3장 64비트 프로세서의 이모저모

**[용어]**

**x86**

x86 is a family of backward compatible instruction set architectures[a] based on the Intel 8086 CPU and its Intel 8088 variant.

**Backward compatible <-> forward compatibility,**

In telecommunications and computing, a product or technology is backward compatible (BC)[1] or downward compatible if it can work with input generated by or meant to an older product or technology such as a legacy system.

**legacy system**

an old method, technology, computer system, or application program, "of, relating to, or being a previous or outdated computer system.

**Hardware interrupts**

are used by devices to communicate that they require attention from the operating system.[2] Internally, hardware interrupts are implemented using electronic alerting signals that are sent to the processor from an external device, which is either a part of the computer itself, such as a disk controller, or an external peripheral. For example, pressing a key on the keyboard or moving the mouse triggers hardware interrupts that cause the processor to read the keystroke or mouse position. Unlike the software type (described below), hardware interrupts are asynchronous and can occur in the middle of instruction execution, requiring additional care in programming. The act of initiating a hardware interrupt is referred to as an interrupt request (IRQ).

**A software interrupt**

is caused either by an exceptional condition in the processor itself, or a special instruction in the instruction set which causes an interrupt when it is executed. The former is often called a trap or exception and is used for errors or events occurring during program execution that are exceptional enough that they cannot be handled within the program itself. For example, if the processor's arithmetic logic unit is commanded to divide a number by zero, this impossible demand will cause a divide-by-zero exception, perhaps causing the computer to abandon the calculation or display an error message. Software interrupt instructions function similarly to subroutine calls and are used for a variety of purposes, such as to request services from low-level system software such as device drivers. For example, computers often use software interrupt instructions to communicate with the disk controller to request data be read or written to the disk.

3.1 운영모드

주변에서 많이 사용하는 인텔 64비트 호환 프로세서(이하 x86-64 프로세서)에는 크게 다섯가지 운영 모드가 있습니다. 우리에게 친숙한 16비트의 리얼 모드, 32비트의 보호 모드, 64비트의 IA-32e 모드와 조금 낯선 시스템 관리 모드, 가상 8086모드가 그것입니다. 각 모드에는 [표 3-1] 같은 특징이 있으며, 컨트롤 레지스터와 인터럽트라는 특수한 이벤트를 통해 각 모드로 전환할 수 있습니다.

[표 3-1] x86-64 프로세서의 운영 모드

운영모드 설명

리얼 모드 프로세서의 초기 상태로서 16비트 모드로 동작하며 8086 프로세서와 호환되는 모드

최대 1MB(2^20) 의 주소 공간을 지원

보호 모드 32비트 모드로 동작하며 세그먼트, 페이징, 보호, 멀티 태스킹 등의 기능을 제공하는 모드

4GB(2^32)의 주소 공간을 지원

IA-32e 모드 32비트 호환모드와 64비트 모드의 두 가지 서브 모드로 구성

16EB(2^64)의 주소 공간을 지원하는 모드

시스템 관리 모드 전원 관리나 하드웨어 제어 같은 특수 기능을 제공하는 모드

가상 8086 모드 보호 모드 내부에서 가상의 환경을 설정하여 리얼 모드처럼 동작하는 모드

64비트 모드를 지원하는 IA-32e 모드는 제조사에 따라 다르게 불려지고 있습니다. 인텔에서는 IA-32e 모드로 표기하지만, 인텔과 양대산맥을 이루는 AMD에서는 long mode로 표기합니다.

3.1.1 64비트 os가 갖추어야 할 필수 운영 모드

[표 3-1]의 다섯 가지 모드 중에서 64비트 OS가 반드시 지원해야하는 세 가지 모드는 리얼 모드, 보호 모드, IA-32e 모드 중 64비트 서브 모드 입니다. 시스템 관리 모드나 가상 8086 모드는 특수한 상황에서 사용되므로 os에서 해당 기능을 사용하지 않는다면 구현하지 않아도 괜찮습니다.

리얼 모드

프로세서가 어떤 상태 또는 모드에 있든 전원이 켜지거나 리셋되면 프로세서는 리얼모드로 진입합니다. 리얼 모드는 과거의 16비트 프로세서와 동일하게 동작하며, 이후에 설명할 BIOS의 여러 기능을 사용할 수 있습니다. BIOS는 디스크 읽기 및 쓰기 기능부터 그래픽 모드로 전환하는 기능까지 여러 가지 기능을 제공하며, 이를 사용하면 별도의 디바이스 드라이버를 제작하지 않고도 원하는 작업을 수행할 수 있습니다.

리얼 모드는 디바이스 드라이버를 제작하지 않아도 되는 장점이 있지만, OS를 처음 개발한다면 상당한 부담을 느낄 수도 있습니다. 리얼 모드에서 하는 작업은 OS 이미지를 디스크에서 메모리로 복사하여 보호 모드로 변경하는 것밖에 없지만, 대부분 작업을 어셈블리어로 처리해야 합니다.

보호 모드

보호 모드는 IA-32e 모드로 전환하려면 반드시 거쳐야 하는 모드로, 32비트 윈도우나 리눅스 os가 동작하는 기본 모드입니다. 최대 4G(2^32) 주소 공간을 제공하며 OS의 필수 기능으로 자리 잡은 보호, 멀티태스킹, 세그먼테이션, 페이징 등의 기능을 하드웨어적으로 지원합니다. 여러 기능을 제공하는 만큼 복잡하고, 레지스터와 자료구조가 다양합니다.

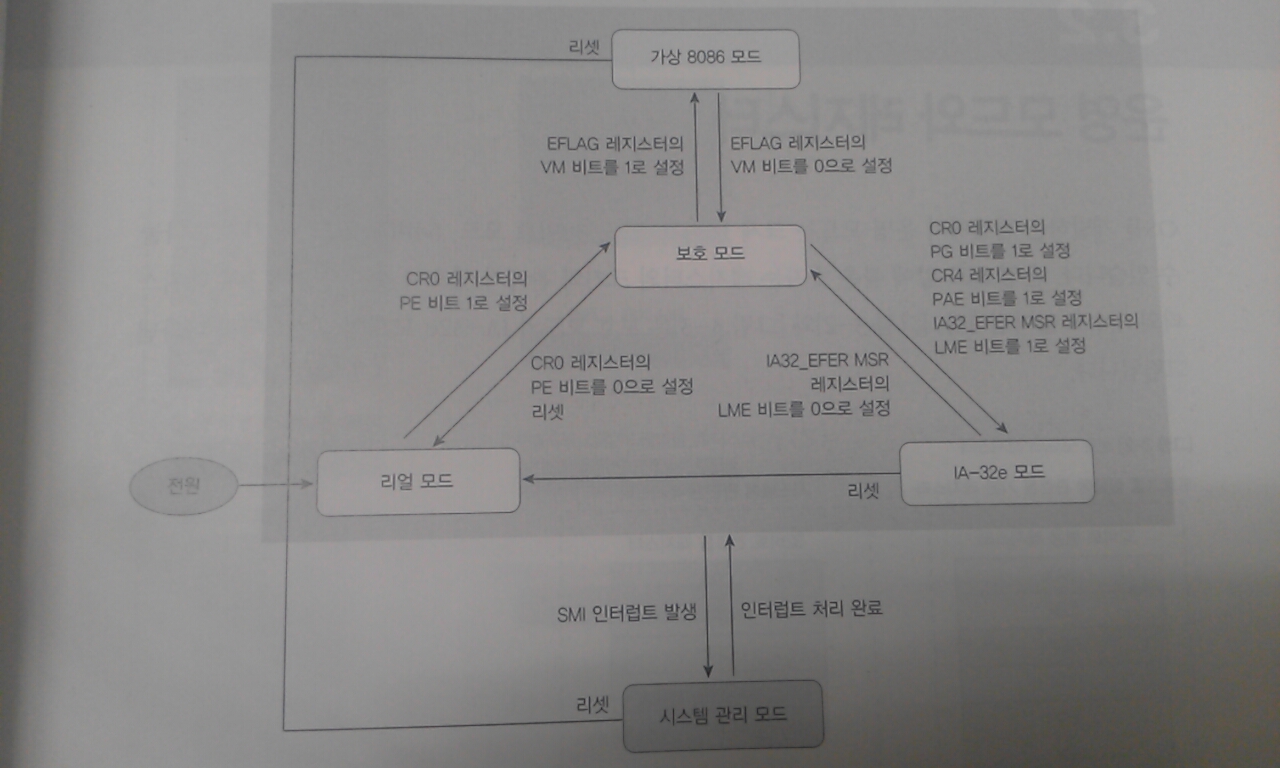
보호 모드에서 사용하는 레지스터는 대부분 IA-32e 모드에서도 같이 사용하므로 보호 모드를 이해하는 것이 매우 중요합니다. 그리고 윈도우와 리눅스를 포함한 대부분 os는 프로세서의 기능을 사용해서 특정 기능을 구현하므로 보호 모드에 대해 알아두면 os의 구조를 이해하는 데 도움이 됩니다.

IA-32e 모드

IA-32e 모드에서는 서브모드로 32비트 호환 모드와 64비트 모드가 있으며, 여러분이 대부분 시간을 투자할 운영 모드입니다. 32비트 호환 모드는 보호 모드와 같은 기능을 수행하므로 64비트 모드를 위주로 살펴보겠습니다. IA-32e 모드는 최대 16E(2^64)의 주소 공간을 제공하며 레즈스터 수도 보호 모드보다 많습니다. 보호 모드보다 더 많은 기능을 제공하므로 자료구조가 더 복잡하리라 생각할지도 모르겠습니다. 하지만, 대부분 자료구조는 보호 모드와 같거나 크기만 2배로 확장되고 일부 필드의 의미가 변하는 정도이므로 보호 모드와 큰 차이는 없습니다.

한 가지 재미있는 사실은 프로세서가 32비트 호환 모드일 때는 보호 모드에 있는 것처럼 동작하므로 32비트 코드를 그대로 실행할 수 있다는 점입니다. 이는 간단히 서브 모드만 변경함으로써 보호 모드 코드를 실행할 수 있다는 뜻입니다. 64비트 os에서 32비트 보호 모드 코드를 별다른 처리 없이 그대로 실행할 수 있는 것도 이러한 서브 모드를 사용하기 때문입니다.

3.1.2 운영 모드 사이의 관계와 운영 모드의 전환

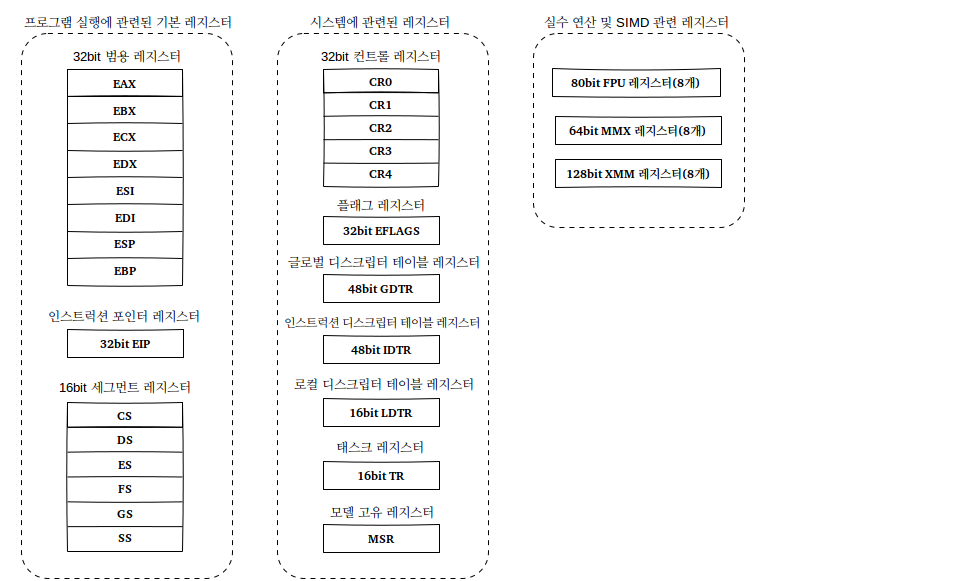


조건에는 컨트롤 레지스터와 인터럽트 발생만 표시되어 있지만, 기본적으로 각 모드에 필요한 자료구조는 미리 설정되어 있어야 합니다.

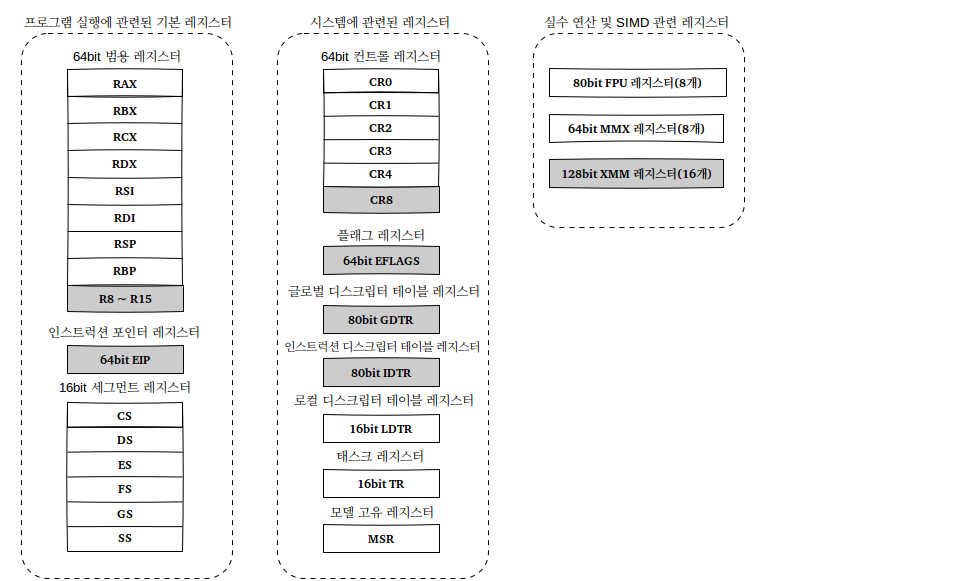
3.2 운영 모드와 레지스터

Os를 개발하는 관점에서 운영 모드는 크게 16비트 모드, 32비트 모드, 64비트 모드 세 가지로 나눌 수 있습니다. 운영 모드 앞에 붙은 숫자는 레지스터의 크기와 관계가 있으며 숫자가 커질수록 레지스터의 개수도 많아집니다.

[그림 3-2] 보호 모드의 레지스터



[그림 3-3] IA-32e 모드의 레지스터



Os를 개발하는 과정에서 큰 비중을 차지하는 레지스터는 범용 레지스터, 세그먼트 레지스터, 컨트롤 레지스터 세 가지입니다.

3.2.1 범용 레지스터

범용 레지스터(general purpose register)는 운영 모드와 가장 관계가 깊은 레지스터입니다. 범용 레지스터는 계산, 메모리 어드레스 지정, 임시 저장 공간 등의 목적으로 사용합니다. 운영 모드 앞에 붙는 숫자와 범용 레지스터의 크기는 대체로 일치하지만, 범용 레지스터의 수는 프로세서가 지원하는 운영 모드에 따라 다릅니다. 16비트와 32비트 모드를 지원하는 x86 계열은 8개, 64비트 모드를 지원하는 x86-64 계열은 16개의 범용 레지스터가 있습니다.

범용 레지스터의 수가 늘어나면 어떤 장점이 있을까요? 가장 대표적인 장점이 수행 속도의 개선입니다. 관련된 값을 레지스터에 모두 올려서 계산함으로써 메모리에 접근하는 시간을 줄이는 것이 가능합니다. 이와 비슷한 효과는 함수 호출에도 그대로 적용됩니다. 호출되는 함수가 작업을 처리하려면 정보가 필요하고, 이러한 정보는 호출하는 쪽에서 파라미터를 통해 넘겨줍니다. 다수의 범용 레지스터에 함수 파라미터를 넘겨줌으로써 스택 영역의 메모리에 접근하는 시간과 스택을 정리하는 시간을 줄일 수 있습니다.

이름에서 알 수 있듯이 범용 레지스터의 용도는 고정된 것이 아니며, 다양한 목적으로 사용될 수 있습니다. 하지만, 모든 상황에서 가능한 것은 아니므로 특정 명령어는 특정 레지스터와 같이 사용해야 합니다. (의미: 레지스터가 특수한 용도로 사용되는 예로 디바이스 드라이버 코드에서 찾을 수 있습니다. Pc에 연결된 디바이스를 제어하려면 장치가 연결된 I/O 어드레스에 접근해야 합니다. 이때 I./O 어드레스를 지정할 수 있는 레지스터는 DX를 통해서만 가능합니다. 정수 곱셉 코드의 경우 넘겨 받는 오퍼랜드가 하나밖에 없습니다. 곱셈 명령이 다른 하나의 오퍼랜드로 AX레지스터를 사용하기 때문입니다. 곱셉 명령은 AX와 오퍼랜드를 곱한 후, 그 결과를 DX:AX 혹은 AX에 저장하도록 설계되었습니다.) 범용 레지스터에는 개별적인 이름이 있는데, 이 이름을 통해 각 레지스터의 특수한 용도를 알 수 있습니다. X86-64 프로세서는 x86 프로세서에서 가지는 범용 레지스터 외에도 R8~R15로 이름 붙여진 8개의 레지스터가 더 있습니다. 이 레지스터의 기능은 다른 범용 레지스터와 같으며 차이점이라면 특수한 용도가 정의되지 않았다는 것입니다.

[표 3-2] x86-64 프로세서의 범용 레지스터와 용도

범용 레지스터 이름 용도

AX 산술 연산을 수행할 때 누산기로 이용

BX 데이터의 어드레스를 지정할 때 데이터 포인터로 사용

CX 루프 또는 문자열의 카운터로 사용

DX I/O 어드레스를 지정할 떄 사용되며, 산술 연산을 수행할 떄 보조 레지스터로 사용

SI 문자열에 관련된 작업을 수행할 때 원본 문자열의 인덱스로 사용

DI 문자열에 관련된 작업을 수행할 때 목적지 문자열의 인덱스로 사용

SP 스택의 포인터로 사용

BP 스택의 데이터에 접근할 때 데이터 포인터로 사용

R8~R15 x86-64 프로세서에서 추가된 범용 레지스터로, 다양한 용도로 사용 가능

범용 레지스터의 크기는 프로세서의 운영 모드와 관련이 있다고 했습니다. 그렇다면, x86-64 프로세서의 내부에는 운영 모드 별로 레지스터가 개별적으로 있을 까요? NO!!! 실제 x86-64 프로세서는 64비트 크기의 범용 레지스터만 있으며 보호 모드나 리얼 모드일 때 레지스터의 일부 영역만 사용하도록 처리하고 있습니다. 64비트 범용 레지스터는 하위 32비트, 16비트, 8비트의 크기로 구분하여 접근할 수 있고, 레지스터에 접두사(prefix)나 접미사(postfix)를 붙여 접근하는 크기를 표시합니다.

