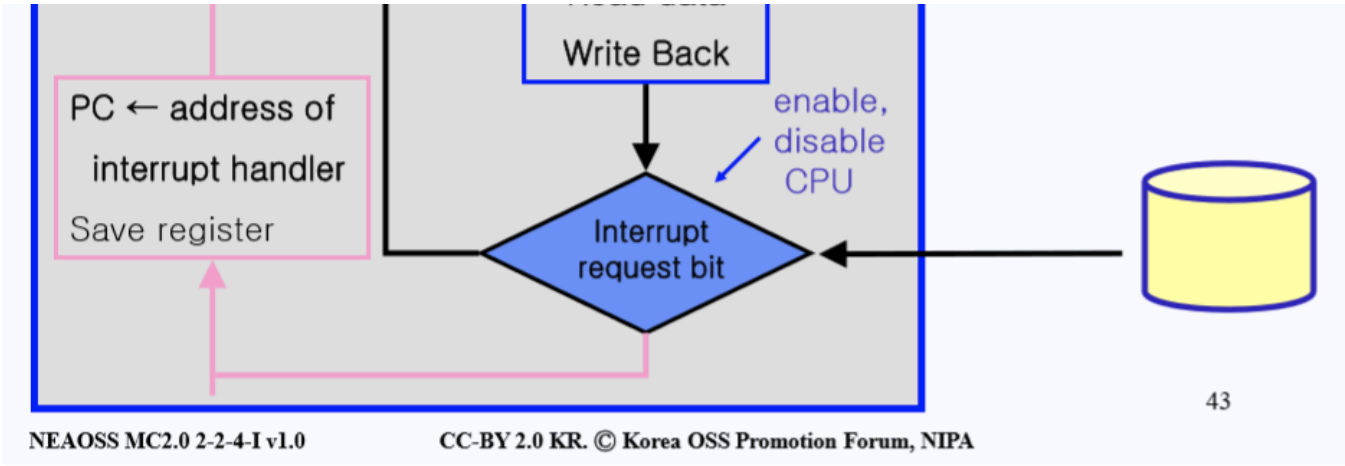
2. 인터럽트

인터럽트란 CPU가 프로그램을 실행하고 있을 때, 입출력 하드웨어나 예외상황 등이 발생해 작업을 처리가 필요할 경우에 커널에게 처리해 달라고 요청하는 것이다. 그럼 먼저 하드웨어적 문제중에서 CPU에 대해 알아보자

2.1 CPU

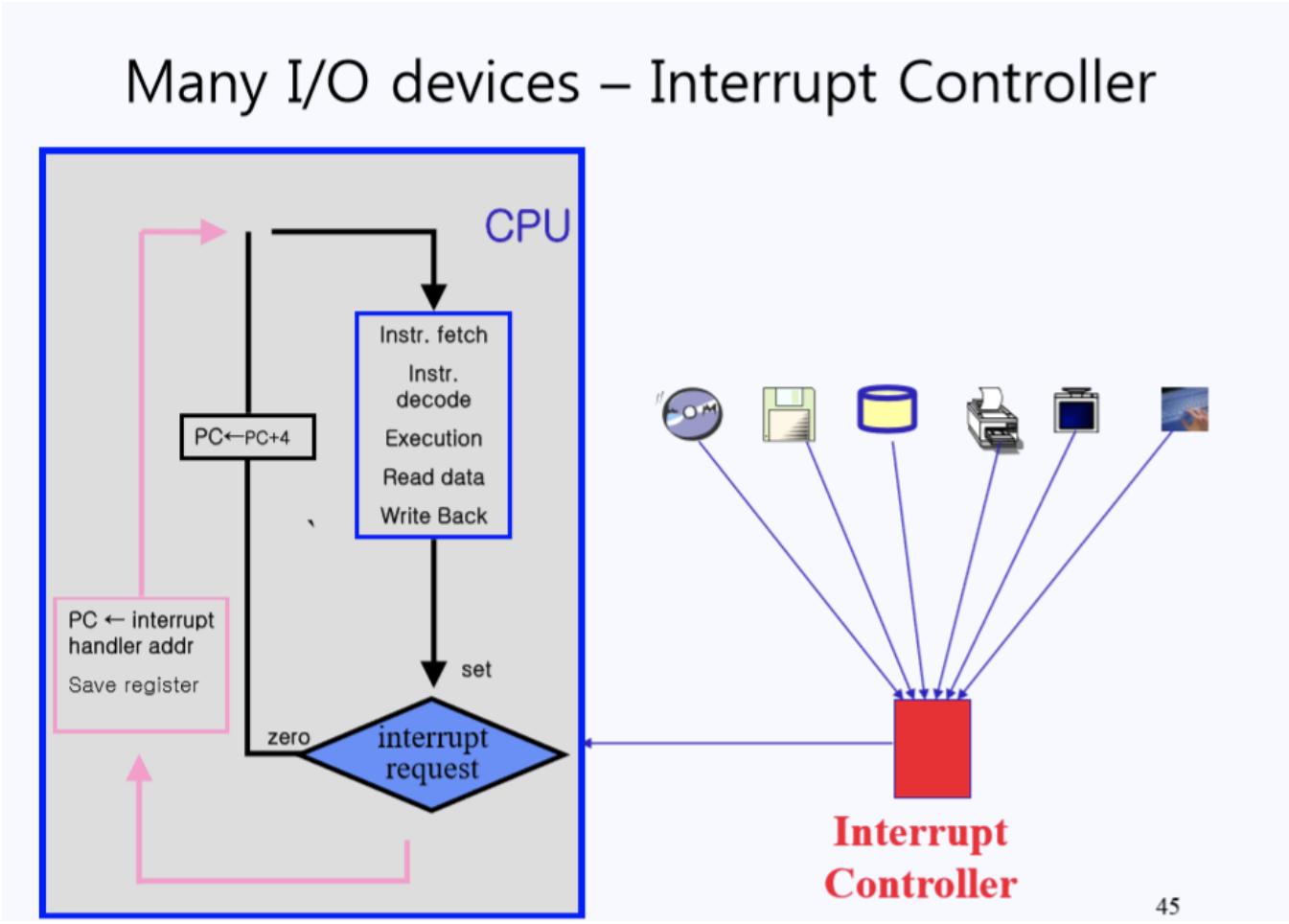
CPU는 먼저 인스트럭션을 가져온다. 그리고 그 인스트럭션을 분석하고 실행을 한다. 이렇게 실행을 하기 위해선 데이터를 읽어와야 하는 경우도 종종 있게 된다. 또한 연산을 처리 했다면 처리된 결과를 반환해준다. 인스트럭션이 다 끝났으면 그 다음 인스트럭션을 가져오기 위해 프로그램 카운터를 증가시킨다(보통은 `4 byte` 정도 증가시킨다).