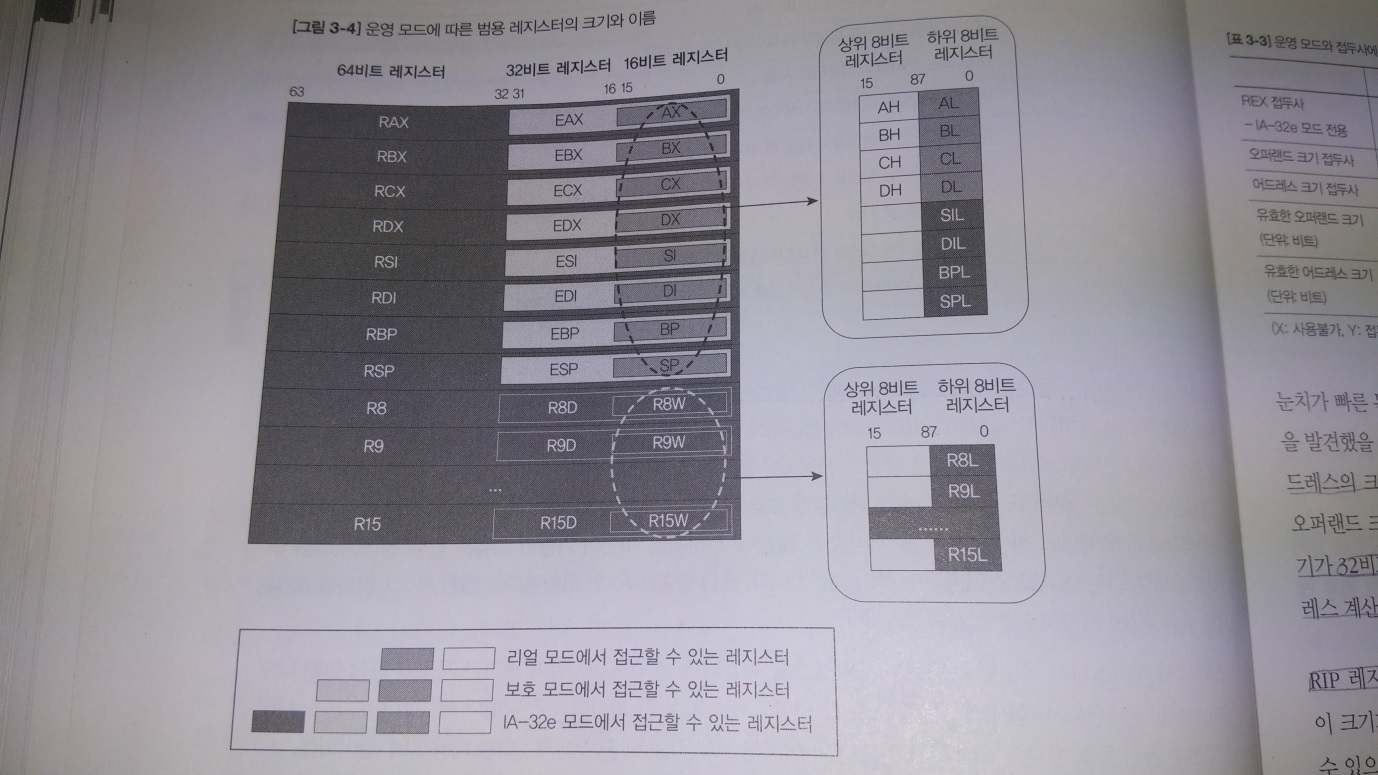
64비트 레지스터 - RAX, RBX처럼 R 접두사

32비트 레지스터 – EAX,EBX,R8D,R9D처럼 E 접두사나 D 접미사

16비트 레지스터 – AX,BX,R8W,R9W처럼 접두사가 붙지 않거나 W 접미사

상위 8비트 레지스터 – AH,BH처럼 끝자리가 H로 변경

하위 8비트 레지스터 – AL,BL,SIL,DIL,R8L,R9L처럼 끝자리가 L로 변경되거나 접미사 L



[그림 3-4]에는 리얼 모드에서 16비트 크기 이하의 레지스터만 접근 가능한 것으로 표시되어 있지만, 오퍼랜드 크기 접두사(operand-size prefix,0x66)를 사용하거나 어드레스 크기 접두사 (address-size prefix,0x67)를 사용하면 32비트 레지스터도 접근할 수 있습니다. 하지만, 리얼 모드에서 이러한 접두사를 사용하는 경우는 보호 모드로 전환하는 것처럼 특수한 상황이므로 리얼 모드에서는 16비트 이하의 크기를 가지는 레지스터에만 접근한다고 생각해도 괜찮습니다. 마찬가지로 보호 모드에서 16비트 레지스터에 접근할 때도 어드레스 크기 접두사와 오퍼랜드 크기 접두사를 사용하며, IA-32e 모드의 경우는 기존의 두 가지 접두사와 함께 새롭게 추가된 REX 접두사를 사용함으로써 다양한 크기의 레지스터를 사용할 수 있습니다.

운영 모드에 따라 접두사를 결합하는 방법에는 일정한 규칙이 있으며, 사용한 접두사에 따라 명령어(instructon)가 처리하는 오퍼랜드나 어드레스의 크기가 달라집니다.

[표 3-3] 운영 모드와 접두사에 따른 오퍼랜드 및 어드레스의 크기

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 리얼 모드 | | | | 보호 모드 | | | | IA-32e 모드 (64비트 서브모드) | | | | | | | |
| REX접두사  -IA-32e모드 전용 | X | X | X | X | X | X | X | X | N | N | N | N | Y | Y | Y | Y |
| 오퍼랜드 크기 접두사 | N | N | Y | Y | N | N | Y | Y | N | N | Y | Y | N | N | Y | Y |
| 어드레스 크기 접두사 | N | Y | N | Y | N | Y | N | Y | N | Y | N | Y | N | Y | N | Y |
| 유효한 오퍼랜드 크기  (단위:비트) | 16 | 16 | 32 | 32 | 32 | 32 | 16 | 16 | 32 | 32 | 16 | 16 | 64 | 64 | 64 | 64 |
| 유효한 어드레스 크기  (단위:비트) | 16 | 32 | 16 | 32 | 32 | 16 | 32 | 16 | 64 | 32 | 64 | 32 | 64 | 32 | 64 | 32 |

(X:사용불가, Y:접두사 사용,N:접두사 사용 안함)

눈치가 빠른 독자라면 [표3-3]을 보면서 IA-32e 모드의 기본 오퍼랜드 크기가 32비트로 표시된 것을 발견했을 것입니다. IA-32e 모드의 64비트 서브 모드는 64비트 모드이므로 오퍼랜드의 크기와 어드레스의 크기가 모두 64비트인 것이 당연합니다. 하지만, 어떤 이유에서인지 IA-32e 모드의 기본 오퍼랜드 크기는 32비트, 기본 어드레스 크기는 64비트로 설계되었습니다. 그리고 기본 오퍼랜드 크기가 32비트로 설계되어 64비트 어드레스를 표현할 수 없으므로 RIP 상대 어드레스라는 새로운 어드레스 계산방식이 도입되었습니다.

RIP 레지스터는 현재 수행 중인 명령의 어드레스를 가리키는 레지스터입니다. 이름에서 알 수 있듯이 크기가 64비트입니다. RIP 레지스터의 값과 32비트 오퍼랜드를 통해 64비트 주소 공간을 나타낼수 있으며, 이것이 RIP 상대 어드레스 방식의 어드레스 지정방법입니다. 하지만, 기본 오퍼랜드의 크기가 32비트이므로 RIP 레지스터의 값에 상위 2G와 하위 2G 범위까지만 표현할 수 있는 단점이 있습니다. 그렇다면 범위를 벗어나는 어드레스는 어떻게 접근해야 할까요? RIP의 위치에서 2GB범위 마다 점프(jump) 코드를 넣어 RIP 상대 어드레스로 접근하도록 해야 할까요? 만약 그렇게 한다면 코드는 중간 중간에 삽입된 점프 코드로 인해 뒤죽박죽 되어버릴 것입니다. 그렇다면 어떻게 해야 할까요? 프로세서 제조사에서는 이런 경우를 대비하여 무조건 분기 명령어(jmp)에 예외를 두었습니다. 즉, 무조건 분기 명령어의 오퍼랜드의 크기는 기본 64비트로 하여 전체 어드레스 범위에서 이동이 가능하도록 한 것입니다.

3.2.2 세그먼트 레지스터

세그먼트 레지스터(segment register)는 16비트 레지스터로 어드레스 영역을 다양한 크기로 구분하는 역할을 합니다. 세그먼트 레지스터의 주된 역할은 어드레스 영역의 구분이지만, 모드마다 조금씩 역할에 차이가 있습니다.

리얼 모드 – 단순히 고정된 크기의 어드레스 영역을 지정하는 역할

보호 모드와 IA-32e 모드 – 접근 권한(privilege level), 세그먼트 시작 어드레스와 크기등을 지정하는데 사용된다. 이러한 기능을 이용하면 응용프로그램으로부터 커널 영역을 보호하는 기능을 쉽게 구현할 수 있습니다.

세그먼트 레지스터는 CS,DS,SS,ES,FS,GS 총 6개로 구성됩니다.

[표 3-4] x86-64 프로세서의 세그먼트 레지스터

세그먼트 레지스터 이름 설명

CS 코드 영역을 가리키는 레지스터

데이터 이동 명령으로 값을 변경할 수 없으며, 점프명령이나 인터럽트 관련 명령으로 변경 가능

DS 데이터 영역을 가리키는 레지스터

ES 데이터 이동 명령으로 값을 변경할 수 있음

FS DS 레지스터는 데이터 영역에 접근할 때 암시적으로 사용됨

GS ES 레지스터는 문자열과 관련된 작업을 처리할 때 암시적으로 사용됨

데이터 영역에 접근하면서 DS 레지스터 이외의 세그먼트 레지스터를 사용하려면 세그먼트 레지스터 접두사 사용

SS 스택 영역을 가리키는 레지스터

데이터 이동 명령으로 값을 변경할 수 있음

스택 관련 레지스터(SP,BP)를 통해 스택에 접근할 때 임시적으로 사용됨

세그먼트 레지스터에는 DS,ES,FS,GS와 같이 총 4개의 데이터 관련 세그먼트 레지스터가 있으며, 세그먼트 레지스터 접두사를 통해 명시적으로 특정 세그먼트를 설정할 수 있습니다. 스택 관련 작업을 수행할 때 역시 세그먼트 레지스터 접두사를 통해 SS 레지스터가 아닌 기타 세그먼트 레지스터(CS 레지스터 제외)를 설정할 수 있습니다. 명시적으로 지정하지 않으면 기본적으로 사용되는 세그먼트 레지스터는 [표3-4]에서 보는 것과 같습니다.

세그먼트 레지스터의 역할은 주소 공간을 목적에 따라 구분하는 것이며, 주소 공간을 구분하는 방법은 메모리 관리 기법과 깊은 관계가 있습니다.

3.2.3 컨트롤 레지스터

컨트롤 레지스터(control register)는 운영 모드를 변경하고, 현재 운영 중인 모드의 특정 기능을 제어하는 레지스터입니다. X86 프로세서에는 CR0,CR1,CR2,CR3,CR4의 5개의 컨트롤 레지스터가 존재하며, x86-64 프로세서에는 CR8이 추가되어 총 6개의 컨트롤 레지스터가 있습니다. 각 컨트롤 레지스터의 역할은 [표 3-5]와 같습니다.

[표 3-5] x86-64 프로세서의 컨트롤 레지스터

컨트롤 레지스터 이름 설명

CR0 운영 모드를 제어하는 레지스터

리얼 모드에서 보호 모드로 전환하는 역할과 캐시,페이징 기능 등을 활성화시킴

CR1 프로세서에 의해 예약된 레지스터

CR2 페이지폴트 발생시 페이지 폴트가 발생한 선형 주소가 저장되는 레지스터

페이징 기법을 활성화한 후에는 페이지 폴트 발생시만 유효한 값을 가짐

CR3 페이지 디렉터리 물리 주소와 페이지 캐시에 관련된 기능을 설정하는 레지스터

CR4 프로세서에 지원하는 각종 확장 기능을 제어한느 레지스터

페이지 크기 확장이나 메모리 영역 확장 등의 기능을 활성화시킴

CR8 태스크 우선순위 레지스터의 값을 제어하는 레지스터

프로세서 외부에서 발생하는 인터럽트를 걸러주는 필터의 역할

IA-32e 모드에서만 접근 가능

컨트롤 레지스터는 리얼 모드와 보호 모드일 때 32비트 크기이며, IA-32e 모드에서는 64비트로 확장되지만 일부 제약사항이 있습니다. CR0와 CR4, CR8 레지스터에서는 64비트 중 상위 32비트를 0으로 설정해야 합니다. CR2 레지스터의 경우는 64비트 영역을 모두 사용할 수 있으며, CR3 레지스터는 비트 40부터 비트 51까지 모두 0으로 설정해야 합니다.

컨트롤 레지스터는 프로세서의 운영 모드와 확장 기능을 제어하는 레지스터인 만큼 구조가 복잡합니다. 각 필드는 저마다 특정 기능을 활성화/비활성화하며, 현재 운영 모드에 따라 필수 필드와 옵션 필드가 달라집니다. 컨트롤 레지스터의 특정 기능은 해당 비트를 1로 설정해도 충분하지만, 특정 기능을 1로 설정하기 전에 프로세서가 사용할 자료구조를 미리 준비해야 합니다. 혹시 PC를 가장 손쉽게 재부팅하는 방법을 찾고 있다면 CR0 레지스터에 아무 값이나 설정해보거나 비트 0의 값을 변경하는 방법을 추천합니다. 비트 0은 리얼 모드와 보호 모드를 전환하는 flag이며, 값을 변경하기 전후에 별도로 처리해야 하는 대표적인 예입니다.

**[용어]**

**1. 가상 주소**

|  |  |
| --- | --- |
| 세그먼트 레지스터 (16bit) | 메모리 주소 (Virtual Address) |

[\*] 세그먼트 레지스터 -> GDT 또는 LDT의 Segment Descriptor 을 참조 (GDTR -> GDT, LDTR-> LDT)

- GDT : 모든 프로그램이 참조 가능한 세그먼트 디스크립터 테이블

- LDT : 태스크당 독립적으로 세그먼트 디스크립터 테이블 정의

- 세그먼트 레지스터가 참조한 세그먼트 디스크립터의 **Base Bits와 가상 주소를 더해서 선형 주소**를 구하게 된다.

**(이를 Sementation이라고 함)**

**\* Segment Register (16bit)**

|  |  |  |
| --- | --- | --- |
| INDEX (13bit) | TI (1bit) | RPL (2bit) |

- INDEX는 디스크립터 테이블 배열의 index 값으로 사용.

- TI bit가 0이면 GDT 사용, 1이면 LDT 사용.

- RPL은 디스크립터를 통해 요구할 때의 특권 레벨.

**\* Segment Descriptor  (64bit)**

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Base Bits  (8bit) | G | D | X | U | Limit Bits | P | DPL | S | TYPE | A | Base Bits  (24bit) | Limit Bits |

- Base Bits 총 32bit로 가상 주소와 더해서 선형 주소를 구한다.

- Limit Bits 총 20bit로 세그먼트가 허용하는 메모리 범위 설정. (G bit가 1일 경우 4KByte 단위로 해서 20bit로 32bit의 모든 메모리 표현 가능)

**\* GDTR (48bit)**

|  |  |
| --- | --- |
| 디스크립터 시작 위치 (32bit) | Limit (16bit) |

- limit specify the size of the table in bytes ( 16bit인 이유: segment는 총 2^13개가 존재하는데 각각 2^3(8)bytes의 크기를 가진다.)

**\* LDTR (16bit)**

|  |  |  |
| --- | --- | --- |
| INDEX | TI | RPL |

**-**세그먼트 레지스터와 같은 형식으로 동작.

- GDT를 사용한다면, 세그먼트 레지스터의 인덱스로 GDT entry를 구함.

- LDT를 사용한다면, GDTR의 GDT에서 LDTR의 인덱스로 LDT의 base address를 구함. 세그먼트 레지스터의 인덱스로 LDT entry를 구함.

**2. 선형 주소**

|  |  |  |
| --- | --- | --- |
| DIR (10 bit) | Page (10 bit) | Offset (12 bit) |

- 위에서 Segmentation 과정을 거쳐서 구한 선형 주소를 위와 같이 표현된다.

- 이제 선형 주소는 Paging 과정을 거쳐 물리주소로 접근하게 된다. 페이징 과정은 아래와 같다.

[\*] CR3 : 프로세스 별로 고유한 값을 가짐.    -> Page Directory를 가리킴

[\*] DIR : Page Directory 에서의 Offset 값.    -> Page Table를 가리킴

[\*] Page : Page Table에서의 Offset 값.       -> 해당 페이지의 물리 메모리를 가리킴

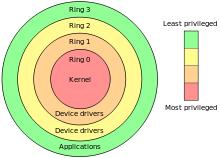
[\*] Offset : 물리 메모리 에서의 Offset 값.     -> 드디어 물리 메모리에 접근

==> CR3 -> [Page Directory] + DIR -> [Page Table] + Page -> [Physical Address] + Offset

디스크립터 - 메모리 영역의 정보를 저장하는 자료구조

Privilege level

A privilege level in the x86 instruction set controls the access of the program currently running on the processor to resources such as memory regions, I/O ports, and special instructions. There are 4 privilege levels ranging from 0 which is the most privileged, to 3 which is least privileged. Most modern operating systems use level 0 for the kernel/executive, and use level 3 for application programs. Any resource available to level n is also available to level 0..n, so the privilege levels are "rings"



3.3 운영 모드와 메모리 관리 기법

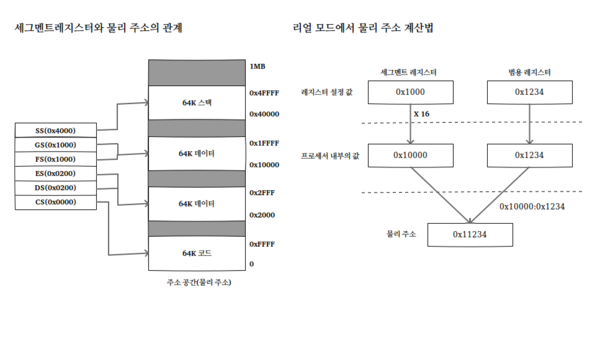
X86과 x86-64 프로세서에서 지원하는 메모리 관리 기법은 크게 두 가지입니다. 한 가지는 세그먼테이션이고 다른 한가지는 페이징입니다. 세그먼테이션과 페이징은 주소 공간을 특정 영역으로 나눈다는 공통점이 있지만, 나누는 방식에 차이가 있습니다. 세그먼테이션을 여러분이 원하는 크기로 잘라 먹을 수 있는 케이크에 비유한다면, 페이징은 정해진 크기로 잘라져 있는 식빵에 비유할 수 있습니다.

메모리 관리 기법을 사용하려면 관련 레지스터에 특정한 자료구조를 설정해야 합니다. 세그먼테이션은 세그먼트 레지스터에 세그먼트의 시작 주소 혹은 디스크립터라고 불리는 자료구조의 위치를 설정해야 합니다. 페이징은 컨트롤 레지스터 중에 CR3 레지스터에 페이지 디렉터리라고 불리는 자료구조의 물리 주소를 설정해야 사용할 수 있습니다. 모든 운영 모드에 공통으로 메모리 관리 기법을 적용하면 좋겠지만, 모든 운영 모드가 세그먼트와 페이징을 지원하는 것은 아닙니다. 모드에 따라 지원하지 않거나 지원하지만 일부 기능이 제한되는 경우도 있습니다. 또한 같은 메모리 관리 기법을 사용하더라도 모드에 따라 필드의 의미가 달라지는 경우도 있습니다. 이러한 차이는 x86-64 프로세서가 기존 프로세서와 호환성을 유지하면서 발전했기 때문입니다. 다행인 점은 기존의 줄기를 완전히 벗어날 정도의 차이는 아니어서 하나만 이해하면 나머지도 그다지 어렵지 않습니다.

3.3.1 리얼 모드의 메모리 관리 방식

리얼 모드는 최대 1MB까지 주소 공간을 사용하며 세그먼테이션만 지원합니다. 리얼 모드에서 세그먼트의 크기는 64K로 고정이고, 세그먼트의 시작 어드레스는 세그먼트 레지스터에 직접 설정합니다. 세그먼테이션에서 세그먼트의 시작 어드레스는 코드나 메모리에 접근할 때 기준 어드레스(Base address)로 사용됩니다.

[그림 3-7] 세그먼트 레지스터와 세그먼트, 물리 주소의 관계 [그림 3-8] 리얼 모드에서 물리 주소 계산법



리얼 모드는 페이징을 사용하지 않으므로 물리 주소로 변환하는 방식이 비교적 간단합니다. 세그먼테이션을 거쳐 나온 어드레스가 바로 물리 주소가 됩니다. 리얼 모드의 세그먼테이션은 세그먼트 레지스터의 값에 범용 레지스터의 값을 더하는 방식으로 동작합니다. 세그먼트 레지스터의 크기도 16비트이고 범용 레지스터의 크기도 16비트인데, 어떻게 최대 1MB 영역까지 접근할 수 있을까요? 비밀은 세그먼트 레지스터에 있습니다. 세그먼트 레지스터의 값을 그대로 범용 레지스터에 더하는 것이 아니라, 세그먼트 레지스터의 값에 16 곱한 값을 세그먼트의 기준 주소로 사용하는 것입니다. 16비트의 최댓값은 65535(2^16-1)이며, 여기에 16(2^4)을 곱하면 세그먼트의 기준 주소는 거의 1MB에 근접하는 1048560(2^20-16)가 됩니다. 이 값에 범용 레지스터를 더한 값이 실제 물리 주소가 되므로 1MB까지 접근할 수 있습니다. [그림 3-8]은 세그먼트 레지스터에 0x1000, 범용 레지스터에 0x1234가 설정되었을 때 물리 주소를 계산하는 방법을 나타낸 것입니다.

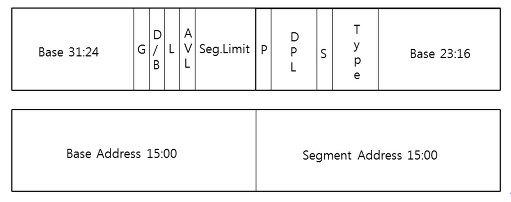
세그먼트의 크기가 64KB인 이유는 바로 범용 레지스터의 크기 때문입니다. 16비트 프로세서에는 32비트 레지스터가 없으며 범용 레지스터는 모두 16비트 크기입니다. 따라서 16비트로 접근할 수 있는 범위가 0~65535(0xffff)이므로 세그먼트 크기도 64KB가 된 것입니다. 리얼 모드의 이러한 세그먼테이션 기법은 보호 모드에도 영향을 주었습니다. 보호 모드 역시 같은 방식으로 세그먼트의 기준 주소에 범용 레지스터를 더해 계산합니다. 다만, 페이징이 추가되어서 계산 결과는 물리 주소가 아닌 선형 주소라고 불리는 논리 주소로 바뀌었고, 선형 주소는 페이징을 거쳐 물리 주소로 바뀌게 됩니다.

3.3.2 보호 모드의 메모리 관리 방식

보호 모드는 리얼 모드와 달리 세그먼테이션과 페이징을 모두 지원합니다. 보호 모드의 세그먼테이션은 리얼 모드의 세그먼테이션보다 많은 기능을 제공합니다. 또한 보호 모드의 세그먼테이션은 세그먼트 레지스터에 세그먼트의 기준 주소를 직접 설정하는 대신 디스크립터 자료구조의 offset를 설정하는 방식으로 바뀌었습니다. 세그먼트 레지스터의 명칭도 세그먼트 디스크립터를 선택한다는 의미에서 세그먼트 셀렉터로 변경되었습니다.

디스크립터는 메모리 영역의 정보를 저장하는 자료구조로 여러 종료가 있으며, 그중에서 세그먼트에 대한 정보를 나타내는 디스크립터를 세그먼트 디스크립터라고 부릅니다. 세그먼트 디스크립터에는 세그먼트의 시작 어드레스와 크기, 권한(privilege), 타입(type) 등의 정보가 있습니다. ([그림 3-9] 참조) 세그먼트 디스크립터에 포함된 특권 레벨(DPL, descritor privilege level)은 해당 세그먼트에 접근하기 위한 최소한의 권한을 나타내며, 특권 레벨은 0~3 사이의 값을 가집니다. 숫자가 작을수록 권한이 높고, 세그먼트에 접근하려면 현재 수행중인 특권 레벨(CPL, current privlege level)이 적어도 디스크립터에 설정된 권한과 같거나 높아야(작은 숫자여야) 합니다. 조건을 만족하지 않으면 프로세서는 예외를 발생시켜 문제가 발생했음을 알립니다. 접근하는 어드레스가 세그먼트의 크기를 넘어서는 경우도 역시 예외가 발생합니다. 예외는 프로세서가 코드를 실행하는 도중 예기치 못한 문제가 발생했을 때 실행을 중단하고 발생시키는 특정 이벤트 같은 것입니다.

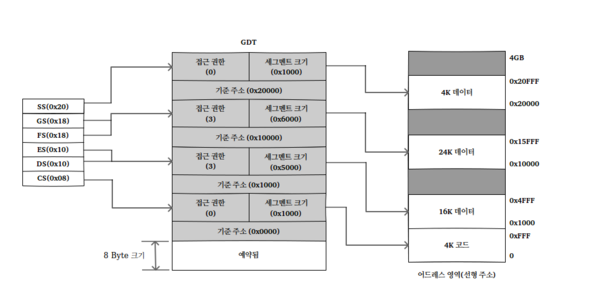
[그림 3-9] 세그먼트 디스크립터의 구조



보호 모드에서 세그먼트 레지스터는 세그먼트 디스크립터의 위치를 가리킵니다. 세그먼트 디스크립터는 메모리 상에 위치하는 자료구조의 일종으로 GDT(global descriptor table)라고 불리는 곳에 모여 있습니다. GDT는 연속된 디스크립터의 집합이며, 최대 8192개의 디스크립터를 포함할 수 있는 테이블 형태의 자료구조입니다. GDT 역시 메모리에 위치하는 자료구조에 불과하므로 프로세서에 GDT의 위치를 직접 알려야 합니다. GDT의 위치와 관련있는 레지스터는 GDTR(global descripotr table register) 레지스터이며, 16비트 GDT 크기 필드와 32비트 기준 주소 필드로 구성된 자료구조의 물리 주소를 넘겨받습니다. 프로세서는 이 값을 내부에 저장했다가 세그먼트 셀렉터를 통해 어드레스에 접근할 때마다 GDT의 위치를 찾는데 참조합니다.

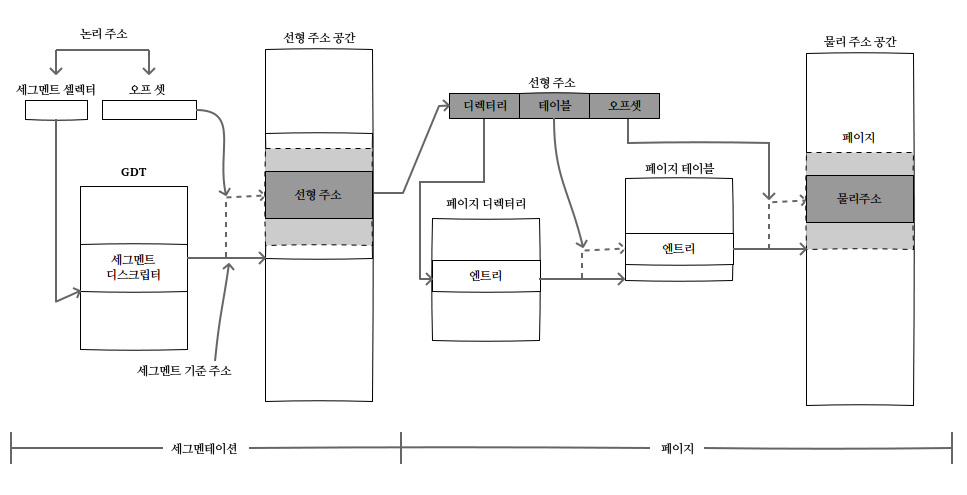
보호 모드에서 주소를 계산하는 방법은 리얼 모드와 마찬가지로 세그먼트 레지스터의 기준 주소에 범용 레지스터의 값을 더해서 구합니다. 이렇게 계산한 결과가 선형 주소이며 프로세서는 실제 메모리에 접근할 떄 선형 주소를 기반으로 물리 주소를 계산합니다. 보호 모드에서는 리얼 모드와 달리 세그먼트의 크기를 지정할 수 있으므로, 세그먼트의 크기가 64KB로 고정된 리얼 모드와 다소 차이가 있습니다. 세그먼트 크기는 해당 세그먼트의 어드레스에 접근할 때 참조하며, 기준 주소에 더해질 값(범용 레지스터의 값 또는 특정 사웃 값)은 세그먼트의 크기(세그먼트 디스크립터에 정의한 크기)를 넘을 수 없습니다. 더해지는 값이 세그먼트의 크기를 넘게 된다면 프로세서는 세그먼트의 접근 권한을 위반한 경우와 마찬가지로 예외를 통해 오류가 발생했음을 알려줍니다.

[그림 3-10] 보호 모드에서 세그먼트 셀렉터, 세그먼트 디스크립터, 선형 주소의 관계



리얼 모드와 달리 세그먼테이션을 거쳐 나온 선형 주소는 물리 주소와 일치할 수도, 그렇지 않을 수도 있습니다. 선형 주소는 이후에 나올 페이징의 입력 값이 되며, 페이징을 사용하지 않는다면 선형 주소는 물리 주소와 1:1로 대응합니다. 우리가 만들 os처럼 작고 간단한 os는 대부분 페이징을 사용하지 않거나 사용하더라도 선형 주소와 물리 주소가 1:1로 대응하도록 설정합니다. 선형 주소와 물리 주소를 1:1로 대응시켜 얻는 이점은 os의 메모리 관리 기능을 작고 가볍게 유지할 수 있고, 메모리 구조가 직관적이어서 디버깅이 편하다는 점입니다.

[그림 3-11] 세그먼테이션과 페이징의 관계



페이징은 물리 메모리를 페이지(page)라고 불리는 일정한 크기로 나누고, 선형 주소와 물리 주소를 나눠 놓은 페이지로 연결하는 방식을 말합니다. 페이징을 사용하면 물리 메모리 크기보다 더 큰 영역의 선형 주소도 물리 페이지만 연결하면 사용이 가능하므로, 주소 공간을 더 넓게 사용할 수 있는 장점이 있습니다. 또한 같은 물리 페이지를 여러 선형 주소에 연결함으로써 응용프로그램끼리 공유하는 메모리를 쉽게 처리할 수 있습니다. 반대로 응용프로그램마다 독립적인 주소 공간을 보장하고 싶다면 페이징 자료구조를 따로따로 생성하고 물리 메모리에 중복되지 않게 연결해주면 됩니다.

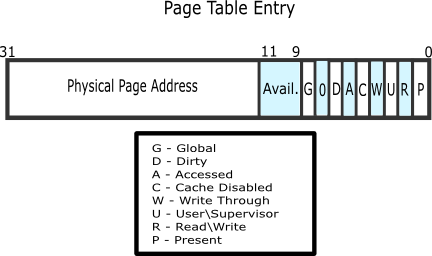
최신 os는 페이징의 이러한 기능을 이용하여 으용프로그램에 독립된 주소 공간 보장, 공유 메모리, 요구 페이징, 공유 라이브러리 등의 기능을 구현하고 있습니다.

보호 모드에서 페이징은 페이지 크기에 따라 크게 두 가지 방식으로 구분합니다. 하나는 물리 메모리를 4KB 크기로 나누고 선형 주소를 3단계로 구분하는 방식이며, 다른 하나는 물리 메모리를 4MB 크기로 나누고 선형 주소를 2단계로 구분하는 방식입니다. 두 가지 방식은 페이지의 크기와 구분 단계만 다를 뿐이지 기본 원리는 같으므로 3단계로 구분하는 페이징으로 설명하겠습니다. 물론 프로세서에서 지원하는 물리 주소 확장 기능을 사용하면 4KB 페이지로 4단계, 2MB 페이지로 3단계로 설정할 수 있지만 기본 원리 설명에는 4KB의 3단계 구분으로 충분합니다.

3단계 페이징은 선형 주소를 디렉터리, 테이블, 오프셋 세 부분을 나누며 물리 메모리를 4KB 페이지로 나누어 관리하는 방식입니다. 선형 주소의 디렉터리 부분과 테이블 부분은 각기 페이지 디렉터리와 페이지 테이블에 있는 엔트리의 위치를 나타냅니다. 페이지 디렉터리와 페이지 테이블은 앞서 설명한 GDT와 마찬가지로 메모리 공간에 있는 자료구조일 뿐입니다. 프로세서가 페이징 처리 과정에서 해댕 테이블을 사용하려면 우리가 직접 위치를 알려줘야 합니다. GDT에는 GDTR 레지스터가 있듯이 페이징에는 CR3 컨트롤 레지스터가 있습니다. CR3 컨트롤 레지스터에는 페이지 디렉터리의 시작 주소를 가리키며 페이지 디렉터리 엔트리의 위치 계산에 사용합니다.

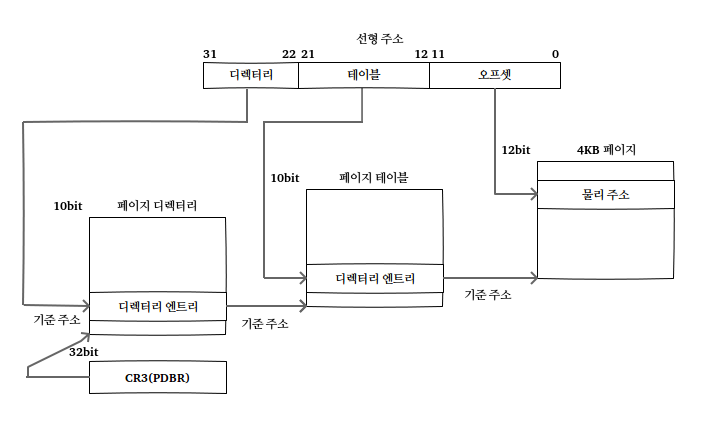
페이지 디렉터리 엔트리와 페이지 테이블 엔트리는 크기가 모두 4바이트입니다. 페이지 크기가 최소 4KB이므로 비트 12~비트 31로 기준 주소를 나타내고 비트 11~비트0은 속성 필드로 사용합니다. [그림 3-13]은 두가지 엔트리 중에서 페이지 테이블 엔트리입니다. 페이지 디렉터리 엔트리 역시 하위 12비트가 속성 필드로 할당되어 있으며, 몇몇 필드를 제외하면 페이지 테이블 엔트리와 거의 같습니다. 속성 필드 중에 유심히 봐야 할 필드는 U/S(User/Supervisor) 필드입니다. U/S 필드는 해당 페이지에 접근할 수 있는 권한을 나타냅니다. 0으로 설정하면 유저 애플리케이션 레벨(3)을 제외한 모든 레벨에서 접근 가능합니다. 1로 설정하면 유저 어플리케이션 레벨 이상에서 접근 가능함을 나타내며 모든 레벨에서 접근이 가능합니다. 페이지의 보호 기능과 세그먼테이션의 보호 기능을 조합하면 메모리 모델을 단순하게 유지하면서 커널 영역과 유저 영역을 구분하는 것이 가능합니다.

[그림 3-13] 보호 모드의 페이지 테이블 엔트리 구성



선형 주소는 최상위 비트부터 디렉터리 오프셋 10비트, 테이블 오프셋 10비트, 페이지 오프셋 12비트로 구분합니다. 디렉터리와 테이블의 오프셋이 10비트이므로 페이지 디렉터리와 페이지 테이블의 엔트리 수는 총 1024(2^10)개 입니다. 선형 주소의 마지막에 있는 페이지 오프셋은 12비트이므로 최댓값은 4KB(4096,2^12)이며, 12비트인 이유는 4KB 페이지의 오프셋을 나타내기 때문입니다. 페이지 디렉터리 엔트리와 페이지 테이블 엔트리 값은 각자 다음에 위치하는 페이지 테이블의 시작 주소와 페이지의 시작 주소를 나타내며 이 값에 선형 주소의 오프셋을 더해 물리 주소를 구합니다.

[그림 3-14] 보호 모드에서 3단계 페이징과 어드레스 변환 과정



[그림 3-14]를 바탕으로 선형 주소에서 물리 주소를 구하는 과정을 아래에 간략히 정리했습니다.

1. CR3 레지스터에 설정된 어드레스로 페이지 디렉터리의 시작주소를 찾아감
2. 페이지 디렉터리의 시작 주소에 선형 주소의 디렉터리 오프셋을 이용, 해당 디렉터리 엔트리를 찾음

(디렉터리 엔트리에 설정된 값이 페이지 테이블의 시작 주소)

1. 페이지 테이블의 시작 주소에 선형 주소의 테이블 오프셋을 이용, 해당 페이지 테이블 엔트리를 찾음

(페이지 테이블 엔트리에 설정된 값이 4KB 페이지의 시작 주소)

1. 페이지의 시작 주소에 선형 주소의 페이지 오프셋 값을 이용, 실제 물리 주소 변환

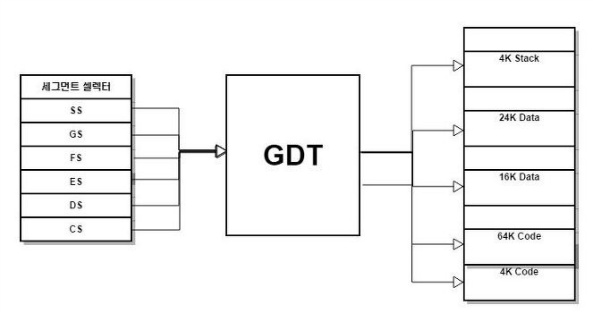
3.3.3 IA-32e 모드의 메모리 관리

IA-32e 모드는 두 가지 서브 모드, 호환 모드와 64비트 모드가 있습니다. 호환 모드는 보호 모드와 동작이 같으므로 생략하고, 64비트 모드만 설명합니다. IA-32e 모드의 64비트 모드(이하 IA-32e 모드)는 64비트이므로 사용 가능한 최대 어드레스 2^64, 즉 16EB까지입니다. 기존의 보호 모드보다 1,000,000,000배 이상의 공간을 자랑하지만, 다행히도 IA-32e 모드의 메모리 관리 기법은 보호 모드와 기본적인 부분이 거의 같습니다.

IA-32e 모드의 세그먼테이션은 보호 모드의 세그먼테이션과 큰 차이가 없습니다. 다만, 주소 공간이 확장되고 두 가지 서브 모드를 지원하는 등 몇 가지 기능이 추가되면서 약간의 차이가 생겼을 뿐입니다. 그 중에서 중요도 순으로 두 가지만 살펴보면 다음과 같습니다.

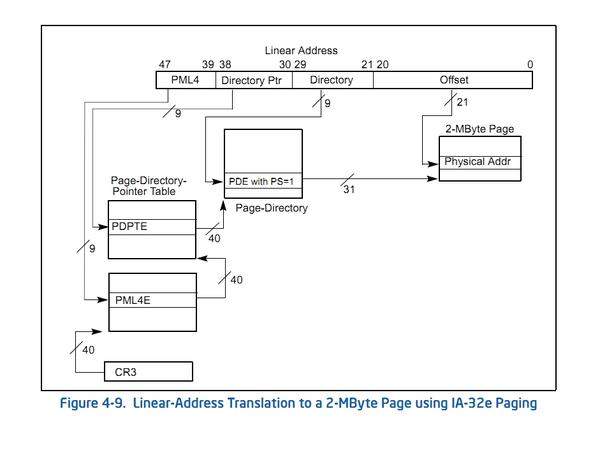
1. 세그먼트 디스크립터에 설정된 기준 주소와 크기에 관계없이 모든 세그먼트가 기준 주소는 0, 크기는 64비트 전체로 설정된다는 것입니다. 이는 보호 모드에서 사용하던 세그먼트 디스크립터가 32비트 어드레스만 저장하게 설게한 탓이며, 64비트 어드레스 지원을 위해 디스크립터를 확장하지 않고 강제로 값을 고정함으로써 이를 해결했습니다. 따라서 IA-32e 모드에서는 선형 주소를 기준 주소가 다른 여러 개의 세그먼트로 구분할 수 없음을 고려해서 os를 설계해야 합니다. 여러분이 예전에 개발한 32비트 os가 커널 영역과 유저 영역을 세그먼트 기준 주소로 구분하고 있다면 IA-32e 모드에서는 유효하지 않으므로 페이징이나 기타 다른 방법을 고려해야 합니다.
2. IA-32e 모드는 호환 모드와 64비트 모드의 두 가지 서브 모드를 지원하므로 이를 구분하고자 코드 세그먼트 디스크립터에 L 필드(비트21)가 추가된 것입니다. 이 필드를 0으로 설정하면 호환 모드로 동작하며 1로 설정하면 64비트 모드로 동작하므로 IA-32e 모드에서는 굳이 보호 모드로 돌아가지 않고도 32비트 코드를 실행할 수 있습니다. [그림 3-15]는 IA-32e 모드의 64비트 모드에서 세그먼트 셀렉터, 세그먼트 디스크립터, 선형 주소의 관계를 나타낸 것입니다. IA-32e 모드의 서브 모드가 호환 모드일 때는 어드레스 변환 방식이 보호 모드와 같으므 [그림 3-10]을 참고하기 바랍니다.