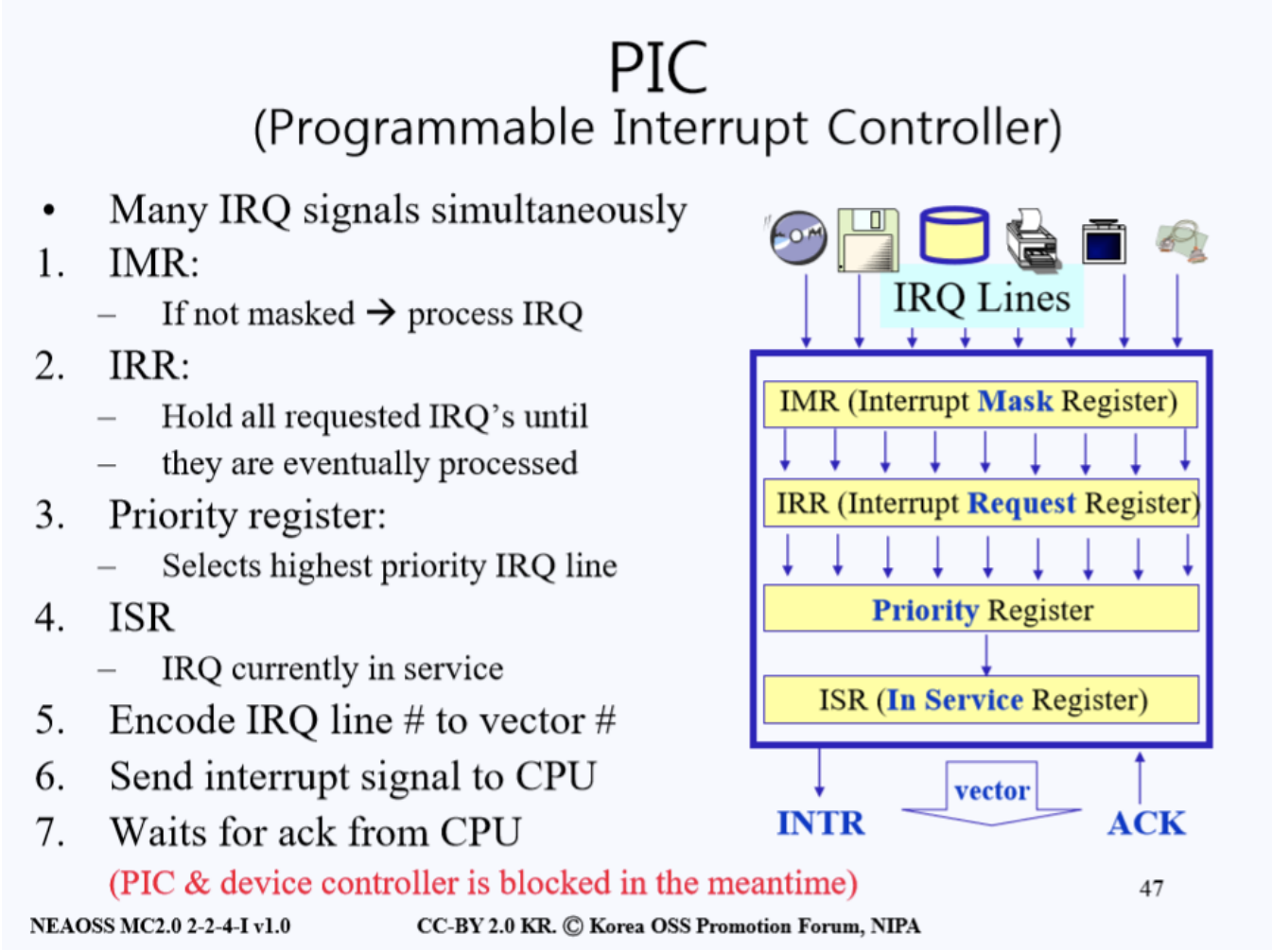
이런 일련의 작업을 하던 중 **Disk가 인터럽트를 걸었다**고 생각해보자. 그러면 **Interrupt Request Bit** 한 비트를 설정을 한다. 이 비트가 걸려있으면 작업을 계속 돌지 않고 **프로그램 카운터에 인터럽트 핸들러의 새로운 주소를 저장**한다. 그리고 다시 진행을 해서 인스트럭션을 가져오면 이제 아까 저장했던 인스트럭션을 가져와 점프를 한 효과를 보게된다. 물론 이런 인터럽트가 걸리지 않게끔 설정할 수도 있다. 예를 들어 컴퓨터를 부팅할 때의 경우엔 `Interrupt Request Bit` 설정을 `disable` 시켜서 인터럽트 당하지 않게 만들 수 있다.