[그림 3-15] IA-32e 모드에서 세그먼트 셀렉터, 세그먼트 디스크립터, 선형 주소의 관계



IA-32e 모드의 페이징은 보호 모드와 달리 주소 공간이 64비트로 늘어나므로 PAE 기능이 기본으로 활성화됩니다. 또한 어드레스가 늘어난 만큼 변환 단계도 늘어나서 4KB 페이지는 5단계로, 2MB 페이지는 4단계로 변경됩니다. 그로 인해 새롭게 추가된 테이블은 페이지 맵 레벨 4 테이블(PML4,page map level 4 table)과 페이지 디렉터리 포인터 테이블(PDPT,page directory pointer table)이며, 변환 과정은 앞서 설명했던 다른 테이블과 같습니다. 보호 모드에서 4KB 페이지의 3단계 방식에 대해 살펴봤으니 이번에는 비교를 위해 4KB 페이지를 가지는 5단계 방식에 대해 알아보겠습니다. [그림 3-16]은 4kb 페이지의 5단계 방식입니다.

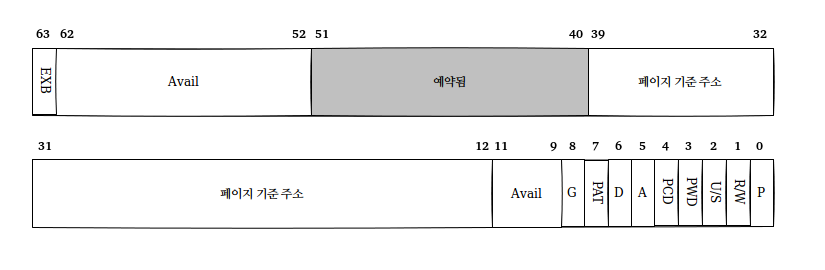
[그림 3-16] IA-32e 모드에서 5단계 페이징과 어드레스 변환 과정



[그림 3-16]에서 보는 바와 같이 보호 모드에서 단계만 더 늘어났을 뿐 변환 과정은 거의 같습니다. 차이점이라면 각 테이블의 인덱스가 9비트로 줄어들어 엔트리의 개수가 512개로 줄어든 정도입니다. 그런데 [그림 3-16]을 유심히 보면 선형주소가 2^64까지 표현할 수 있음에도 불구하고 비트 48~64까지 부호 확장으로 채워진다는 점을 알 수 있습니다. 다시 말하면 64비트 선형 주소 중에 실제로 변환에 사용되는 부분은 48비트까지이며, 최대로 표현 가능한 어드레스 범위가 128TB(2^48)까지라는 말입니다. 그렇다면 48비트 어드레스는 모두 물리 주소로 변환될까요?

그렇지 않습니다 물리 주소로 변환할 수 있는 능력은 각 프로세서에 따라 다르며 프로세서가 40비트의 물리 주소를 지원하는 경우에만 최대 1TB(2^40)의 물리 메모리를 사용할 수 있습니다. 이는 [그림 3-17]을 보면 더욱 명확하게 알 수 있습니다. [그림 3-17]을 보면 더욱 명확하게 알 수 있습니다. [그림 3-17]은 IA-32e 모드의 페이지 테이블 엔트리를 나타낸 것이며 비트 40에서 비트 51까지 모두 0으로 예약된 것을 확인할 수 있습니다. 40비트 물리 메모리를 지원하는 프로세서는 주로 서버용 프로세서이므로 pc에 사용하는 프로세서는 보통 1TB 이하만 (64GB,128GB 등)만 지원합니다. 64비트 어드레스를 모두 사용하지 못해 조금 아쉽지만 64비트 크기의 주소 공간이 필요한 경우는 거의 없으므로 40비트 어드레스만으로도 충분합니다.

[그림 3-17] IA-32e 모드이 페이지 테이블 엔트리



IA-32e 모드의 페이지 엔트리는 64비트로 늘어난 어드레스로 인해 8바이트로 늘어 났습니다. 하위 4바이트는 보호 모드와 구조가 같고 상위 4바이트는 기준 주소 필드와 예약된 영역 임의로 사용 가능한 영역 EXB로 구성됩니다. 이중에서 EXB 필드는 해당 페이지에서 명령어가 실행되는 것을 막는 필드입니다. 이 기능을 이용하면 데이터 영역에서 명령어가 실행되는 것을 막을 수 있으며 os를 더 안전하게 만들 수 있습니다.

데이터 영역에서 명령어가 실행되는 것이 무엇 때문에 문제가 될까요? 이는 데이터 영역이 데이터만 있는 즉 프로그램 코드가 퐘되지 않은 영역이기 때문입니다. 일반적으로 프로그램은 크게 코드 영역과 데이터 영역으로 구분되며, 정상적인 프로그램이라면 데이터 영역에서 명령어가 실행되지 않습니다. 하지만, 간혹 데이터 영역에서 명령어가 실행되는 경우가 있는데, 그 대표적인 예가 악의적인 공격을 받았을 때입니다. 가장 흔한 공격 방법인 버퍼 오버플로우(buffer overfow)와 스택 오버 플로우(stack overflow)는 버퍼나 스택에 프로그램 코드를 삽입한 후 해당 영역의 최대값 이상으로 데이터를 밀어 넣어 강제로 코드를 실행하는 기법입니다. 이때 삽입된 코든느 버퍼나 스택 안에 있으며, 두 영역은 모두 데이터 영역에 포함됩니다. 만약 데이터 영역의 페이지에 EXB 속성을 활성화했다면, 명령어가 실행되는 순간 페이지 폴트 예외가 발생하고 프로그램 실행이 중지되므로 공격을 막을 수 있습니다.

4장 내 PC를 부팅하자

**[용어]**

**Graphic card**

Dedicated vs integrated graphics

As an alternative to the use of a video card, video hardware can be integrated into the motherboard or the CPU. Both approaches can be called integrated graphics. Motherboard-based implementations are sometimes called "on-board video" while CPU-based implementations are known as accelerated processing units (APUs). Almost all desktop computer motherboards with integrated graphics allow the disabling of the integrated graphics chip in BIOS, and have a PCI, or PCI Express (PCI-E) slot for adding a higher-performance graphics card in place of the integrated graphics. The ability to disable the integrated graphics sometimes also allows the continued use of a motherboard on which the on-board video has failed. Sometimes both the integrated graphics and a dedicated graphics card can be used simultaneously to feed separate displays. The main advantages of integrated graphics include cost, compactness, simplicity and low energy consumption. The performance disadvantage of integrated graphics arises because the graphics processor shares system resources with the CPU. A dedicated graphics card has its own random access memory (RAM), its own cooling system, and dedicated power regulators, with all components designed specifically for processing video images. Upgrading to a dedicated graphics card offloads work from the CPU and system RAM, so not only will graphics processing be faster, but the computer's overall performance may also improve.

Both of the dominant CPU makers, AMD and Intel, are moving to APUs. One of the reasons is that graphics processors are powerful parallel processors, and placing them on the CPU die allows their parallel processing ability to be harnessed for various computing tasks in addition to graphics processing. (See Heterogeneous System Architecture, which discusses AMD's implementation.) APUs are the newer integrated graphics technology and, as costs decline, will probably be used instead of integrated graphics on the motherboard in most future low and mid-priced home and business computers. As of late 2013, the best APUs provide graphics processing approaching mid-range mobile video cards[3] and are adequate for casual gaming. Users seeking the highest video performance for gaming or other graphics-intensive uses should still choose computers with dedicated graphics cards. (See Size of market and impact of accelerated processing units on video card sales, below.)

**그래픽 카드 부품**

**1.그래픽 처리 장치**

GPU는 3차원 그래픽 렌더링에 필수적인 부동 소수점 계산에 최적화된 전용 그래픽 마이크로프로세서이다. GPU의 기본 특성은 현대의 카드 기준으로 보통 250MHz에서 1200Mhz 속도를 내는 코어 클럭 속도와 3D 이미지를 픽셀로 변환하는 파이프라인의 수(버텍스 셰이더, 프레그먼트 셰이더)이다.

**2.비디오 메모리**

비디오 카드가 메인보드 통합형인 경우 컴퓨터 램 메모리의 일부를 직접 사용하는 것이 보통이다. 통합형이 아닌 경우, 그래픽 카드가 자체적으로 비디오 램(VRAM)을 장착하고 있다. 현대의 그래픽 카드 메모리 용량은 128MB부터 32GB까지 다양하다. 2003년 이전에는 DDR SDRAM 기술 기반이었지만, 그 뒤에는 더 뛰어난 DDR2, GDDR3, GDDR4, GDDR5 등의 기술을 사용한다. 현대의 카드들의 메모리 클럭 속도는 보통 400MHz에서 1.6Ghz 사이이다. 비디오 메모리에서 아주 중요한 요소는 3D 그래픽의 심도를 관리하는 Z 버퍼이다. 추가로, GDDR4 또는 GDDR5 의 뜻은, 메모리의 클럭을 GDDR(원하는수)만큼 배로 늘리는 것이다. 예시로, 1000MHZ의 메모리를 GDDR5하면 5000MHZ가 된다.

**3.비디오 바이오스.**

비디오 바이오스 또는 펌웨어는 그래픽 카드의 운영을 관리하고 컴퓨터와 소프트웨어가 카드와 소통할 수 있게 도와 주는 명령어를 제공하는 기본 프로그램을 담고 있는 칩이다. 이 칩은 메모리 타이밍, 동작 속도, RAM과 프로세서의 전압 등에 대한 정보를 담고 있다. 그래픽 카드 오버클럭을 위해 바이오스를 수정하여 해당 바이오스 칩에 다시 플래시 적용을 할 수 있다. 그러나 이는 제조회사가 권장하는 방식이 아니며, 그래픽 카드에 잠재적인 위험이 노출될 수 있으니 주의해야 한다.

**4.램댁**

RAMDAC(Random Access Memory Digital-to-Analog Converter의 준말)은 컴퓨터 프로세서가 만들어낸 디지털 신호를 컴퓨터 디스플레이가 알아들을 수 있도록 아날로그 신호로 변환한다. RAMDAC 데이터 전송 속도와 사용되는 비트의 수에 따라, RAMDAC은 컴퓨터 디스플레이의 다른 화면 재생 빈도를 지원한다. CRT 디스플레이의 경우, 깜박임을 최소화하기 위해서는 75 Hz 이상이 최적이며, 60Hz 이하는 권장하지 않는다.[4] (LCD 디스플레이의 경우 어떠한 재생 빈도에서도 깜박임은 문제가 되지 않는다.) 디지털 컴퓨터 디스플레이의 인기와 메인보드의 기능이 성장함에 따라, RAMDAC은 점차 사라지고 있다. 현재의 LCD 및 플라스마 디스플레이, TV는 디지털 방식으로 동작하며 RAMDAC을 요구하지 않는다. 아날로그 입력을 지원하는 일부 오래된 LCD와 플라스마 디스플레이는 RAMDAC을 요구한다.

**5.출력부**

4.1 부팅과 부트 로더

모든 os는 512바이트의 작은 코드로 시작되는데 이 코드가 바로 부트 로더(boot loader)라고 불리며, os의 나머지 코드를 메모리에 복사해 실행시킵니다.

4.1.1 부팅과 BIOS

부팅(Booting)은 pc가 켜진 후에 os가 실행되기 전까지 수행되는 일련의 작업 과정을 의미합니다. 부팅 과정에서 수행하는 작업에는 프로세서 초기화(멀티코어 관련 처리 포함), 메모리와 외부 디바이스 검사 및 초기화, 부트 로더를 메모리에 복사하고 OS를 시작하는 과정 등이 포함됩니다.

[그림 4-1]은 MINT64 OS의 부팅 과정입니다. PC 환겨에서는 부팅 과정 중 하드웨어와 관련된 작업을 BIOS(Basic input/output system)이 담당하며, BIOS에서 수행하는 각종 테스트나 초기화를 POST(Power on Self Test)라고 부릅니다.

BIOS는 메인 보드에 포함된 펌웨어(Firmware)의 일종으로, 이름 그대로 입출력을 담당하는 작은 프로그램입니다. 보통 PC 메인 보드에 롬이나 플래시 메모리에 존재하며, 전원이 켜짐과 동시에 프로세서가 가장 먼저 실행하는 코드입니다. BIOS는 부팅 옵션 설정이나 시스템 전반적인 설정 값(Configuration)을 관리하는 역할도 겸하고 있으며, 설정 값으로 시스템을 초기화하여 os를 실행할 수 있는 환경을 만듭니다. BIOS는 예전 16비트 환경부터 디스크 입출력과 비디오 모드 전환 등 여러가지 기능을 제공해왔습니다. BIOS에서 제공하는 기능은 인터럽트를 통해 사용할 수 있으며, MS-DOS 같은 과거의 16비트 OS는 BIOS의 기능에 많이 의존했습니다. MINT64 os도 os 이미지를 메모리에 복사하고 GUI 모드로 변환할 때 BIOS의 기능을 사용합니다.

[그림 4-1] 우리가 만들 os의 부팅 과정

전원 켜짐

↓

POST

프로세서 초기화

↓

메모리 검사와 초기화

↓

주변 장치 검사와 초기화

부트 로더 이미지를 메모리에 복사 (0x7C00) BIOS 영역

↓ OS 영역

부트 로더

os 이미지를 1MB 이하의 메모리에 복사

보호 모드로 전환

↓

32비트 커널

64비트 커널을 2MB 이상의 메모리로 복사

IA-32e 모드로 전환

↓

64비트 커널

멀티 코어 설정 및 초기화

↓

멀티 태스킹 모듈 초기화

↓

메모리 관리 모듈 초기화

↓

주변 장치 초기화

(키보드,마우스,하드 디스크,시리얼)

↓

그래픽 모드로 전환

↓

응용 프로그램 실행

BIOS는 부팅 과정에서 시스템 초기화 외에 수많은 작업을 하지만, 그 중에서 우리에게 가장 중요한 것은 부트 로더 이미지를 메모리로 복사하는 단계입니다. 부트 로더는 부트스트랩(bootstrap) 코드라고도 불리며 우리가 BIOS에서 처음으로 제어를 넘겨받는 부분입니다. 부트 로더는 플로피 디스크나 하드 디스크 등 저장 매체의 가장 앞부분에 존재합니다. Pc는 디스크나 플래시 메모리 등 다양한 장치로 부팅할 수 있으므로 BIOS는 POST가 완료된 후 여러 장치를 검사하여 앞부분에 부트 로더가 있는지 확인합니다. 부트 로더가 존재하면 코드를 0x7c00 어드레스에 복사한 후 프로세서가 0x7c00 어드레스부터 코드를 수행하도록 합니다. 부팅 가능한 모든 장치를 검사했는데도 부트 로더를 찾을 수 없다면 BIOS ‘Operating System Not Found’와 같은 메시지를 출력하고 작업을 중단합니다.

부트 로더가 디스크에서 메모리로 복사되어 실행되었다는 것은 BIOS에 의해 PC가 정상적으로 구동되었다는 것을 의미합니다. 다시 말하면 우리가 만든 OS를 메모리에 올려서 실행할 준비가 된 것입니다.

4.1.2 부트 로더의 역할과 구성

부트 로더는 플로피 디스크나 하드 디스크 같은 외부 저장 매체에 있으며, 저장 매체에서 가장 첫 번째 섹터 MBR(Master Boot Record)에 있는 작은 프로그램입니다 섹터(Sector)는 디스크를 구성하는 데이터의 단위로 섹터 하나는 512바이트로 구성됩니다. 부트 로더의 가장 큰 역할은 os 실행에 필요한 환경을 설정하고, os 이미지를 메모리에 복사하는 일입니다. 부트 로더는 BIOS가 가장 먼저 실행하는 중요한 프로그램이므로 기능이 다양하다고 생각할지도 모릅니다. 하지만, 부트 로더는 크기가 512바이트로 정해져 있습니다. 즉 공간 제약이 있어서 처리할 수 있는 기능이 한정됩니다. 이렇게 작은 공간에 다양한 기능을 우겨 넣는 일은 무리이므로 대부분 부트 로더는 OS 이미지를 메모리에 복사하고 제어를 넘겨주는 정형화된 작업을 수행합니다. 4장과 5장에서 우리가 만들 부트 로더 역시 os 이미지를 디스크에서 메모리로 복사하는 역할만 수행합니다.

부트 로더의 크기 문제는 부트 로더의 기능을 최소화해서 해결했습니다. 그렇다면 BIOS에 첫 번째 섹터가 부트로더란 것을 어떻게 알려줄까요? 그리고 BIOS는 디스크에서 읽은 첫 번째 섹터가 정상적인 부트 로더인지 어떻게 판단할까요? 여러분이 디스크를 부팅할 용도로 사용하지 않는다면 첫 번쨰 섹터는 부트 로더가 아닌 일반 데이터가 저장됩니다. 만약 BIOS가 실수로 데이터를 메모리에 올려 실행한다면 모니터네 번쩍하는 섬광과 함께 PC가 리부팅되는 스릴을 맛볼 것입니다. 이러한 사태를 방지하려면 BIOS는 첫 번째 섹터에 있는 데이터가 부트 로더인지 확인해야 합니다.

이를 위해 BIOS는 읽어 들인 512 바이트 중에 가장 마지막 2바이트의 값이 0x55,0xAA인지 검사해서 부트 로더인지 확인합니다. 읽은 데이터가 0x55,0xAA로 끝나지 않는다면 데이터로 인식하고 부팅 과정을 더 진행하지 않습니다. 여러분이 첫 번째 섹터에 부트 로더가 아닌 데이터를 저장할 생각이라면 정신 건강과 pc 수명 연장을 위해 적어도 마지막 2바이트는 0x55,0xAA가 아닌 다른 값으로 쓰기 바랍니다.

디스크의 첫 번째 섹터인 MBR 영역에는 부트 로더 외에 디스크의 파티션 정보도 있습니다. 파티션(partition)은 디스크 영역을 논리적으로 구분하는 단위입니다. MBR 영역에는 4개의 파티션 영역이 있으며, 파티션 영역에 정의된 영역을 독립된 공간을 보장받습니다.

4.2 부트 로더 제작을 위한 준비

4.2.3 makefile 파일 생성

make 프로그램

makefile을 서명하기에 앞서 make 프로그램에 대해 먼저 설명하겠습니다. make 프로그램은 소스 파일을 이용해서 자동으로 실행 파일 또는 라이브러리 파일을 만들어주는 빌드 관련 유틸리티입니다. make 프로그램은 소스 파일과 목적 파일을 비교한 뒤 마지막 빌드 후에 수정된 파일만 선택하여 빌드를 수행하므로 빌드 시간을 크게 줄여줍니다. 또한 빌드를 편리하게 해주는 여러 가지 문법과 규칙을 지원하므로 이를 활용하면 많은 수의 소스 파일도 한 번에 빌드할 수 있습니다. 하지만, make 프로그램도 만능이 아닙니다. make 프로그램이 빌드를 자동으로 수행하려면 각 소스 파일의 의존 관계나 빌드 순서, 빌드 옵션 등에 대한 정보가 필요합니다. 이러한 내용이 저장된 파일이 바로 makefile입니다.

4장에서는 부트 로더에 관련된 소스 파일 하나만 있지만, 앞으로 기능이 추가되면서 각 디렉터리에 수십 개의 소스 파일이 가득 찰 것입니다. 이러한 환경에서 일일이 빌드 명령을 내리거나 배치 파일로 빌드하는 것은 비효율적이며 실수하기 쉽습니다. MINT64 OS에서는 디렉터리별로 계층 관계가 있는 makefile을 구성하고, 이를 통해 최종 os 이미지를 생성하게 했습니다. 즉 운영 모드별로 구분된 디렉터리가 최종 빌드 결과물을 최상위 디렉터리로 복사하고, 최상위 디렉터리의 makefile은 이 결과물을 이용해서 최종적으로 os 이미지를 생성하는 방식입니다. 각각의 하위 디렉터리에 있는 makefile은 해당 디렉터리의 c 언어 파일과 어셈블리어 파일을 자동으로 추출함으로써 더 편리하고 효율적으로 빌드를 수행하도록 했습니다.

make 문법

make의 문법은 복잡하고 다양합니다.

[make의 기본 문법]

Target: Dependency

<tab> command

<tab> command

Target은 일반적으로 생성할 파일을 나타내며, 특정 레이블(label)을 지정하여 해당 레이블과 관련된 부분만 빌드하는 것도 가능합니다. Dependency는 Target 생성에 필요한 소스 파일이나 오브젝트 파일 등을 나타내고, command는 Dependency에 관련된 파일이 수정되면 실행할 명령을 의미합니다. command에는 명령창이나 터미널에서 실행할 명령 또는 프로그램을 기술합니다. <tab>으로 표시한 부분은 반드시 TAB 문자로 띄워야 합니다. 공백으로 탭 문자를 대체하면 make가 정상적으로 실행되지 않으니 주의해야 합니다.

[간단한 makefile 예제]

# a.c, b.c를 통해서 output.exe 파일을 생성하는 예제 <- 주석 (comment)

all: output.exe <- 별다른 옵션이 없을 떄 기본적으로 생성하는 target을 기술

a.o: a.c

gcc –c a.c

b.o: b.c

gcc –c b.c

output.exe: a.o b.o

gcc –o output.exe a.o b.o

make는 최종으로 생성할 Taget의 의존성을 추적하면서 빌드를 처리하기 때문에 makefile은 역순으로 따라가면 됩니다.

make는 빌드를 수행하는 도중에 다른 make를 실행할 수 있습니다. 이는 빌드 단계를 세부적으로 나누고, 계층적으로 수행할 수 있음을 의미합니다. 최상위 디렉터리의 하위에 Library 디렉터리가 있고, 빌드 과정에서 Library 디렉터리를 빌드해야 한다면 –C 옵션을 사용해서 다음과 같이 간단히 처리할 수 있습니다.

[계층적 빌드]

all: output.exe

# Library 디렉터리로 이동한 후 make를 수행

libtest.a:

make –C Library

output.o: output.c

gcc –c output.c

output.exe: libtest.a output.o

gcc –o output.exe output.c –ltest –L./

4.3 부트 로더 제작과 테스트

4.3.1 세상에서 가장 간단한 부트 로더

부트 로더를 메모리에 정상적으로 복사하려면 한 가지만 지키면 됩니다. 부트 섹터 512바이트에서 마지막 2바이트를 0x55,0xAA로 저장하면 됩니다.

[표 4-1] 주로 사용되는 어셈블리어 명령어

그룹 명령어 설명

사칙 연산 add A,B A에 B의 값을 더한 후 A에 저장

sub A, A에서 B의 값을 뺀 후 A에 저장

mul A AX의 레지스터 값과 A의 값을 곱한 후 AX 또는 DX:AX에 저장

inc A A의 값을 1 증가

dec A A의 값을 1 감소

div A AX나 DX:AX의 값을 A로 나누어 몫과 나머지를 각기 AL와 AH 또는 AX와 DX에 저장

논리 연산 and A,B A에 B 값을 AND하여 A에 저장

or A,B A에 B 값을 OR하여 A에 저장

xorA,B A에 B 값을 XOR하여 A에 저장

not A A의 값을 반전(0<->1)하여 A에 저장

대입 및 분기 mov A,B B에서 A로 값을 이동

cmp A,B 두 값을 비교하여 결과를 FLAGS 레지스터에 적용

jmp A 무조건 해당 어드레스로 이동하여 A 위치의 코드를 실행

je,ja,jb,jz,jne,jna, 조건 분기 명령으로 FLAGS 레지스터의 값에 따라 jmp 수행

jnb,jnz A 일반적으로 값을 비교하는 cmp 명령어와 함께 사용

Equal(e),Above(a),Bellow(b),Zero(z),Not(n) 등의 다양한 조건 포함

함수 호출 call A 스택에 call 명령 다음의 어드레스를 삽입하고 A 위치의 코드를 실행

함수를 호출하는 용도로 사용

ret A 스택을 A만큼 줄인 후에 되돌아갈 어드레스를 꺼내 해당 어드레스의 코드실행

일반적으로 call 명령과 짝을 이루어 사용

스택 push A A의 값을 스택에 저장

pop A 스택에서 값을 꺼내 A에 저장

인터럽트 int A A 번째 소프트웨어 인터럽트를 발생

cli 인터럽트를 발생 불가능하도록 설정

sti 인터럽트를 발생 가능하도록 설정

I/O 포트 제어 in A,B I/O 포트 B에서 값을 입력 받아 A에 저장

주변 장치에서 값을 읽은 용도로 사용

out B,A A의 값을 I/O 포트 B에 출력

주변 장치에 값을 쓰는 용도로 사용

AX 레지스터의 내용을 BX 레지스터로 복사하는 어셈블리어 코드(인텔 문법과 AT&T 문법)

인텔 문법

mov bx, ax ; BX 레지스터에 AX 레지스터의 값을 복사

AT&T 문법

movw %ax,%bx ; BX 레지스터에 AX 레지스터의 값을 복사

4.3.2 QEMU 실행

qemu-x86\_64.bat 파일의 내용

qemu-system-x86\_64.exe –L . –m 64 –fda c:\MINT64\Disk.img –localtime –M pc

-L . : bios 디렉터리 설정

-m 64 : 64MB의 물리 멤리 할당

–fda c:\MINT64\Disk.img : 플로피 디스크 이미지로 MINT64 이미지(Disk.img) 설정

-localtime : Set the real time clock to local time

-M pc : 가상 머신을 일반 pc 환경으로 설정

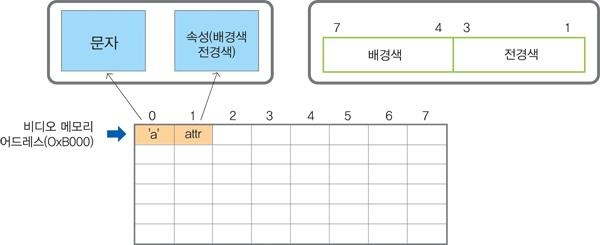
4.3.3 화면 버퍼와 화면 제어

리눅스 계열이나 과거 16비트 os는 부팅이 시작 되면 환영 메시지와 함께 부팅 과정을 화면에 표시했습니다. MINT64 OS 부트 로더에도 그와 비슷한 기능을 넣어보겠습니다.

화면에 문자를 출력하려면 현재 동작 중인 화면 모드와 관련된 비디오 메모리의 어드레스를 알아야 합니다. 비디오 메모리는 화면 출력과 관련된 메모리로 모드 별로 정해진 형식에 따라 데이터를 채우면 화면에 원하는 문자나 그림을 출력하는 구조로 되어 있습니다.

pc 부팅후 기본으로 설정되는 화면 모드는 텍스트 모드로 화면 크기는 가로 80문자, 세로 25문자이며 비디오 메모리 어드레스는 0xB800에서 시작합니다. 또한 화면에 표시하는 한 문자는 문자값 1바이트와 속성값 1바이트로 구성되며 총 메모리 크기는 화면 크기를 곱한 크기의 80\*25\*2=4,000 바이트입니다.

[그림 4-6] 텍스트 모드의 화면 구조



속성 값은 하위 4비트의 전경색과 상위 4비트의 배경색으로 구분됩니다. 각 전경색과 배경색은 다시 최상위의 특수 기능 비트와 하위 3비트의 색상으로 구분됩니다. 전경색은 강조 효과만 지원하지만, 배경색은 강조와 깜빡임 두 기능을 제공합니다. 전경색과 배경색의 최상위 비트를 모두 강조 기능으로 사용하면 16가지 색상을 표현할 수 있으므로 이를 이용하면 그럴듯한 텍스트 화면을 표시할 수 있습니다. [그림 4-6]은 이러한 텍스트 모드의 화면 구조이며, [표 4-2]는 전경색과 배경색의 값과 의미를 정리한 것입니다.

속성을 나타내는 1바이트에서 최상위 비트는 배경색 강조 효과 또는 깜빡임 효과를 설정하는 역할을 합니다. 이 비트의 실제 역할은 비디오 컨트롤러(video controller)의 속성 모드 제어 레지스터(attribute mode control register)에 따라 결정됩니다. 속성 모드 제어 레지스터의 blink 비트를 1로 설정하면 깜빡임 효과가 되며, 0으로 설정하면 배경색 강조 효과가 됩니다. qemu에서는 기본으로 배경색 강조 효과로 설정되어 있으며 안타깝게도 속성 모드 제어 레지스터에 blink 비트를 1로 변경해도 깜빡임 효과는 적용되지 않습니다.

[표 4-2] 텍스트 모드 속성값의 구성

|  |  |  |  |
| --- | --- | --- | --- |
| 비트 위치 | 값 | 배경색(상위 4비트) | 전경색(하위 4비트) |
| 0~2 | 0x00 | 검은색(black) | |
| 0x01 | 파란색(blue) | |
| 0x02 | 녹색(green) | |
| 0x03 | 청색(cyan) | |
| 0x04 | 빨간색(red) | |
| 0x05 | 자홍색(magenta) | |
| 0x06 | 노란색(yellow) | |
| 0x07 | 흰색(white) | |
| 3 | 0x00 | 효과 없음 | |
| 0x01 | 하위 3비트 색상에 깜빡임 효과나 강조 효과(밝기 상승)추가 | 하위 3비트 색상에 강조 효과(밝기 상승) 추가 |

이제 화면에 문자를 표시하려면 0xB800 어드레스에 문자와 속성을 순서대로 지정하면 된다는 것을 알았습니다. 이를 이용해서 화면 맨 위에 M을 빨간색 배경에 밝은 녹색으로 출력해 보겠습니다. 3장에서 설명했듯이 리얼 모드의 어드레스 계산 방식은 세그먼트 레지스터에 정의된 기준 주소에 값을 더해 계산합니다. 따라서 0xB8000 어드레스에 접근하려면 세금너트 레지스터에 세그먼트의 기준 주소부터 설정해야 합니다. 세그먼트 레지스터와 범용 레지스터를 조합해서 0xB800 어드레스에만 접근할 수 있다면 어떤 값을 설정해도 문제는 없습니다. 그러나 세그먼트 레지스터의 값을 0xB800으로 설정하면 세그먼트 레지스터:오프셋이 0xB800:0x0000이 되어 범용 레지스터의 0을 비디오 메모리의 첫 번째 어드레스로 지정할 수 있어서 편리합니다. 세그먼트 레지스터에 0xB800을 설정하는 코드는 아래와 같이 mov 명령을 이용하여 간단히 처리할 수 있습니다.

[세그먼트 레지스터에 비디오 메모리 어드레스 설정]

mov ax, 0xB800 ; AX 레지스터에 0xB800 복사

mov ds, ax ; DS 세그먼트 레지스터에 AX 레지스터의 값(0xB800)을 복사

DS 세그먼트 레지스터에 0xB800의 값을 설정했으니, 이후 데이터에 접근하는 명령어는 물리 주소 0xB8000이 기준 어드레스로 사용됩니다. 화면 맨 위의 어드레스는 0xB8000과 같으므로 0xB8000과 0xB8001에 각각 ‘M’과 0x4A를 쓰면 빨간색 배경에 밝은 녹색으로 ‘M’을 출력할 수 있습니다.

어셈블리어에서 어드레스에 해당하는 메모리 값을 참조할 때 [ ]기호를 사용하며, 기호 앞에 byte, word(2바이트), dword(4바이트), qword(8바이트)를 사용하여 메모리 크기를 지정합니다.

[화면 최상단에 M을 표시하는 소스 코드]

mov byte[0x00] , ’M’ ; DS 세그먼트:오프셋 0xB800:0x0000에 “M”을 복사

mov byte[0x01], 0x4A ; DS 세그먼트:오프셋 0xB800:0x0001에 0x4A를 복사

4.3.4 세그먼트 레지스터 초기화와 Hello, World~!

지금까지 화면에 문자를 표시하는 방법을 알아봤습니다. 이제 이 코드를 응용해서 환영 메시지를 출력해 보겠습니다.

세그먼트 레지스터 초기화

[예제 4-4]에서는 부트 로더가 잘 동작하는지 눈으로 확인할 수 있게 문자를 출력하는 데 초점을 맞추었습니다. 앞으로 작업을 더 진행하기 위해서는 문자를 출력하는 코드 이전에 세그먼트 레지스터를 초기화하는 코드가 필요합니다. 왜냐하면 BIOS가 부트 로더를 실행했을 때 세그먼트 레지스터에는 BIOS가 사용하던 값이 들어 있기 때문입니다. 세그먼트 레지스터를 초기화하지 않으면 엉뚱한 어드레스에 접근할 수 있으므로 미리 초기화하고 사용해야 좋습니다.

그렇다면 어떤 값으로 세그먼트 레지스터를 초기화해야 할까요? MINT64 OS에서는 0x7C00으로 초기화 했습니다. 그 이유는 BIOS가 부트로더를 디스크에서 읽어 메모리에 복사하는 위치가 0x7C00이기 때문입니다. 또한 부트 로더의 코드와 데이터는 0x7C00부터 512 바이트 범위에 존재하므로 CS와 DS 세그먼트 레지스터를 모두 0x7C00으로 설정했습니다. CS 세그먼트 레지스터를 제외한 세그먼트 레지스터는 앞서 설명했듯이 mov 명령으로 처리할 수 있습니다. 하지만, CS 세그먼트 레지스터는 mov 명령으로 처리할 수 없으며, 수정하려면 jump 명령를 이용해야 합니다. 다음은 jmp 명령과 mov 명령을 이용해서 세그먼트를 초기화하는 코드입니다.

[세그먼트 레지스터 초기화]

SECTION .text ; text 섹션(세그먼트)을 정의

jmp 0x07C0:START ; CS 세그먼트 레지스터에 0x07C0을 복사하면서 , START 레이블로 이동 (CS 세그먼트는 데이터 이동 명령으르 설정불가능하며 점프나 인터럽트 관련 명령으로 변경 가능

START:

mov ax, 0x07C0 ; 부트 로더의 시작 어드레스(0x7C00)를 세그먼트 레지스터 값으로 변환

mov ds, ax ; DS 세그먼트 레지스터에 설정

mov ax, 0xB800 ; 비디오 메모리의 시작 어드레스(0xB800)를 세그먼트 레지스터 값으로 변환

mov es, ax ; ES 세그먼트 레지스터에 설정

비디오 모드에 관련된 세그먼트 레지스터가 DS 세그먼트 레지스터에서 ES 세그먼트 레지스터로 변경되었으니 이후 출력에 관계된 코드는 모두 ES 세그먼트 레지스터를 기준으로 하게 수정해야 합니다. 별다른 처리 없이 메모리에 접근하면 암시적으로 DS 세그먼트 레지스터가 사용됩니다. 비디오 메모리에 접근하려면 ES 세그먼트 레지스터를 사용해야 하는 데 어떻게 하면 세그먼트 레지스터를 지정할 수 있을까요? 그렇습니다. 3장의 3.3.2절에서 설명했던 세그먼트 레지스터 접두사를 사용하면 됩니다. 세그먼트 레지스터 접두사는 해당 명령을 수행하는 동안 일시적으로 세그먼트를 교체합니다. 세그먼트 레지스터 접두사를 쓰는 방법은 아주 간단합니다. 어드레스를 지정하는 오퍼랜드에 ES;0x01처럼 [세그먼트 레지스터:오프셋] 형식으로 쓰면 됩니다.

[세그먼트 레지스터 접두사를 사용해서 화면 맨 위에 M을 표시하는 소스 코드]

mov byte [es:0x00], ‘M’

mov byte [es:0x01],0x4A

실제 PC에서 부팅 테스느

플로피 디스크에 MINT64 OS의 이미지를 옮기려면, 플로피 디스크의 섹터 순서대로 os 이미지를 복사해야 합니다. 이러한 작업을 수행하는 프로그램을 작성해도 괜찮지만, NTRawrite라는 윈도우 프로그램이 이미 공개되어 있으니 이를 활용하겠습니다.

[Disk.img 파일을 플로피 디스크에 복사]

NTRawrite –f disk.img –d a:

A 드라이브에 플로피 디스크 드라이브가 연결되어 있고, Disk.img 파일을 플로피 디스크에 기록해야 할때

5장 플로피 디스크에서 OS 이미지를 로딩하자

4장에서 PC 부팅 과정과 부트 로더 코드를 살펴봤습니다. 앞 장에 이어서 이번 장에서는 OS 이미지를 메모리로 복사하는 기능을 추가해 보겠습니다. 빌드가 끝나면 MINT64 OS의 최종 이미지는 플로피 디스크용으로 생성됩니다. 이는 부트 로더가 플로피 디스크에서 os 이미지를 읽어 메모리로 복사해야한다는 것을 의미합니다.(추후에 usb로 부팅하는 방법도 살펴보겠습니다.) 플로피 디스크를 제어하는 방법은 1. 직접 플로피 디스크 컨트롤러에 접근하는 방법과 2. bios 서비스를 이용하는 방법 두 가지가 있습니다. 플로피 디스크 컨트롤러에 직접 접근하는 방법은 컨트롤러에 대한 이해가 선행되어야 하고, 제어 코드를 직접 작성해야 하므로 복잡하고 어렵습니다. 컨트롤러에 직접 접근해서 사용하는 예는 하드 디스크를 제어하는 24장에서 살펴보기로 하고, 5장에서는 bios의 기능을 사용해서 처리하겠습니다.

bios 서비스를 이용하면 os 이미지를 로딩하는 기능을 구현할 수 있습니다

5.1 bios 서비스와 소프트웨어 인터럽트

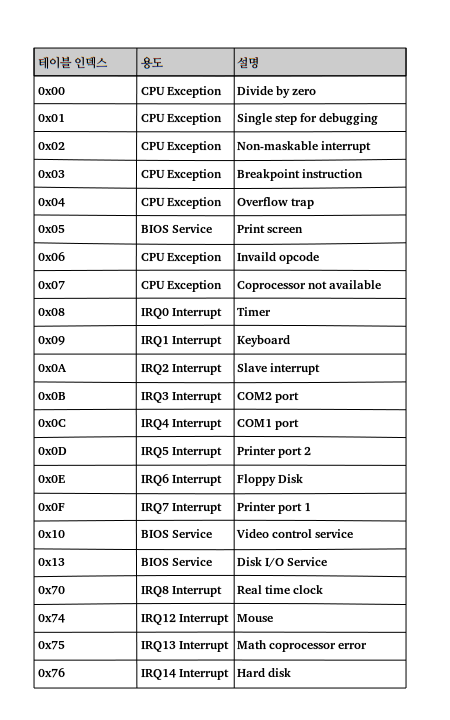
bios는 키보드/마우스에서 디스크나 프린터까지 거의 모든 pc 주변 기기를 제어하는 기능을 제공합니다. 여러분이 16비트 os를 목표로 한다면 bios의 기능만 활용해도 os를 만들 수 있습니다.

bios는 우리가 일반적으로 많이 쓰는 라이브러리(library)파일과 달리 자신의 기능을 특별한 방법으로 외부에 제공합니다. 함수의 어드레스를 인터럽트 벡터 테이블(interrupt vector table)에 넣어두고, 소프트웨어 인터럽트(swi,software interrupt)를 호추라는 방법을 사용합니다. 인터럽트 벡터 테이블은 메모리 어드레스 0에 있는 테이블로 특정 번호의 인터럽트가 발생했을 때 인터럽트를 처리하는 함수(인터럽트 핸들러,interrupt handler) 검색에 사용합니다. 테이블의 각 항목은 인덱스에 해당하는 인터럽트가 발생했을 때 처리하는 함수 어드레스가 지정되어 있으며, 각 항목은 크기가 4바이트 입니다. why? 리얼 모드는 주소가 2^20승인데 왜 4바이트가 필요한다. 2byte – segment address(base address) 2byte – offset 또한 인터럽트는 최대 256개까지 설정할 수 있으므로 리얼 모드의 인터럽트 벡터 크기는 최대 256\*4=1024바이트가 됩니다.

IRQ와 CPU 예외에 대한 내용은 인터럽트 처리와 관련된 장에서 자세히 다룰 예정이므로 지금은 bios 서비스에 초점을 맞추겠습니다. bios가 제공하는 디스크 서비스를 사용하려면 [표 5-1]에서도 나와 있듯이 0x13 인터럽트를 발생시켜야 합니다. 이는 우리가 임의로 인터럽트를 발생시킬 수 있어야 한다는 것을 의미하며, 이때 소프트웨어 인터럽트 명령을 사용합니다. swi는 cpu에 가상으로 특정 인터럽트가 발생했다고 알리는 명령어로 int 0x13 형태로 사용합니다. 여러분이 만든 함수의 어드레스를 인터럽트 벡터 테이블에 넣어뒀다면 int 명령으로 언제든지 해당 함수로 이동할 수 있습니다.

bios 서비스는 swi를 통해 호출할 수 있지만 bios도 만능은 아니어서 작업에 관련된 파라미터를 넘겨줘야합니다. bios의 기능을 사용할 때는 ax,bx,cx,dx 레지스터와 es 세그먼트 레지스터를 사용해서 파라미터를 넘겨주며 결과값도 레지스터를 통해 넘겨받습니다. 물론, bios 서비스마다 요구하는 파라미터의 수가 다르므로 서비스를 호출할 때 파라미터로 정의된 레지스터를 꼭 확인해야 합니다.