위의 경우는 디스크 하나만 인터럽트를 걸었을 경우에 대한 설명이었고, 아래 그림은 인터럽트를 거는 장치들이 많을 때의 상황을 설명한다.



여러개의 장치가 인터럽트를 거는 것을 통제하기 위해 **Interrupt Controller**라는 개념을 도입했다. 우리는 이런 **Interrupt Controller**를 **PIC(Programmable Interrupt Controller)**라고 부른다. 여기서 `Programmable` 이 붙는 이유는 소프트웨어적으로 관리가 가능하기 때문에 프로그램이 가능하다는 뜻이 붙은 것이다.

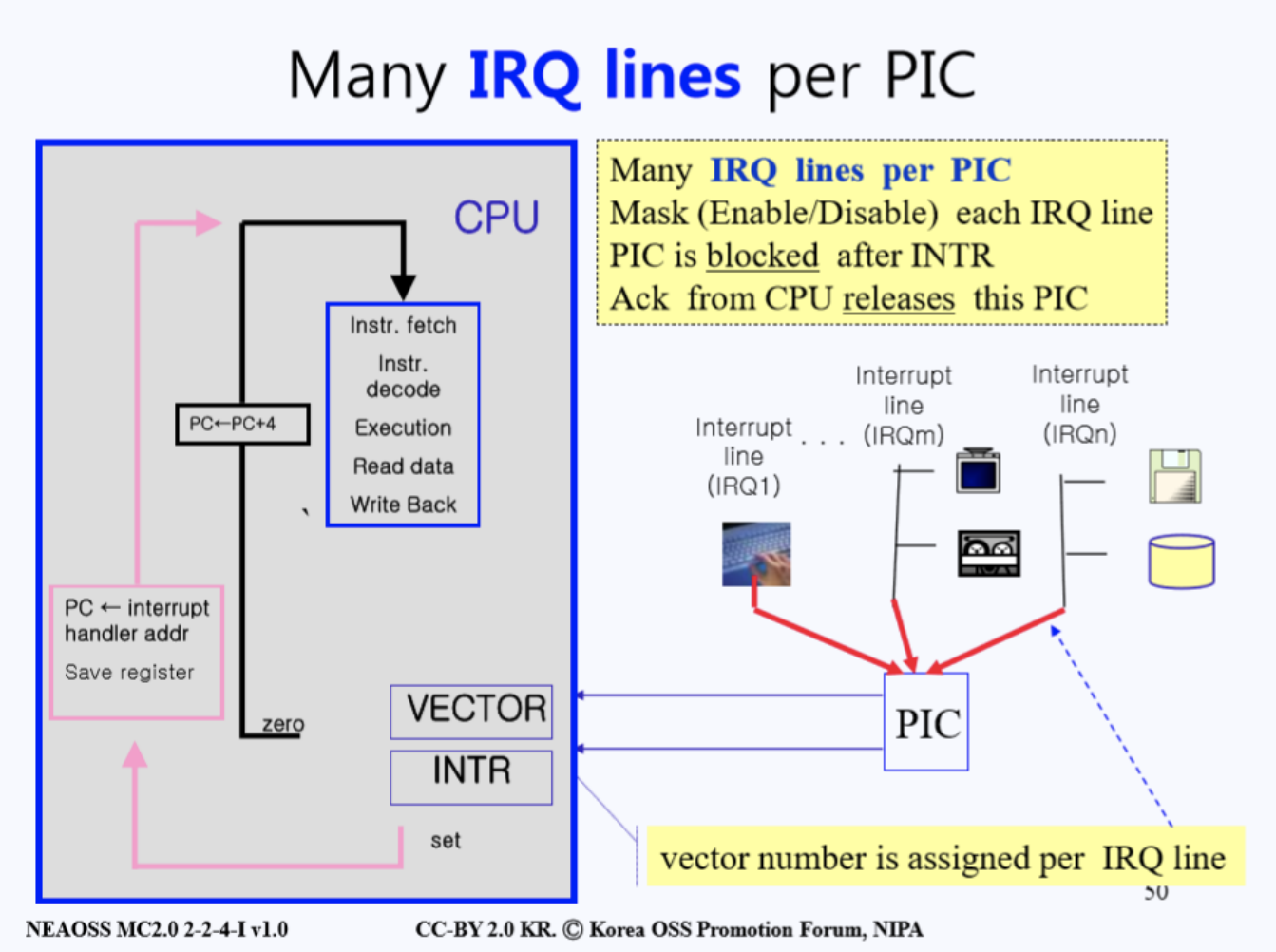
2.2 PIC



PIC에서는 여러개의 장치들이 인터럽트 요청을 한다. 이때 이런 요청들을 한 장치들은 **Interrupt Request Line(IRQ Line)**에 연결이 된다. 이렇게 요청이 들어온 요청 라인들은 **Mask Register**을 통해서 `0` 일 경우 차단이 되고 `0` 이 아닐 경우엔 그 다음 단계로 통과가 된다. 