[표 5-1] 리얼 모드의 주요 인터럽트 벡터 테이블



[표 5-2] 리셋과 섹터 읽기에 사용하는 레지스터

기능 입/출력 레지스터 설명

리셋 입력 AH 기능 번호

리셋 기능을 사용하려면 0으로 설정

DL 드라이브 번호

플로피 디스크(0x00), 첫 번째 하드 디스크(0x80), 두 번째 하드 디스크(0x81) 선택가능

출력 AH 기능 수행 후 드라이브 상태 값

성공(0x00)외 나머지 값은 에러 발생

FLAGS의 CF 비트 성공 시 CF 비트를 0으로 설정

에러 발생 시 CF 비트를 1로 설정

섹터읽기 입력 AH 기능 번호

섹터 읽기 기능을 사용하려면 2로 설정

AL 읽을 섹터의 수

1~128 사이의 값

CH 트랙이나 실린더 번호

CL의 상위 2비트를 포함하여 총 10비트 크기

0~1023 사이의 값

CL 읽기 시작할 섹터 번호

1~18 사이의 값

DH 읽기 시작할 헤드 번호

0~15 사이의 값

DL 드라이브 번호

플로피 디스크(0x00), 첫 번째 하드 디스크(0x80), 두 번째 하드 디스크(0x81) 선택가능

ES:BX 읽은 섹터를 저장할 메모리 어드레스

64KB 경계에 걸치지 않게 지정

출력 AH 기능 수행 후 드라이브 상태

성공(0x00)외 나머지 값은 에러 발생

AL 읽은 섹터 수

FLAGS의 CF 비트 성공 시 CF 비트를 0으로 설정

에러 발생 시 CF 비트를 1로 설정

플로피 디스크는 자기 원반(magnetic disk)으로 구성된 저장매체입니다. 다양한 종류와 용량의 플로피 디스크가 있지만, 지금까지 널리 쓰이는 3.5인치 플로피 디스크를 기준으로 설명하겠습니다. 일반적인 플로피 디스크는 물리적으로 섹터, 트랙, 헤드로 구성되며 각 부분의 의미는 다음과 같습니다.

헤드 – 디스크의 표면을 의미합니다. 디스크 하나는 두 개의 표면으로 구성되고 각 표면에 데이터 저장이 가능하므로 헤드의 개수는 디스크의 수X2가 됩니다. 헤드 번호는 0~1의 값을 갖습니다.

트랙 – 디스크를 여러 개의 동심원으로 나눴을 때, 그 동심원 하나가 가지는 영역을 의미합니다. 플로피 디스크는 디스크를 모두 80개의 동심원으로 구분하므로 트랙의 수는 모두 80개입니다. 트랙 번호 역시 0부터 시작하기 때문에 트랙 번호는 0~79의 값을 갖습니다.

섹터 – 디스크를 구성하는 가장 작은 단위로 트랙을 다시 여러 조각으로 자른 것입니다. 플로피 디스크는 한 트랙을 18개의 같은 크기로 구분하므로 트랙에 포함된 섹터의 수는 모두 18개입니다. 섹터 하나는 512바이트로 구성되며, 이는 디스크에 관련된 작업을 수행할 때 사용한는 기본 단위가 됩니다. 섹터 번호는 위의 두 가지와 달리 1부터 시작하므로, 섹터 번호는 1~18의 값을 갖습니다.

디스크의 물리적인 구조는 비록 섹터, 트랙, 헤드로 구성되지만, 논리적인 관점에서 보면 디스크는 순차적으로 정렬된 섹터의 집합으로 볼 수 있습니다. 섹터(1~18)->헤드(0~1)->트랙(0~79)의 순서로 증가시키면서 읽으면 됩니다.

5.2 os 이미지 로딩 기능 구현

5.2.1 디스크 읽기 기능 구현

mint 64 os의 이미지는 크게 부트 로더, 보호 모드 커널, ia-32e 모드 커널로 구성되며, 각 부분은 섹터 단위로 정렬해서 하나의 부팅 이미지 파일로 합칩니다. 따라서 디스크의 두 번째 섹터부터 읽어서 특정 메모리 어드레스에 순서대로 복사하면 이미지 로딩은 끝입니다. mint64 os는 os 이미지를 0x10000(64kbyte)에 로딩해서 실행합니다. 하지만, os 이미지를 반드시 0x10000 위치에 로딩해야 실행되는 것은 아닙니다. 부트 로더 이후(0x07c00)에 연속해서 복사해도 os 실행에 문제는 없습니다. mint64 os는 0x10000 하위 영역을 다른 용도로 사용하기에 남겨둔 것입니다.

플로피 디스크의 첫 번째 섹터는 부트 로더로 bios가 메모리에 로딩합니다. 따라서 플로피 디스크의 두 번째 섹터부터 os 이미지 크기만큼은 읽어서 메모리에 복사하면 됩니다. 플로피 디스크의 섹터는 섹터->헤드->트랙의 순서로 배열되어 있으므로 이 순서만 지킨다면 큰 문제 없이 로딩할 수 있습니다.

[1024 섹터 크기의 이미지를 메모리로 복사하는 소스코드(c언어)]

int main(int argc, char\* argv[])

{

int iTotalSectorCoun=1024;

int iSectorNumber=2;

int iHeadNumber=0;

int iTrackNumber=0;

// 실제 이미지를 복사할 어드레스(물리 주소)

char\* pcTargetAddress=(char\*)0x10000;

while(1)

{

// 전체 섹터 수를 하나씩 감소하면서 0이 될 때까지 섹터를 복사

if(iTotalSectorCount==0)

{

break;

}

iTotalSectorCount -=1;

// 1 섹터를 읽어 들여서 메모리 어드레스 복사

if(BIOSReadOneSector(iSectorNumber,iHeadNumber,iTrackNumber,pcTargetAddress)==ERROR)

{

HandleDiskError();

}

// 1 섹터를 512(0x200) 바이트이므로, 복사할 섹터 수만큼 어드레스 증가

pcTargetAddress = pcTargetAddress + 0x200;

// 섹터->헤드->트랙 순으로 번호 증가

iSectorNumber = iSectorNumber+1;

if(iSectorNumber<19)

{

continue;

}

iHeadNumber = iHeadNumber^0x01;

iSectorNumber = 1;

if(iHeadNumber!=0)

{

continue;

}

iTrackNumber=iTrackNumber+1;

}

return 0;

}

OxA0000 0xA0000 is the pointer address to the Graphical Mode and 0xB8000 is the pointer address to the Color Text Mode and 0xB0000 is the pointer to the Monochrome text Mode.

Therefor those OSDEVERs who are using GUI writes to 0xA0000 and those who are still stuck in the text mode write to 0xB80000.

Now YOu have said that they use C/C++ and write directly to the address. This is because these are Memory Mapped I/O. If you can write directly to them you save a lot of CPU cycles that fasten up your OS.

5.2.2 스택 초기화와 함수 구현

x86 프로세서에서 함수를 사용하려면 스택(stack)이 꼭 필요합니다. 스택은 데이터를 삽입하는 포인터와 제거하는 포인터가 같아서, FIFO 형식입니다.

x86 프로세서에서는 함수를 호출한 코드의 다음 어드레스, 즉 되돌아갈 어드레스(이하 복귀 어드레스,return address)를 저장하는 용도로 스택을 사용합니다. 함수를 호출(call)하면 프로세서가 자동으로 되돌아올 어드레스를 스택에 저장하며, 호출된 함수에서 되돌아감(ret)을 요청하면 자동으로 스택에서 어드레스를 꺼내 다음 어드레스로 이동하는 것입니다. 스택은 복귀 어드레스를 저장하는 역할뿐만 아니라 함수의 파라미터를 저장하는 역할도 겸합니다. 호출하는 쪽(caller)과 호출되는 쪽(calle)은 정해진 규칙에 따라 파라미터를 스택에 저장함으로써 협업할 수 있습니다.

함수 호출을 위해 가장 먼저 해야 할일은 스택 생성입니다. x86 프로세서는 스택 관련 레지스터가 세 가지 있습니다. 스택 세그먼트 레지스터(ss)와 스택 포인터 레지스터, 그리고 베이스 포인터 레지스터(bp)가 그것입니다. 스택 세그먼트 레지스터(ss)는 스택 영역으로 사용할 세그먼트의 기준 주소를 지정합니다. 스택 포인터 레지스터(sp)는 데이터를 삽입하고 제거하는 상위(top)를 지정합니다. 마지막으로 베이스 포인터 레지스터(bp)는 스택의 기준 주소를 임시로 지정할 떄 사용합니다. 16비트 모드는 세그먼테이션 방식으로 어드레스를 변화하므로 스택 세그먼트 레지스터를 사용해서 최대 64kb(0x10000)를 스택 영역으로 지정할 수 있습니다. 스택 세그먼트 레지스터(ss)에 0x0000을 설정한다면 사용 가능한 영역은 0x00000~0x0FFFF까지 되며, 스택 세그먼트 레지스터(ss)에 0x1000을 설정한다면 사용 가능한 영역은 0x010000~0x01FFFF까지 됩니다. 스택 세그먼트 레지스터(ss)로 스택 세그먼트의 범위는 지정할 수 있지만, 실제 스택의 크기는 지정할 수 없습니다. 스택의 실제 크기는 스택 포인터 레지스터(sp)와 베이스 포인터 레지스터(bp)의 초깃값으로 지정합니다. x86 프로세서의 스택은 [그림 5-3]과 같이 데이터가 삽입될 때마다 스택의 상위(top)를 나타내는 스택 포인터 레지스터(sp)가 낮은 어드레스(0x00에 가까운 어드레스)로 이동합니다. 따라서 두 레지스터의 초깃값을 어떻게 설정하는가에 따라서 스택의 크기가 결정됩니다.

이러한 내용을 참고하여 부트 로더에 스택을 생성해 보겠습니다. 먼저 스택으로 사용할 영역을 결정해야 하는데, 0x010000(64kb) 어드레스부터는 os 이미지가 로딩되므로 0x010000이하, 즉 0x0000:0000~0x0000:FFFF 영역을 사용하겠습니다. 따라서 스택 세그먼트 레지스터(ss)의 값을 0x0000으로 설정하겠습니다. 또한 스택은 넉넉한 것이 좋으므로 스택 포인터 레지스터(sp)를 0xFFFE로 설정하여, 스택 영역의 크기를 세그먼트의 최대 크기로 지정하겠습니다. why? memory aligment 때문에

메모리는 하드웨어 구성 특성상 주소버스 용량이 16비트일 경우에는 2바이트 단위

32비트일 경우에는 4바이트 단위 등으로 정렬해서 읽는게 속도가 더 빠릅니다.

즉 짝수 번지에서 2바이트를 읽을 땐 1번의 메모리 읽기 명령으로 읽는게 가능하고

홀수 번지에서 2바이트를 읽을 땐 1바이트 읽기 2번을 통해 2바이트를 읽습니다.

스택 포인터를 0xFFFF로 초기화하면

첫 PUSH 연산에 의해

sp가 2 감소하고 그 위치에 데이터를 삽입합니다.

0xFFFF - 0x2 = 0xFFFD

가 되고 이는 홀수 번지이므로 스택 연산이 계속 2번의 메모리 읽기를 통해 2바이트를 읽어버립니다.

따라서 임의로 0xFFFE로 설정하여 짝수번지에서 시작되도록 한 것입니다.

하지만 위처럼 설정하면 0xFFFE, 0xFFFF는 사용하지 않게 됩니다.

그래서 sp를 0x0000으로 설정하면 PUSH 연산이 일어나기전에 2가 감소되어

언더플로우에 의해 0xFFFE~0xFFFF에

2바이트가 들어가 실제로 0x0000~0xFFFF를 모두 사용할 수 있습니다.

리얼모드에서는 2바이트 레지스터를 사용한다고 볼 수 있습니다.

0xFFFE + 0x0002 = 0x0000 (오버 플로우)

0x0000 - 0x0002 = 0xFFFE (언더 플로우)

PUSH 연산은 SP값을 2 감소시킨 후 해당 주소에 데이터를 삽입.

따라서 SP가 0x0000일 때 PUSH 연산이 일어나면,

SP : 0x0000 - 0x0002 = 0xFFFE

따라서 0xFFFE ~ 0xFFFF에 2바이트 데이터가 삽입됩니다.

[스택 초기화 코드]

; 스택을 0x0000:0000~0x0000:FFFF 영역에 64kb 크기로 생성

mov ax, 0x0000 ; 스택 세그먼트의 시작 어드레스(0x0000)를 세그먼트 레지스터 값으로 변환

mov ss, ax ; ss 세그먼트 레지스터에 설정

mov sp, 0xFFFE ; sp 레지스터의 어드레스를 0xFFFE로 설정

mov bp, 0xFFFE ; bp 레지스터의 어드레스를 0xFFFE로 설정

push — Push stack (Opcodes: FF, 89, 8A, 8B, 8C, 8E, ...)

The push instruction places its operand onto the top of the hardware supported stack in memory. Specifically, push first decrements ESP by 4, then places its operand into the contents of the 32-bit location at address [ESP]. ESP (the stack pointer) is decremented by push since the x86 stack grows down - i.e. the stack grows from high addresses to lower addresses.

Syntax

push <reg32>

push <mem>

push <con32>

Examples

push eax — push eax on the stack

push [var] — push the 4 bytes at address var onto the stack

pop — Pop stack

The pop instruction removes the 4-byte data element from the top of the hardware-supported stack into the specified operand (i.e. register or memory location). It first moves the 4 bytes located at memory location [SP] into the specified register or memory location, and then increments SP by 4.

Syntax

pop <reg32>

pop <mem>

Examples

pop edi — pop the top element of the stack into EDI.

pop [ebx] — pop the top element of the stack into memory at the four bytes starting at location EBX.

leave - stack frame을 해제하는데 사용하는 명령어

mov esp,ebp

pop ebp 와 동일하다.

push와 pop 명령은 sp 레지스터와 관계가 있으므로 스택에 많은 데이터를 넣거나 빼야 한다면 push와 pop 명령 대신 데이터를 스택에 직접 복사하고 나서 sp, 레지스터의 값을 변경하는 방법으로 같은 결과를 얻을 수 있습니다.

지금까지 스택에 대해서 살펴봤으니 다시 화면 출력 함수로 돌아오겠습니다. 화면에서 원하는 위치에 문자열을 출력하려면 x 좌표, y 좌표, 출력할 문자열 어드레스가 필요합니다. 이 세 가지를 함수 파라미터로 정의하고 스택에 삽입하는 순서를 정하겠습니다. 만일 다른 라이브러리나 다른 언어와 같이 사용하지 않는다면 여러분이 스택에 삽입하는 순서를 임의로 정해도 괜찮습니다. 즉, 호출하는 쪽과 호출되는 쪽에서 같은 순서로 스택에서 데이터를 꺼내기만 하면 정상적으로 처리할 수 있습니다. 하지만, 앞으로 c 언어와 연계를 고려한다면 중복 작업을 피할 수 있게 c 언어의 호출 규약(cdecl)을 따르는 편이 좋습니다. c 언어는 파라미터의 역순(오른쪽에서 왼쪽)으로 삽입하여 스택에서 꺼낸 순서가 파라미터 순서와 같게 합니다. 다음 소스 코드는 c 언어와 어셈블리어의 함수 호출 코드를 비교한 것입니다. 파라미터는 오른쪽에서 왼쪽 방향으로 스택에 삽입하고, 함수 호출이 끝난 후에 스택을 정리하는 것을 볼 수 있습니다.(어셈블리어 코드에서 word는 메모리에 접근할 때 2바이트(word) 단위로 접근하라는 것을 의미합니다.)

[c 언어의 함수 호출 코드]

PrintMessage(iX,iY,pcString);

[어셈블리어의 함수 호출 코드]

push word[pcString]

push word[iY]

push word[iX]

call PRINTMESSAGE

add sp, 6

위의 코드에서 함수를 호출하고 난 뒤, 스택 포인터(sp) 레지스터에 6을 더하는 이유는 함수 파라미터로 스택에 삽입된 값을 제거하기 위함입니다. 16비트 모드에서는 스택에 2바이트(word) 크기로 삽입/제거되고 삽입은 스택 포인터 레지스터(sp)를 아래로 이동시킵니다. 따라서 파라미터 3개가 삽입되면 삽입되기 전의 위치에서 -6(2바이트\*3)만큼 이동할 것입니다. 함수 수행이 끝난 후, 스택을 다시 원래대로 복원하려면 감소한 만큼 더해주면 되므로 6을 더하는 것입니다.

함수를 호출하는 코드를 살펴봤으니 호출되는 쪽 코드를 살펴보겠습니다. 호출되는 함수는 파라미터가 순서대로 삽입되어 있다는 것을 이미 알고 있습니다. 따라서 스택의 특정 위치를 기준으로 오프셋을 이용해 접근하면 파라미터를 찾게 됩니다. 그런데 여기서 문제가 한 가지 있습니다. 스택의 상위(top)을 의미하는 스택 포인터 레지스터(sp)는 스택 관련 명령(push,pop)에 따라 계속 변한다는 것입니다. 스택에 삽입된 파라미터에 접근하려면 시시각각 변하는 스택 포인터 레지스터(sp)대신 스택에 고정된 값을 가리키는 레지스터를 사용하는 것이 편리합니다. 이러한 역할을 하는 것이 베이스 포인터 레지스터(bp)이며, 호출된 함수는 베이스 포인터 레지스터(bp)+오프셋으로 파라미터에 접근하게 됩니다.

호출되는 함수에 추가하는 부분은 파라미터에 접근하는 코드만이 아닙니다. 호출된 함수가 작업을 마치고 호출한 코드로복귀했을 때 코드가 정상적으로 수행되려면 호출되기 전후의 레지스터 상태가 같아야 합니다. 이를 위해서 호출되는 함수에서는 자신이 사용하는 레지스터의 값을 미리 스택에 저장해두고, 수행이 끝나면 이를 복원하여 호출한 이후의 코드 수행에 영향을 미치지 않아야 합니다. 함수의 이러한 특징 때문에 대부분 어셈블리어 함수는 다음과 같은 형태로 정형화되어 있습니다.

[어셈블리어 함수의 일반적인 형식]

push bp ; 베이스 포인터 레지스터(bp)를 스택에 삽입

mov bp, sp ; 베이스 포인터 레지스터(bp)에 스택 포인터 레지스터(sp)의 값을 설정

; 베이스 포인터 레지스터(bp)를 이용해서 파라미터에 접근할 목적

push es ; es 세그먼트 레지스터부터 dx 레지스터까지 스택에 삽입

push si ; 함수에서 임시로 사용하는 레지스터로 함수의 마지막 부분에서

push di ; 스택에 삽입된 값을 꺼내 원래 값으로 복원

push ax

push cs

push dx

- 생략 –

mov ax, word[bp+4] ; 파라미터 1(iX,화면 x 좌표)

mov bx, word[bp+6] ; 파라미터 2(iY,화면 y 좌표)

mov cx,word[bp+8] ; 파라미터 3(pcString, 출력할 문자열의 어드레스)

메모리에 접근할 때 2바이트(word) 단위로 접근함. 16비트 모드에서 스택은 2바이트 크기이다.

- 생략 –

pop dx ; 함수에서 사용이 끝난 dx 레지스터부터 es 레지스터까지를 스택에

pop cx ; 삽입된 값을 이용해서 복원

pop ax ; 스택은 가장 마지막에 들어간 데이터가 가장 먼저 나오는

pop di ; 자료구조이므로 삽입의 역순으로 제거해야 함

pop si

pop es

pop bp ; 베이스 포인터 레지스터(bp) 복원

ret ; 함수를 호출한 다음 코드의 위치로 복귀

보호 모드에서 사용되는 세 가지 함수 호출 규약

호출 규약(calling convention)은 함수를 호출할 떄 파라미터와 복귀 어드레스 등을 지정하는 규칙입니다. 보호 모드에서 사용하는 대표적인 호출 규약에는 stdcall, cdecl, fastcall이 있으며 약간씩 차이가 있습니다. stdcall 방식은 파라미터를 스택에 저장하며, 호출된 쪽에서 스택을 정리합니다. cdecl 방식도 역시 파라미터를 스택에 저장하지만, 함수를 호출한 쪽에서 스택을 정리합니다. fastcall 방식은 일부 파라미터를 레지스터에 저장하는 것을 제외하면 stdcall 방식과 같습니다.