이러한 `Masking` 은 소프트웨어적으로 차단할 장치등을 설정할 수 있다. 그 후에**Interrupt Request Register**에서 `Masking` 이 되지 않은 장치들만 요청을 할 수 있게 설정해 주는 단계를 처진다.

이렇게 설정이 된 요청들 중 우선순위가 제일 높은 요청을 **Priority Register**에서 받고 지금 인터럽트 요청이 진행 중 이라고**In Service Register**에 등록하

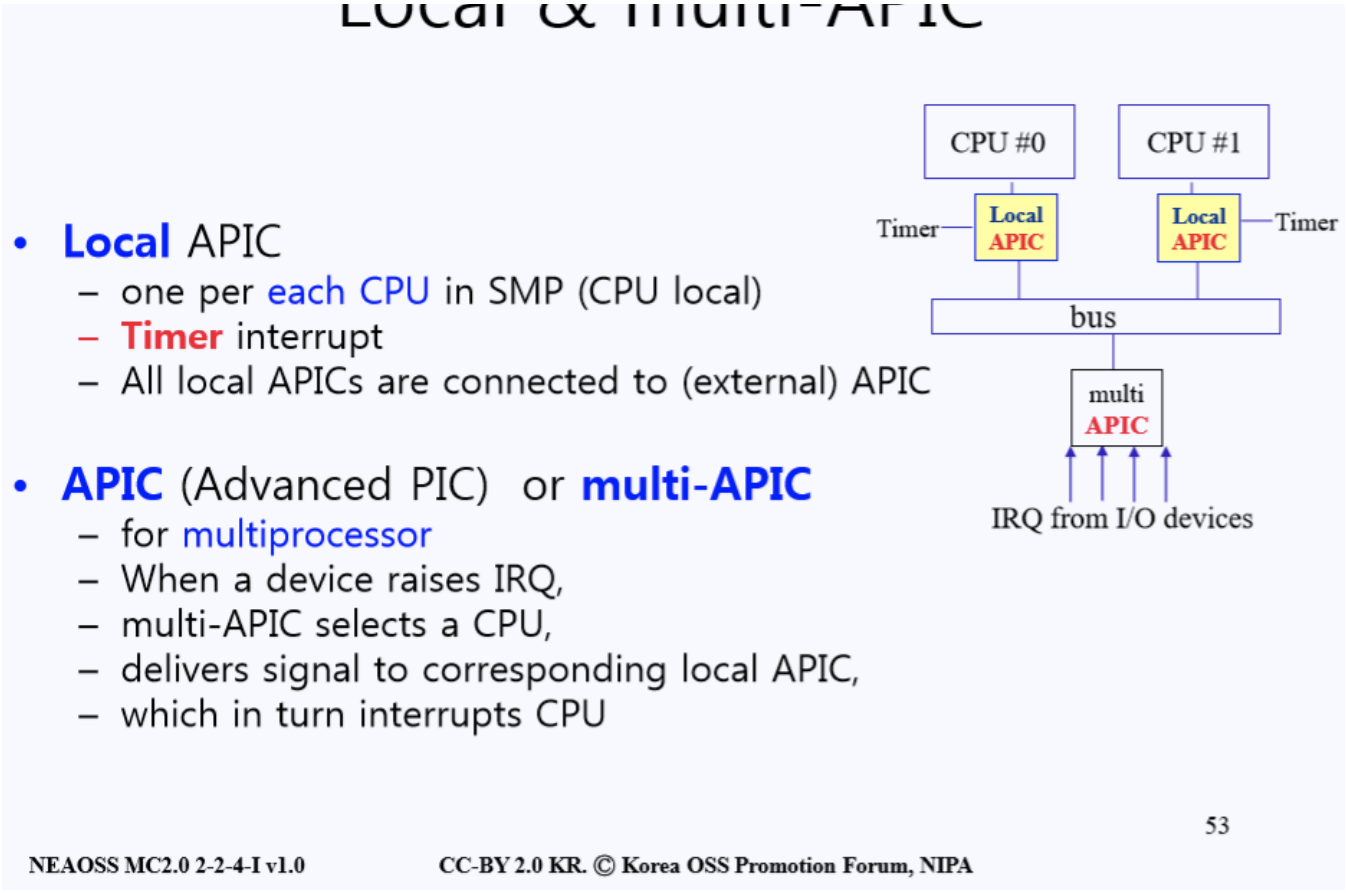
고 **INTR**, 즉 CPU 에게 인터럽트 요청을 한다. 이때 어떤 IRQ Line 에서 요청이 들어온건지를 **Vector**에 넣어 보낸다. 그렇게 요청을 보내고 나서 CPU 가 요청을 처리 했다는 **ACK**신호를 보낼 때 까지 다른 PIC 와 장치들은 차단되어 있다. 이러한 요청 처리 단계를 위해 CPU는 빠르게 일들을 처리해 줘야 한다.



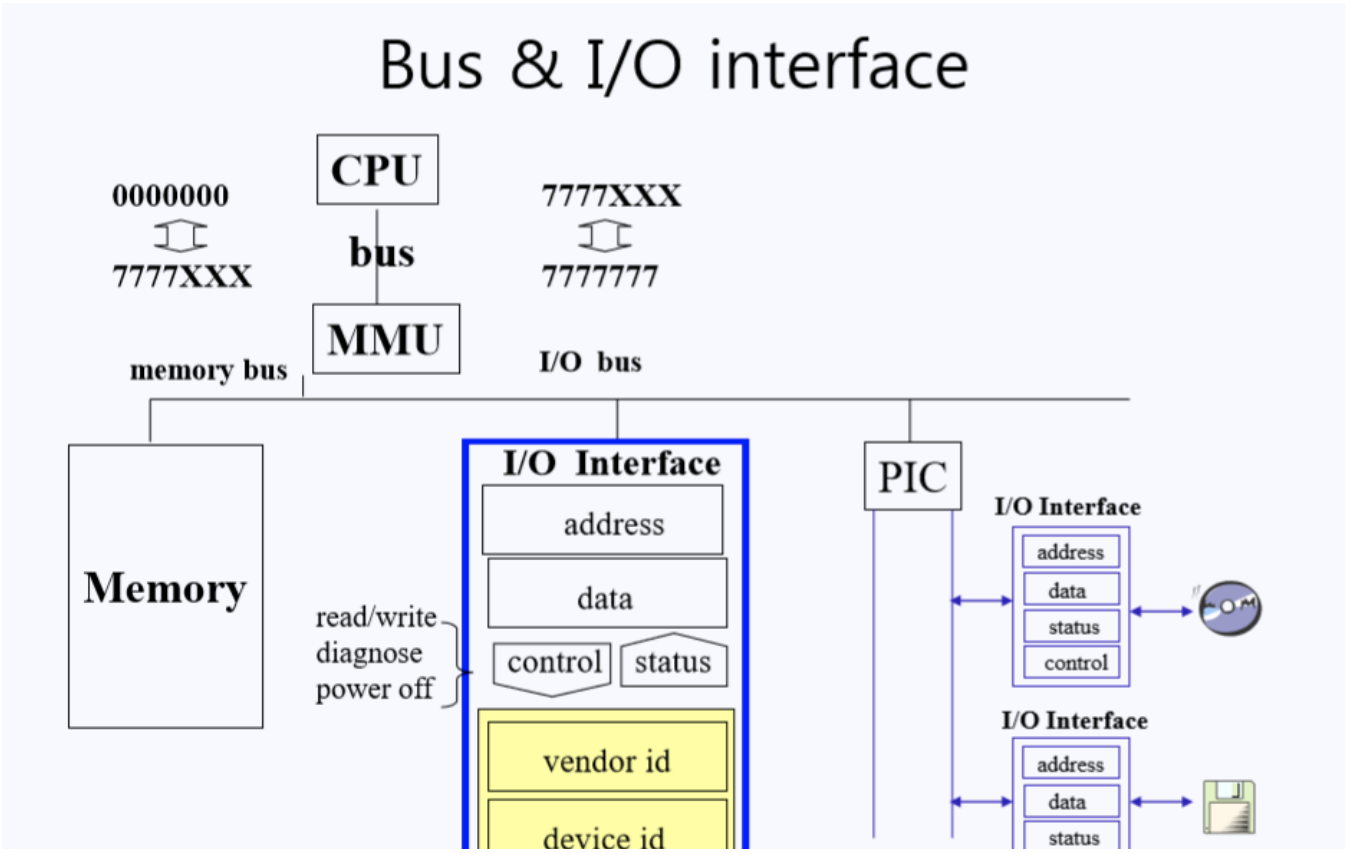
지금까지 설명한 요청 단계를 정리해 보면 위의 그림과 같이 나온다. PIC 가 INTR 과 Vector 을 CPU 에게 보내고 그럼 Interrupt Request Bit 을 설정하고 그럼 그걸 본 CPU 는 그 요청을 처리할 새로운 공간을 만들어 일들 처리후 ACK 를 다시 PIC 에게 보낸다. 이러는 동안 요청을 보낸 PIC 은 ACK 가 올때까지 다른 요청들을 차단하고 기다리고 있는다.