보호 모드는 32비트이므로 레지스터와 스택의 크기가 리얼모드의 2배입니다. 따라서 스택의 기본 크기는 2바이트(word)가 아닌 4바이트(dwrod)가 됩니다. 앞서 설명했던 스택의 파라미터 접근 방법은 보호 모드나 리얼 모드가 같지만, 스택의 기본 크기가 2배이므로 오프셋이 2배를 해줘야 합니다. 레지스터 역시 크기가 리얼 모드의 2배ㅣ며, 16비트 레지스터 이름에 e 접두사가 붙은 eax, ebx, ecx, esp, ebp 레지스터는 4바이트 레지스터를 의미합니다.

함수 호출 예(c언어)

int add(int iA,int iB, int iC)

{

return iA+iB+iC;

}

void main(void)

{

int iReturn;

iReturn = add(1,2,3);

}

1. stdcall(standard call)

파라미터 삽입 – 오른쪽에서 왼쪽 순서

함수의 반환 값 – EAX 레지스터

파라미터 제거 담당 – callee

[함수 호출 예제(stdcall 방식)]

add: ; add 함수

push ebp

mov ebp, esp

mov eax, dword[ebp+8]

add eax, dword[ebp+12]

add eax, dword[ebp+16]

pop ebp

ret 12 ; ret 후, add esp,12 (스택 정상화)와 같은 역할을 한다.

main:

push ebp

mov ebp, esp

sub esp, 4 ; sp 레지스터에서 4만큼을 빼서 지역변수 iReturn을 위한 공간 할당

push 3

push 2

push 1

call Add

mov dword[ebp-4],eax ; iReturn 변수에 add 함수의 반환값 저장

leave

ret

2. cdecl(c-declare call)

파라미터 삽입 – 오른쪽에서 왼쪽 순서

함수의 반환 값 – EAX 레지스터

파라미터 제거 담당 – caller

[함수 호출 예제(cdecl 방식)]

add: ; add 함수

push ebp

mov ebp, esp

mov eax, dword[ebp+8]

add eax, dword[ebp+12]

add eax, dword[ebp+16]

pop ebp

ret

main:

push ebp

mov ebp, esp

sub esp, 4 ; sp 레지스터에서 4만큼을 빼서 지역변수 iReturn을 위한 공간 할당

push 3

push 2

push 1

call Add

mov dword[ebp-4],eax ; iReturn 변수에 add 함수의 반환값 저장

add esp, 12 ; 스택 정상화

leave

ret

3. fastcall

컴파이러마다 구현하는 방식이 조금씩 다릅니다. 윈도우에서 많이 쓰이는 마이크로소프트사의 컴파일러를 기준으로 설명하면 처음 2개의 파라미터를 ecx, edx 레지스터에 삽입하는 점을 제외하고는 stdcall과 같습니다.

파라미터 삽입 – 오른쪽에서 왼쪽 순서(레지스터 이용)

함수의 반환 값 – EAX 레지스터

파라미터 제거 담당 – callee

[함수 호출 예제(fastcall 방식)]

add: ; add 함수

push ebp

mov ebp, esp

mov eax, ecx

add eax, edx

add eax, dword[ebp+8]

pop ebp

ret 4

main:

push ebp

mov ebp, esp

sub esp, 4 ; sp 레지스터에서 4만큼을 빼서 지역변수 iReturn을 위한 공간 할당

push 3

mov edx, 2

mov ecx, 1

call Add

mov dword[ebp-4],eax ; iReturn 변수에 add 함수의 반환값 저장

leave

ret

IA-32e 모드의 호출 규약은 fastcall을 확장한 방식이며, 보다 많은 레지스터를 파라미터 전달용으로 사용합니다. Ia-32e 모드로 전환하면 기존 레지스터에 8개의 범용 레지스터(R8~R15 레지스터)가 추가됩니다. IA-32e 모드의 호출 규약은 기존 레지스터와 추가된 레지스터를 포함하여 파라미터를 최대 6개까지 전달할 수 있도록 설계되었기 때문에, 파라미터 개수만 제한하여 사용한다면 스택 관련 작업을 줄일 수 있습니다.

5.3 테스트를 위한 가상 os 이미지 생성

가상 os 이미지는 5장에서만 사용하고 후에 실제 os 이미지로 대체하므로 세세한 부분까지 구현하지 않아도 됩니다. 따라서 부트 로더 코드를 기반으로 기능을 간소화하여 os가 실행되었음을 표시하는 기능만 넣겠습니다. 자신의 섹터 번호를 화면 위치에 대응시켜서 0~9까지 번호를 출력한다면 화면에 출력된 문자의 위치와 수를 확인하여 정상여부를 판단할 수 있습니다.

[그림 5-5] 가상 os 이미지를 통한 부팅과 테스트

0x100000(1M) -------------------------------------

0x90000 -------------------------------------

가상 os 이미지( 1024)

-------------------------------------

……..

0x10400 -------------------------------------

가상 os 이미지( 2)

0x10200 -------------------------------------

가상 os 이미지( 1)

0x10000 -------------------------------------

스택 영역

0x7E00 -------------------------------------

부트 로더 (0)

0x7C00 -------------------------------------

0 -------------------------------------

nasm 전처리기는 매크로, 조건, 반복 구문을 지원하며 변수 할당과 값 지정도 가능합니다.

1024 섹터의 가상 os 이미지를 만드는 방법은 1024 섹터 중에 마지막 섹터를 제외한 1024 섹터의 코드는 화면에 자신을 출력하는 코드 및 다음 섹터의 어드레스로 이동하는 코드를 반복하면 됩니다. 그리고 마지막 섹터 하나는 더 이상 섹터가 없으므로 자신을 출력하고 무한 루프를 수행하도록 하면 끝입니다.

리얼 모드에서 bios 서비스를 최대한 활용하려다 보니 os 이미지가 최대 1152섹터라는 제약사항이 생겼습니다. 사실 1152섹터라고는 했지만 bios 제조사에 따라서 비디오 메모리 어드레스의 앞쪽을 다른 용도로 사용할 수 있기 때문에 실제로는 이보다 더 작을 수도 있습니다. 따라서 os 이미지가 아주 크다면 os 이미지를 압축하거나, bios 서비스로 os의 일부만 메모리에 로딩하고 나머지 부분은 보호 모드로 전환한 후 디바이스에 직접 접근하는 방식으로 해결해야 합니다. 다행히도 mint64 os의 최종 크기는 응용프로그램을 포함해서 570kb 정도이므로 bios 서비스만으로도 충분합니다.

Part2. 64비트 세상으로 점프

6장 32비트 보호 모드로 전환하자

리얼 모드에서 보호 모드로 전환하려면 [그림 6-1]과 같이 크게 6단계를 거쳐야 합니다. 그중에서 상위 2단계는 보호 모드 전환에 필요한 자료 구조를 생성하는 단계이며, 나머지 4단계는 생성된 자료구조를 프로세서에 설정하는 단계입니다. 보호 모드에서 반드시 생성해야 하는 자료구조는 세그먼트 디스크립터와 GDT이며, 두 가지 자료구조는 보호 모드로 전환하는 즉시 프로세서에 의해 참조되므로 미리 생성해둬야 합니다. 자료구조를 생성하는 두 단계를 제외한 나머지 단계는 프로세서에 자료 구조를 설정하거나 보호 모드로 전환하고 나서 필요한 초기화 작업을 수행합니다.

[그림 6-1] 보호 모드로 전환하는 여섯 가지 단계

세그먼트 디스크립터 생성

(보호 모드 코드와 데이터용 세그먼트 디스크립터 생성)

↓

GDT 정보 생성

(세그먼트 디스크립터의 시작 어드레스와 디스크립터의 전체 크기 저장)

↓

프로세서에 GDT 정보 설정

(GDTR 레지스터에 GDT의 시작 어드레스와 크기 설정)

↓

CR0 컨트롤 레지스터 설정

(CR0 컨트롤 레지스터의 PE 비트=1, PG 비트=0)

↓

jmp 명령으로 cs 세그먼트 셀렉터 변경과 보호 모드로 전환

(jmp 0x08:보호 모드 커널의 시작 어드레스) 16비트 리얼 모드

-----------------------------------------------------------------------------------------------------------------------------------------

↓ 32비트 보호 모드

각종 세그먼트 셀렉터 및 스택 초기화

(DS,ES,FS,GS,SS 세그먼트 셀렉터와 esp,ebp 레지스터 초기화)

↓

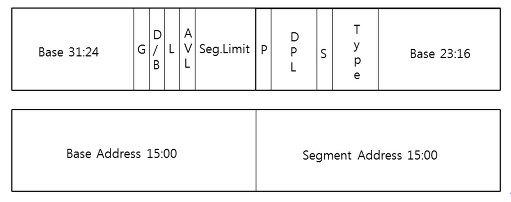
보호 모드 커널 실행

6.1 세그먼트 디스크립터 생성

세그먼트 디스크립터(segment descriptor)는 세그먼테이션 기법(메모리 관리 기법)에서 세그먼트의 정보를 나타내는 자료구조입니다. 세그먼트는 메모리 공간을 임의의 크기로 나눈 영역을 의미하며, 세그먼트를 복잡하게 구성할수록 세그먼트 디스크립터의 수도 증가합니다. MINT64 OS에서는 보호 모드의 기본 기능만 사용하므로 4GB 전체 메모리 공간을 지정하는 커널 코드와 데이터 세그먼트만 사용합니다.

세그먼트에 대한 정보를 나타내는 세그먼트 디스크립터는 크게 코드 세그먼트 디스크립터와 데이터 세그먼트 디스크립터로 나누어집니다. 코드 세그먼트 디스크립터는 실행 가능한 코드가 포함된 세그먼트에 대한 정보를 나타내며, cs 세그먼트 셀렉터에 의해 가리켜집니다. 데이터 세그먼트 디스크립터는 데이터가 포함된 세그먼트에 대한 정보를 나타내며, cs 세그먼트 셀렉터를 제외한 나머지 셀렉터에 사용할 수 있습니다. 스택 영역 또한 데이터를 읽고 쓰는 데이터 영역의 한 종류이므로 데이터 세그먼트 디스크립터를 사용합니다.

[그림 6-2] 세그먼트 디스크립터의 구조



[표 6-1] 세그먼트 디스크립터의 필드와 의미

필드 설명

기준 주소 세그먼트의 시작 어드레스

32비트 크기이며, 0바이트~4GB까지 설정 가능

세그먼트 크기 세그먼트의 크기

20비트크기

G비트가 0으로 설정되면 0~1MB까지 설정 가능

G비트가 1로 설정되면 4KB를 곱하여 0~4GB까지 설정 가능

타입 세그먼트의 타입

코드 또는 데이터 세그먼트로 설정 가능

설정 가능한 값은 [표 6-2] 참조

S 디스크립터의 타입

1로 설정할 경우 세그먼트 디스크립터를 나타내며, 0으로 설정하면 시스템 디스크립터를 나타냄

DPL Descriptor Privilege level의 약자로 해당 디스크립터를 사용에 필요한 권한을 의미

0(highest)~3(lowest)의 범위를 가짐

CPL(current privilege level)나 RPL(requested priveilege level)과 조합되어 접근 권한을 제한하는 데 사용

P Present의 의미로 현재 디스크립터가 유효한 디스크립터인지 표시

1로 설정하면 유효한 디스클립터임을 나타내며, 0으로 설정하면 유효하지 않은 디스크립터임을 나타냄

AVL available의 약자로 os가 임의의 용도로 사용할 수 있는 영역

L IA-32e 모드에서 사용하는 필드로 64비트 코드 세그먼트나 32비트 코드 세그먼트를 의미

1로 설정하면 IA-32e 모드의 64비트용 코드 세그먼트임을 나타내며, 0으로 설정하면 IA-32e 모드의 32비트 호환 모드용 코드 세그먼트임을 나타냄

D/B Default operation size를 의미

1로 설정하면 32비트용 세그먼트임을 나타내며, 0으로 설정하면 16비트용 세그먼트임을 나타냄

G granularity의 약자로 디스크립터의 세그먼트 크기 필드에 곱해질 가중치를 의미

1로 설정하면 세그먼트 크기에 4KB를 곱하며, 0으로 설정하면 가중치를 곱하지 않음

MINT64 os에 필요한 세그먼트는 다음과 같습니다.

커널 코드와 데이터용 세그먼트 디스크립터 각 1개

커널 코드와 데이터용 세그먼트는 0~4GB까지 모든 영역에 접근할 수 잇어야함

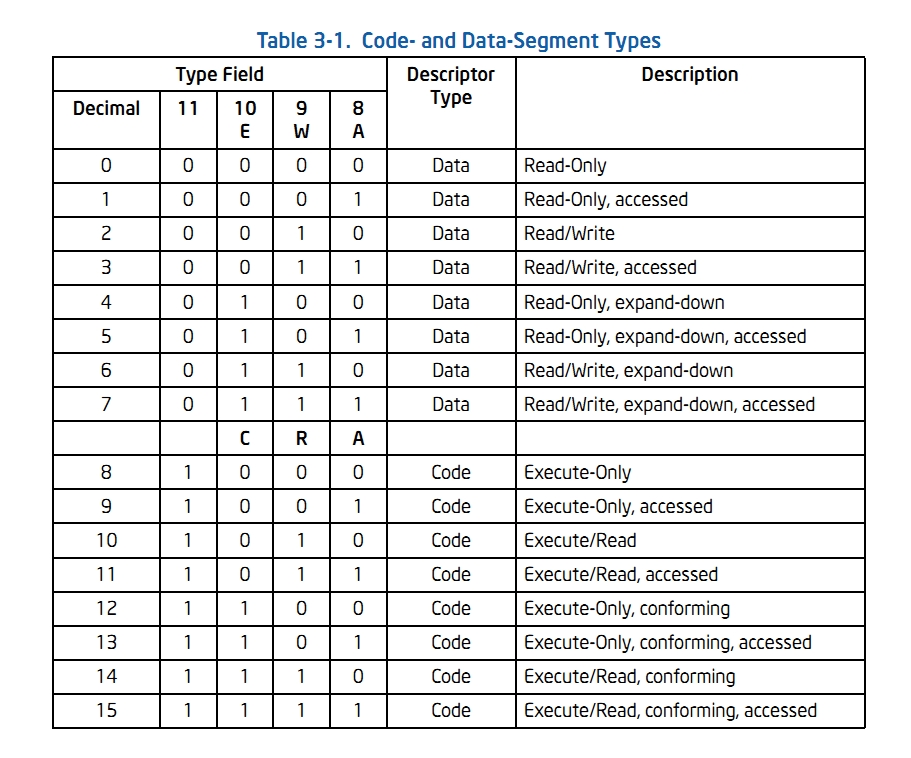
보호 모드용 코드와 데이터에 사용할 기본 오퍼랜드 크기는 32비트여야 함

보호 기능은 사용하지 않으며, 프로세서의 명령을 사용하는 데 제약이 없어야 하므로 최상위 권한(0)이어야 함

6.1.1 코드 세그먼트 디스크립터와 데이터 세그먼트 디스크립터 타입 설정

코드 세그먼트와 데이터 세그먼트를 설정하려면 S 필드와 타입 필드를 조합해야 합니다. 우선 s 필드부터 살펴보겠씁니다. 코드 세그먼트와 데이터 세그먼트는 세그먼트 디스크립터이므로 간단하게 s 필드의 값을 1로 설정합니다. 세금너트 타입은 4비트 크기의 타입 필드를 이용해서 설저앟며 각 비트의 값은 [표 6-2]와 같은 의미를 나타냅니다. MINT64 OS에서는 기본적인 세그먼트 타입만 사용하고, 코드 세그먼트는 실행/읽기 타입으로 설정하고 데이터 세그먼트는 읽기/쓰기 타입으로 설정합니다. [표 6-2]에서 찾아보면 코드 세그먼트 타입은 0x0A(execute/read), 데이터 세그먼트 타입은 0x02(read/write)가 됩니다.

[표 6-2] 타입 필드의 값과 의미



세그먼트 타입의 비트 8에 있는 접근 여부 비트는 프로세서에서 설정하는 비트로 프로세서는 해당 디스크립터가 참조될 때마다 비트 8을 1로 설정합니다. 이를 이용하면 비트 8을 이용해서 특정 세금너트 디스크립터가 사용되었는지 여부를 확인할 수 있습니다.

비트 10에 있는 역방향 확장 비트는 아래로(상위 어드레스->하위 어드레스) 자라는 스택을 위한 옵션이며, 스택 세그먼트의 크기는 동적으로 확장 또는 축소할 목적으로 사용합니다. 역방향 확장 기능을 사용하면 세금너트 크기(limit) 값이 기준 주소에서 아래 방향(0바이트) 방향을 적용되며, 세그먼트의 범위는 (세그먼트의 기준 주소 – 세그먼트 크기) ~ 세그먼트의 기준 주소가 됩니다. 따라서ㅏ 스택의 변화에 맞추어 세그먼트 크기를 동적으로 변경할 수 있습니다.

여기 같은 비트 10에 위치한 접근 승인 비트는 권한에 관계없이 해당 코드 세그먼트에 접근할 수 있음을 나타냅니다. 승인 기능을 사용하면 애플리케이션도 커널 코드를 직접 실행할 수 있으며, 커널과 애플리케이션 간에 코드 공유가 필요할 때도 요긴하게 쓸 수 있습니다.

6.1.2 세그먼트의 영역 설정

MINT64 os의 커널 세그먼트 디스크립터는 4GB 전체 영역에 접근할 수 있어야 합니다. 그러므로 커널용 세그먼트 디스크립터의 기준 주소는 0으로 설정합니다. 크기 필드는 총 20비트며 20비트로 표현할 수 잇는 최댓값은 2^20(1MB)입니다. 크기 필드만으로는 4GB까지의 영역을 표현할 수 없으므로 20비트의 크기를 4GB로 확장할 무엇인가가 필요합니다. 이때 사용하는 것이 G 필드이며, G 필드의 값을 1로 설정하면 크기 필드에 4KB를 곱한 것이 실제 세그먼트의 크기가 됩니다. 1MB에 4KB를 곱하면 4GB가 되므로 크기 필드와 G 필드를 사용하면 메모리 전체 영역을 세그먼트의 영역으로 설정할 수 있습니다.

6.1.3 기본 오퍼랜드 크기와 권한 설정

보호 모드는 32비트로 동작하므로 기본 오퍼랜드의 크기 역시 32비트로 설정합니다. 여러 필드 중에 기본 오퍼랜드의 크기는 D/B 필드가 담당하며, 1로 설정하면 기본 오퍼랜드의 크기를 32비트로 설정할 수 있습니다. 기본 오퍼랜드의 크기와 관계된 필드가 D/B 필드만 있는 것은 아닙니다. IA-32e 모드의 64비트 서브 모드 또는 32비트 호환 모드를 설정하는 L 필드도 있습니다. 6장에서 생성하는 디스크립터는 보호 모드용이므로 L 비트는 0으로 설정합니다.

권한 필드는 보호 모드의 주요 특징 중 하나인 보호 기능에 핵심 역할을 합니다. 프로세서는 디스크립터의 권한 필드에 설정된 값과 세그먼트 셀렉터의 권한을 비교하여 접근이 가능한지를 판단하며, x86 프로세서에서 동작하는 운영체제의 대부분도 이 기능을 사용하여 os의 핵심 부분을 보호하고 있습니다. MINT64 OS의 보호 모드는 권한을 따로 구분하지 않으므로 권한 필드를 모두 최상위 권한(0)으로 설정합니다.

6.1.4 기타 필드 설정

생성한 세금너트 디스크립터는 보호 모드로 전환하는 과정에서 사용하므로 유효한 디스크립터라는 것을 알려야 합니다. 디스크립터가 유효함을 나타내는 필드는 p, 필드이며, 1로 설정하면 해당 디스크립터를 사용할 수 있습니다.

AVL 필드는 임의로 사용할 수 있는 필드로 MINT64 OS에서는 별도의 값을 쓰지 않으므로 0으로 설정합니다.

보호 모드는 현대 os가 제공하는 4GB의 주소 공간, 멀티 태스킹, 페이징, 메모리 보호 등의 기능을 하드웨어적으로 지원합니다.

6.2 GDT 정보 생성

GDT(global descriptor table) 자체는 연속된 디스크립터의 집합입니다. 즉 MINT64 os에서 사용하는 코드 세그먼트 디스크립터와 데이터 세그먼트 디스크립터를 연속된 어셈블리어 코드로 나타내면 그 전체 영역이 GDT가 됩니다. 다만, 한 가지 제약 조건이 있다면 널 디스크립터(Null descriptor)를 가장 앞부분에 추가해야 한다는 것이니다. 널 디스크립터는 프로세서에 의해 예약된 디스크립터로 모든 필드가 0으로 초기화된 디스크립터이며 일반저긍로 참조되지 않습니다. No segment register should be set to 0, otherwise this will cause a General Protection fault, and is a protection feature of the processor.