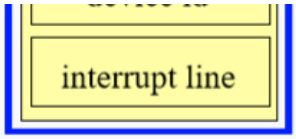
2.3 SMP & AMP

이번에는 여러개의 요청들이 멀티프로세싱 환경에서는 어떤 방법으로 처리되는지 알아보자.



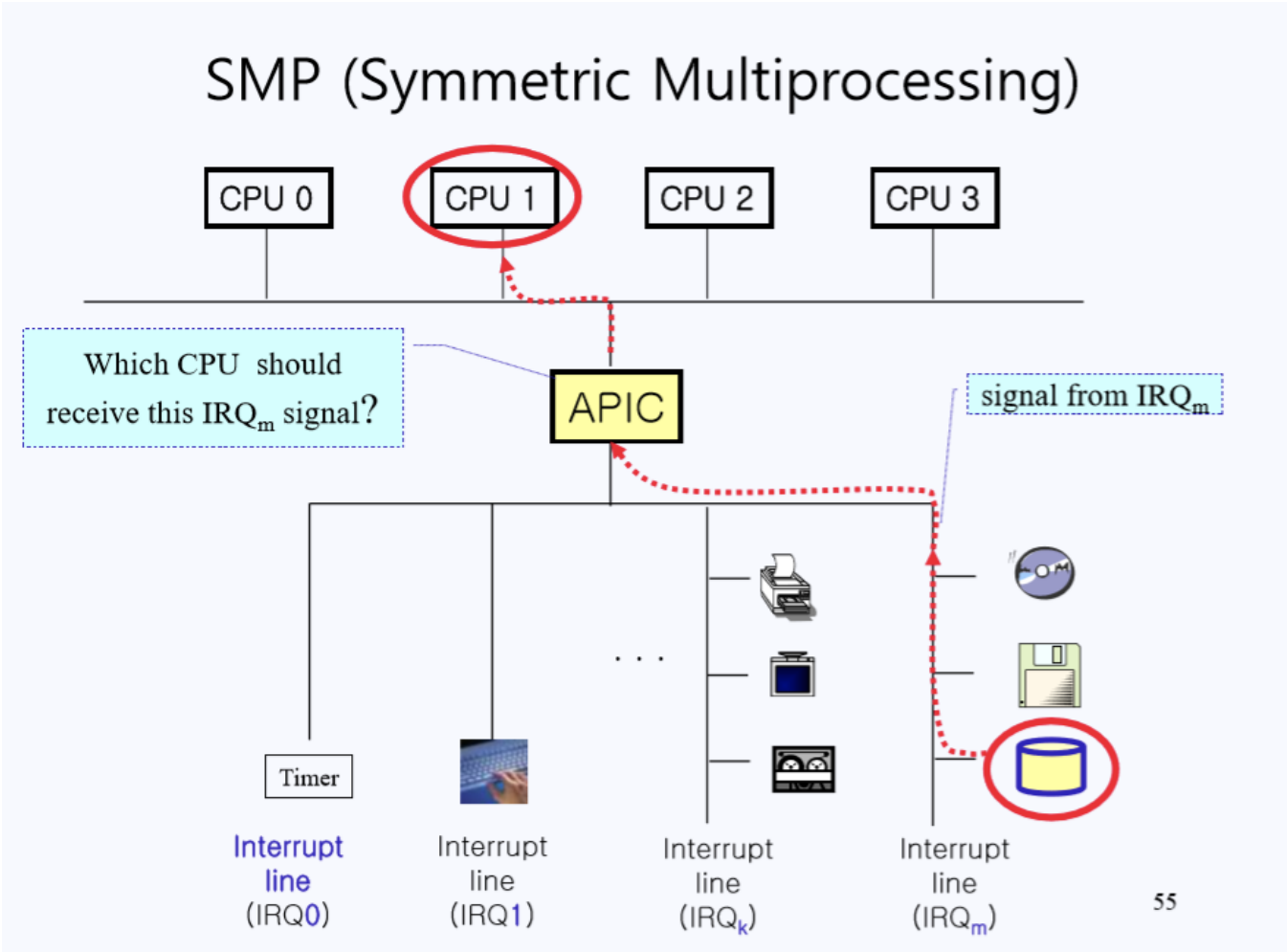
먼저 CPU 2개가 있고 이 CPU 가 bus 에 달려있다고 하자. 그리고 모든 I/O 장치들은 **multi APIC(Advanced Pic)**에 달려있다. APIC 이란 멀티 프로세서를 위한 PIC 이다. 또 다른 APIC 은 각각의 CPU 에 달려있는데 이때의 조그만한 PIC 들은 **Local APIC**이라 하며 여기에는 정기적으로 인터럽트를 걸어주는 **Timer**만 달려있다.





여기서 잠시 컴퓨터 구조에 대한 얘기를 해보자. CPU 가 0000000~7777XXX 번 메모리를 메모리 관리 유닛에게 보내면 위 그림의 좌측 **Memory** 쪽으로 달려간다. 그러나 7777XXX ~ 7777777 번 메모리를 유닛에게 보내면 **I/O Interface**쪽으로 버스를 타고 달려가게 되어있다.

I/O 버스 들에는 각각 I/O 장치들이 연결이 되어있는데 이런 장치들은 컴퓨터 뒤에 보면 있는 각종 연결장치, 즉 **I/O Interface Card**들로 연결이 되어있다. 이런 I/O Interface Card 의 구조는 Address, Data 와 보조 레지스터인 Control, Status 등으로 구성되어 있다. 그 외에 장치를 만든 회사 이름, 모델 아이디 등등이 들어있으며 어느 카드에 꽂혀있는 장치인지 등에 대한 정보도 들어있다. 이러한 인터페이스를 바탕으로 위 그림의 오른쪽 처럼 각종 장치들이 인터페이스 카드들에 연결이 되어 있는 것이다.



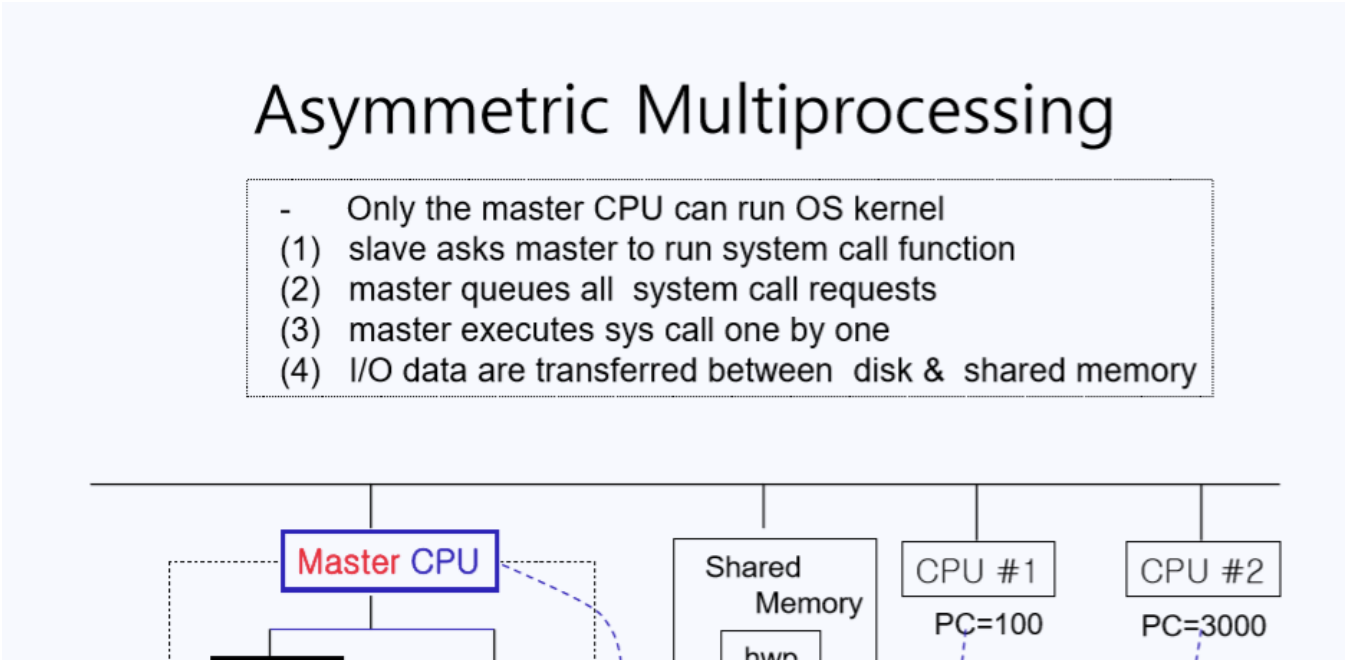
위 그림은 방금 설명한 과정을 **SMP(Symmetric Multiprocessing), 즉 대칭형 멀티 프로세싱 방식**에서 처리하는 과정을 나타내는 그림이다. SMP 란 **두 개 이상의 동일한 프로세서가 하나의 메모리, I/O 디바이스, 인터럽트 등의 자원을 공유하여 단일 시스템 버스를 통해 각각의 프로세서는 다른 프로그램을 실행하고 다른 데이터를 처리하는 시스템**을 말한다.

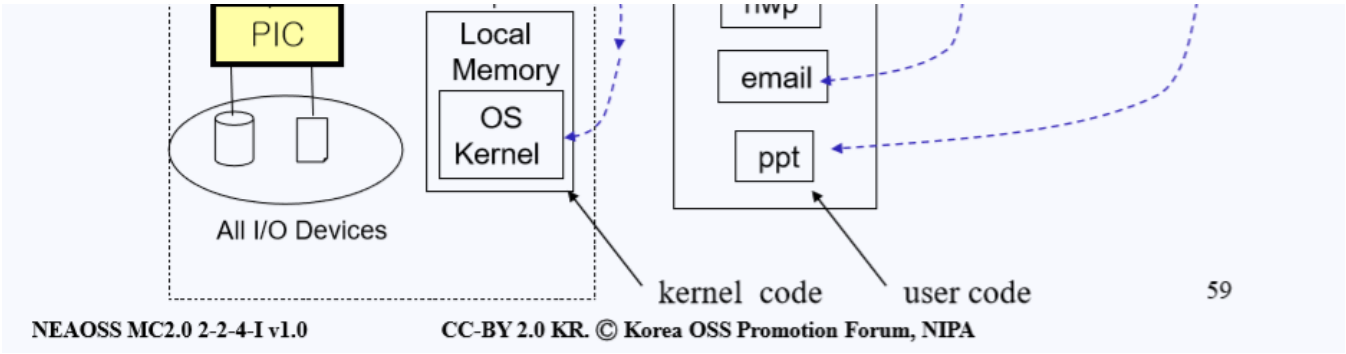
즉, **두 개 이상의 프로세서가 하나의 컴퓨터 시스템을 공유**하도록 연결되어 있으며, **각각의 프로세서가 독립적으로 자신의 작업을 처리하는 방식**이다. 이러한 방식에서 디바이스가 I/O 인터페이스 카드에 연결이 되어 요청을 보내면 APIC 이 받아 처리를 하는데 CPU 간의 차이가 없는 대칭형이기 때문에 어느 CPU 에 요청을 전달하는지는 다음과 같은 두 가지 방식을 사용한다.

첫번째 방식은 **Static Distribution방식으로** 정적으로 정해진 곳에 보낸다. 이 경우에는 이미 만들어진 Static Table 을 통해 결정을 하게 된다.

두번째 방식은 **Dynamic Distribution 방식**으로 동적으로 결정을 하는데 이때 **동적 IRQ 분배 알고리즘**을 통해 보낼 곳을 정한다. 이 알고리즘의 목표는 **우선순위가 제일 낮은 프로세스를 돌리고 있는 CPU에게 IRQ를 주는것**이다.

그렇다면 프로세스를 실행하고 있는 경우에는 어떻게 처리를 해야 할까. 이를 해결하기 위해 모든 프로세스에 카운터를 둔다. 이 **카운터의 값이 가장 큰 CPU**가 IRQ 를 받게 되는데 이때 카운터를 0 으로 낮춰주고 IRQ 를 받지 않은 다른 모든 CPU 의 카운터는 증가시킨다. 이렇게 카운터를 증가시킴으로서 **나중에 어떤 CPU가 IRQ 를 제일 적게 받아 제일 처리를 많이 안했는지 분별할 수 있는 척도로 사용을 하게된다.**





그럼 이제 **Asymmetric Multiprocessing(AMP), 비대칭형 멀티 프로세싱에선 어떤 방식을 사용하는지 알아보자.** AMP 란 **두개 이상의 각각의 프로세서가 자신만의 다른 특정 기능을 수행하는 방식**을 말한다.

예를 들어 하나의 프로세서가 메인 운영체제를 실행하도록 하고 다른 프로세서는 I/O 기능을 전용으로 수행하는 형식 등을 말한다. 이런 구조에서 IRQ 를 처리하는 방식이 위 그림에 나와있다. 각각의 CPU 가 다른 작업들을 하고있다. 이때 **Master CPU**는 **본인만의 메모리**를 가지고 있는데 여기 **OS 커널**이 들어있다.

즉, **Master CPU만 I/O 인스트럭션을 가지고 있는 구조**이기에 다른 CPU 가 I/O 를 하려면 Master CPU 에게 신호를 보내야 한다. 이러한 **주종 관계라는 간단한 구조 덕분에 디자인이 쉽다.**

그러나 Master CPU 에게 시스템 콜 요청이 많아지면 과부화가 쉽게 걸리며 Master CPU 가 망가지면 아무것도 할 수 없기 때문에 문제가 생긴다. 과거에는 AMP 방식만 사용하다 처리 방식이 발전이 되어 SMP 로 바뀌었다.

3. 마치며

Timer와 장치들의 인터럽트 및 처리방법에 대해 공부하였다.

Linux Linux Kernel Operating Systems