GDT는 디스크립터의 집합이므로 프로세서에 GDT의 시작 어드레스와 크기 정보를 로딩해야 합니다. 따라서 이것을 저장하는 자료구조가 필요하며, 이 자료구조는 [그림 6-4]와 같은 구조를 하고 있습니다.

[그림 6-4] GDT 정보를 저장하는 자료 구조

47 1615 0

32비트 기준 주소 크기

GDT 정보를 저장하는 자료구조의 기준 주소는 32비트의 크기이며, 데이터 세그먼트의 기준 주소와 관계 없이 어드레스 0을 기준으로 하는 선형 주소입니다. 따라서 GDT의 시작 어드레스를 실제 메모리 공간상의 어드레스로 변환할 필요가 있습니다. GDT의 선형 주소는 현재 코드가 실행되고 있는 세그먼트의 기준 주소를 알고 있으므로 현재 세그먼트의 시작을 기준으로 GDT의 오프셋을 구하고, 세그먼트 기준 주소를 더해주면 구할 수 있습니다. 현재 코드는 부트 로더에 의해 0x10000에 로딩되어 실행되고 있으므로 자료구조를 생성할 때 GDT 오프셋에 아래의 코드와 같이 0x10000을 더해주면 선형 주소가 됩니다.

GDT 정보를 담는 자료 구조에서 GDT의 크기를 나타내는 필드가 2바이트(16비트)이므로, GDT에 포함 가능한 디스크립터(8바이트 크기) 수는 65536/8이 되어 최대 8192개가 됩니다. 얼핏 생각하기에는 충분한 크기 같지만 멀티태스킹을 위해 태스크 디스크립터와 태스크별 세그먼트 디스크립터, 기타 디스크립터를 생성하다 보면 공간이 부족할 수 있습니다. 그래서 x86 프로세서는 이런 문제를 해결하고자 GDT와 비슷한 역할을하는 LDT(local descriptor table)를 제공합니다. LDT 역시 GDT와 같이 디스크립터를 포함하는 테이블이며, 크기도GDT와 같이 최대 8192개로 제한되어 있습니다. LDT를 사용하면 태스크에 관련된 세그먼트 디스크립터나 기타 시스템 디스크립터를 GDT에서 LDT로 옮길 수 있으므로 더 많은 태스크를 생성할 수 있다는 장점이 있습니다.

MINT64 os는 멀티 태스킹에 핵심적인 역할을 하는 태스크 스위칭 기능을 소프트웨어 적으로 구현하여, x86 프로세서의 하드웨어 기능에 의존하지 않습니다. 따라서 태스트 증가에 따른 GDT 사용량이 많지 않아 LDT를 사용하지 않습니다.

6.3 보호 모드로 전환

보호 모드로 전환하려면 GDTR 레지스터 설정, CR0 컨트롤 레지스터 설정, jmp 명령 수행 3단계만 수행하면 됩니다.

6.3.1 프로세서에 GDT 정보 설정

[GDT 정보를 프로세서에 설정하는 코드]

lgdt [ GDTR ] ; GDTR 자료구조를 프로세서에 설정하여 GDT 테이블을 로드

보호 모드로 전환하는 과정과 전환 후 인터럽트 설정을 완료하기 전까지는 인터럽트가 발생하지 않게 하는 것이 좋습니다. 보호 모드로 전환하여 인터럽트에 관련된 설정이 완료되기 전에 인터럽트가 발생하면 프로세서는 인터럽트 처리 함수(인터럽트 핸들러)를 찾을 수 없습니다. 따라서 예기치 못한 문제가 발생할 수 있으며, 최악의 상황에는 리부팅될 수도 있습니다.

6.3.2 CR0 컨트롤 레지스터 설정

CR0 컨트롤 레지스터에는 보호 모드 전환에 관련된 필드 외에 캐시(cache), 페이징(paging), 실수 연산 장치(FPU) 등과 관련된 필드가 포함되어 있습니다.

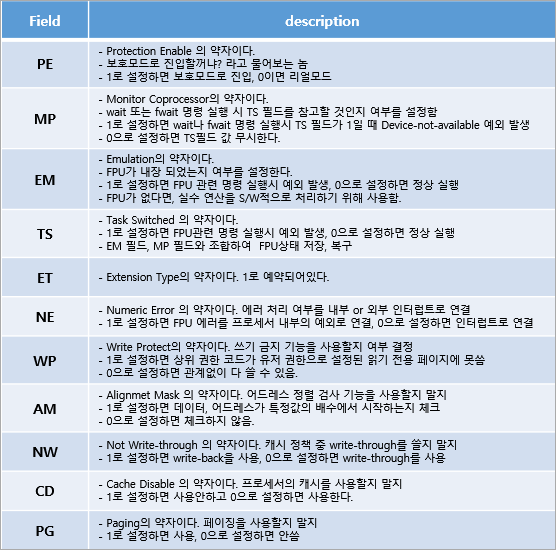
[그림 6-5] CR0 컨트롤 레지스터의 구조와 보호 모드 전환을 위한 설정 값

31 30 29 28 19 18 17 16 15 6 5 4 3 2 1 0

PG CD NW 예약된 영역 AM WP 예약된 영역 NE ET TS EM MP PE

0x 4 0 0 0 0 0 3 B

[표 6-3] CR0 컨트롤 레지스터의 필드와 의미



MINT64 os에서 보호 모드는 거쳐 가는 임시 모드에 불과하므로 세그먼테이션 기능 외에는 사용하지 않습니다. 따라서 페이징, 캐시, 메모리 정렬 검사, 쓰기 금지 기능은 모두 사용하지 않음으로 설정하면 됩니다. FPU 역시 쓰지 않으므로 임시 값으로 설정합니다. FPU에 관련된 필드를 제외한 나머지 필드는 해당 필드를 설정하는 것만으로 관련 기능을 제어할 수 있습니다. 하지만 FPU에 관련된 필드(EM,ET,MP,TS,NE)는 서로 연관되어 있으므로 FPU 관련 필드를 설정하는 방법에 대해서 잠시 알아보겠습니다.

먼저, FPU 내장 여부에 관련된 필드부터 설정하겠습니다. x86 프로세서에서는 FPU가 내장되어 있으므로 EM 필드를 0으로 설정해서 FPU 명령을 소프트웨어로 에물레이션하지 않게 하고, ET 필드를 1로 설정합니다. 지금은 임시로 초기화를 수행한 것이므로 FPU를 사용하면 정상적으로 동작하지 않습니다. 따라서 MP 필드와 TS 필드와 NE 필드를 1로 설정하여 FPU 명령이 실행되었을 때 예외가 발생하게 설정합니다. 보호 모드에서는 예외에 대한 처리를 하지 않으므로 가능하면 실수 연산을 하지 않는 것이 좋습니다.

캐시의 write-back 정책은 메모리를 통해 I/O를 수행하는 메모리 맵 I/O 방식을 사용하는 디바이스에는 문제가 발생할 수 있습니다. 메모리 맵 I/O 방식은 메모리 어드레스에 읽고 쓰는 데이터를 데이터를 디바이스로 송수신하는 방식이며, write-back 방식을 사용하면 외부 메모리에 업데이트가 바로 되지 않아서 문제가 생깁니다. 따라서 MINT64 os는 IA-32e 모드에 진입해서 모든 초기화 적업이 끝나고 나서 캐시를 활성화하고 있습니다.

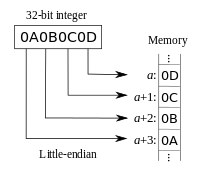
6.4 보호 모드용 커널 이미지 빌드와 가상 os 이미지 교체

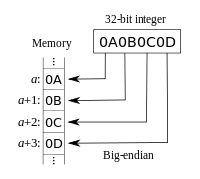
엔트리 포인트(entry point)는 외부에서 해당 모듈을 실행할 때 실행을 시작하는 지점을 의미합니다. 6장에서 작성한 코드는 부트 로더(외부)에서 보호 모드 커널로 진입하는 부분이므로, 보호 모드 커널의 엔트리 포인트라고 할 수 있습니다.

makefile 매크로

$< : dependency의 첫 번쨰 파일을 의미하는 매크로

$^ : dependency에 나열된 전체 파일을 의미하는 매크로





7장 c언어로 커널을 작성하자

7.1 실행 가능한 c코드 커널 생성 방법

이번 장에서 해야할일은 C 소스파일을 추가하고, 이를 빌드하여 보호 모드 커널 이미지에 통합하는 것입니다. 실제로 하는 일은 C언어로 작성한 커널을 보호 모드 엔트리 포인트의 뒷부분에 연결하고 엔트리 포인트에서는 C커널의 시작부분으로 이동하는 것이 전부입니다.

C코드는 어셈블리어 코드와 달리 컴파일과 링크과정을 거쳐서 최종결과물이 생성됩니다. 컴파일 과정은 소스 파일을 중가 단계인 오브젝트 파일로 변화하는 과정으로 소스파일을 해석하여 코드영역과 데이터 영역으로 나누고 이러한 메모리 영역에 대한 정보를 생성하는 단계입니다. 링크 단계는 오브젝트 파일들의 정보를 취합하여 실행 파일에 통합하며, 필요한 라이브러리 등을 연결해주는 역할을 하는 단계입니다.

[그림 7-1] MINT64 OS의 보호 모드 커널 이미지 생성 단계

엔트리 포인트 소스 파일 어셈블리어 소스 파일 c언어 소스 파일

컴파일 ------------------------------------------------------------------------------------------------------------------

nasm nasm gcc

↓ ↓ ↓

링크 ---------------------------------------------------------------------------------------------------------------------

nasm ld ld

바이너리 파일 ↓ ↓

실행 파일

↓ ↓ objcopy

바이너리 파일

↓

파일 결합 ----------------------------------------------------------------------------------------------------------------

커널 이미지 파일

7.1.1 빌드 조건과 제약 사항

엔트리 포인트가 c코드를 실행하려면 적어도 아래의 세가지 제약 조건을 만족해야 합니다.

1. c라이브러리를 사용하지 않게 빌드해야 한다. 부팅된 후 보호 모드 커널이 실행되면 c 라이브러리가 없으므로 라이브러리에 포함된 함수를 호출할 수 없습니다. 만약 커널 코드에서 애플리케이션 코드처럼 printf() 함수를 사용한다면 c 라이브러리를 동적 링크 또는 정적 링크하게 됩니다. 하지만 커널은 자신을 실행하기 위한 최소한의 환경만 설정하므로 라이브러리의 함수가 정상적으로 실행되지 않습니다. 따라서 작성된 커널 코드만 사용하도록 빌드해야합니다.

2. 0x`10200위치에서 실행하게끔 빌드해야 합니다. ox1000 의 위치에는 6장에서 작성한 한 섹터 크기의 보호 모드 엔트리 포인트가 있으므로, 결합된 c 코드는 512바이트 이후인 0x10200의 위치부터 로딩됩니다. 따라서 c로 작성한 커널 부분은 빌드할 때 0x10200 위치에서 실행되는 것을 전제로 해야 하며, 해당 위치의 코드는 c코드 중에 가장 먼저 실행되어어야하는 함수(엔트리 포인트)가 위치해야 합니다. 커널이 실행되는 어드레스가 중요한 이유는 선형 주소를 참조하게 생성된 코드나 데이터때문입니다. c언어에서 전역 변수의 어드레스나 함수의 어드레스를 참조하는 경우, 실제로 존재하는 선형 주소로 변환됩니다. 따라서 메모리에 로딩되는 어드레스가 변한다면, 이러한 값들 역시 변경해줘야 정상적으로 동작합니다. 가령 다음과 같은 c 코드가 있을 때, 0x0000d에서 로딩되어 실행되는 경우와 0x10200에서 로딩되어 실행되는 경우의 어셈블리어 코드는 아래와 같습니다.(g\_iIndex 변수는 메모리 어드레스의 가장 앞쪽에 위치한다고 가정했습니다.)

메모리에 로딩되는 어드레스에 따라 다르게 생성된 코드의 예

///////////////////////////////

// c 코드

//////////////////////////////

int g\_iIndex=0;

void AddIndex(void)

{

g\_iIndex++;

}

/////////////////////////////////////

/ 0x0000에 로딩된 어셈블리어 코드

////////////////////////////////////

--생략—

g\_iIndex: DD 0x00000000 <- g\_iIndex 변수는 로딩된 메모리 어드레스 처음에 위치한다고 가정

AddIndex:

mov eax, dword[0x0000]

add eax,1

mov dword[0x0000], eax

////////////////////////////////////

/ 0x10200에 로딩된 어셈블리어 코드

///////////////////////////////////

g\_iIndex: DD 0x00000000

AddIndex:

mov eax, dword[0x10200]

add eax,1

mov dword[0x10200],eax

위의 코드를 보면 메모리에 로딩되는 어드레스에 따라 전역 변수의 어드레스에 접근하는 부분이 변한다는 것을 알 수 있습니다. 이러한 이유 때문에 커널이 0x10200의 어드레스에서 실행되게 빌드하는 것이 필요합니다.

3. 코드나 데이터 외에 기타 정보를 포함하지 않은 순수한 바이너리 파일 형태여야 한다는 것입니다. 일반적으로 여러분이 설치한 gcc를 통해 실행 파일을 생성하면 elf 파일 포맷이나 pe 파일 포맷과 같이 특정 os에서 실행할 수 있는 포맷으로 생성됩니다. 이러한 파일 포맷들은 실행하는데 필요한 코드와 데이터 정보 이외의 불필요한 정보를 포함하고 있습니다. 해당 파일 포맷을 그대로 사용하게 되면 엔트리 포인트에서 파일 포맷을 해석하여 해당 정보에 따라 처리하는 기능이 포함되어야 하므로 코드가 복잡해집니다. 만일 부트 로더나 보호 모드 엔트리 포인트처럼 코드와 데이터만 포함된 바이너리 파일의 형태를 사용한다면 엔트리 포인트에서 해당 어드레스로 점프하는 것만으로 c 코드를 실행할 수 있습니다.

7.1.2 소스 파일 컴파일 – 라이브러리를 사용하지 않는 오브젝트 파일 생성 방법

c 코드를 컴파일하여 오브젝트 파일을 생성하는 방법은 아주 간단합니다. 이미 2장에서 gcc 컴파일러를 설치했으므로 gcc의 옵션으로 ‘-c’만 추가하면 됩니다. 이 옵션은 소스 파일을 오브젝트 파일로 변환하는 컴파일 단게까지만 처리하는 것을 의미합니다. 여러분이 ‘-o’ 옵션으로 직접 오브젝트 파일 이름을 지시하지 않는 한 생성되는 오브젝트 파일의 이름은 c 파일 이름에 확장자만 .o로 변경되어 생성됩니다.

앞서 언급했듯이 우리가 생성하는 오브젝트 파일은 라이브러리를 사용하지 않고 홀로(freestanding) 동작할 수 있는 형태여야 합니다. gcc 컴파일러는 이러한 오브젝트 파일을 위해 ‘-ffreestanding’ 옵션을 지원하고 있으며 ‘-c’ 옵션을 함께 조합하면 라이브러리를 사용하지 않는 오브젝트 파일을 생성할 수 있습니다.

[main.c를 라이브러리를 사용하지 않는 main.o로 컴파일 하는 방법]

gcc –c –m32 –ffreestanding main.c

7.1.3 오브젝트 파일 링크 – 라이브러리를 사용하지 않고 특정 어드레스에서 실행 가능한 커널 이미지 파일 생성 방법

오브젝트 파일을 링크하여 실행 파일을 만드는 방법은 소스 파일을 컴파일하는 방법보다 까다롭습니다. 실행 파일을 구성하는 섹션의 배치와 로딩될 어드레스, 코드 내에서 가장 먼저 실행될 코드인 엔트리 포인트를 지정해줘야 하기 때문입니다. 특히 섹션을 배치하는 작업은 오브젝트 파일이나 실행 파일 구조와 관련이 있으므로 다른 작업보다 좀 더 까다로울 수 있습니다. 하지만, 섹션을 배치하는 방식과 크기 정렬 방식에 따라서 os의 메모리 구조와 크기가 달라지므로 꼭 한번은 읽고 넘어가기 바랍니다.

섹션 배치를 다시 하는 이유는 실행 파일이 링크될 때 코드나 데이터 이외에 디버깅 관련 정보와 심볼(symbol, 함수나 변수의 이름) 정보 등이 포함되기 때문입니다. 이러한 정보들은 커널을 실행하는 데 직접적인 관련이 없으므로 최종 바이너리 파일을 생성할 때 이를 제거하려고 섹션을 재배치하는 것 입니다. 섹션을 재배치하여 코드와 데이터를 실행 파일 앞쪽으로 이동시키면 손쉽게 나머지 부분을 제거할 수 있습니다.

섹션 배치와 링커 스크립트, 라이브러리를 사용하지 않는 링크

섹션(section)은 실행 파일 또는 오브젝트 파일에 있으며 공통된 속성(코드, 데이터, 각종 심볼과 디버깅 정보 등)을 담는 영역을 뜻합니다. 실행 파일이나 오브젝트 파일에는 무수히 많은 섹션이 있지만 핵심 역할을 하는 섹션은 세 가지가 있습니다.

1. 실행 가능한 코드가 들어 있는 .text 섹션 .text 섹션은 여러분이 작성한 main()이나 함수의 실제 코드가 저장되는 영역입니다. 프로그램이 실행되면 코드를 수정할 일이 거의 없으므로 일반적으로 읽기 전용(read-only) 속성을 가집니다.

2. 초기화된 데이터가 들어 있는 .data 섹션 .data 섹션은 0이 아닌 값으로 초기화된 전역 변수(global varible) 혹은 0이 아닌 값으로 초기화된 정적 변수(static variable) 등을 포함하며, 데이터를 저장하는 섹션이기 때문에 일반적으로 읽기/쓰기 속성을 가집니다.

3. 초기화되지 않은 데이터가 들어 있는 .bss 섹션입니다. .bss 섹션에 포함되는 데이터는 .data에 포함되는 데이터와 거의 같으나 초기화되지 않은 변수만 포함한다는 것이 차이점입니다. .bss 섹션은 0으로 초기화될 뿐, 실제 데이터가 없으므로 실행파일이나 오브젝트 파일 상에는 별도의 영역을 차지하지 않습니다. 하지만, 메모리에 로딩되었을 때 코드는 해당 영역 변수들의 초깃값이 0이라고 가정합니다. 따라서 정상적으로 프로그램을 실행하려면 메모리에 로딩할 때, .bss 영역을 모두 0으로 초기화해야 합니다.

BSS (from Block Started by Symbol)

In C, statically-allocated objects without an explicit initializer are initialized to zero (for arithmetic types) or a null pointer (for pointer types). Implementations of C typically represent zero values and null pointer values using a bit pattern consisting solely of zero-valued bits (though this is not required by the C standard). Hence, the BSS segment typically includes all uninitialized objects (both variables and constants) declared at file scope (i.e., outside any function) as well as uninitialized static local variables (local variables declared with the static keyword); static local constants must be initialized at declaration, however, as they do not have a separate declaration, and thus are typically not in the BSS section, though they may be implicitly or explicitly initialized to zero. An implementation may also assign statically-allocated variables and constants initialized with a value consisting solely of zero-valued bits to the BSS section.

Peter van der Linden, a C programmer and author, says, "Some people like to remember it as 'Better Save Space.' Since the BSS segment only holds variables that don't have any value yet, it doesn't actually need to store the image of these variables. The size that BSS will require at runtime is recorded in the object file, but BSS (unlike the data segment) doesn't take up any actual space in the object file."

소스 코드를 컴파일 하여 생성한 오브젝트 파일은 각 섹션의 크기와 파일 내에 있는 오프셋 정보만 들어 있습니다.

왜냐하면, 오브젝트 파일은 중간 단계의 생성물로 다른 오브젝트 파일과 합쳐지기 때문입니다 .합쳐지는 순서에 따라서 섹션의 어드레스는 얼마든지 바뀔 수 있습니다.

오브젝트 파일들을 결합하여 정리하고 실제 메모리에 로딩될 위치를 결정하는 것이 바로 링커(linker)이며, 이러한 과정을 링크(link) 또는 링킹(linking)이라고 부릅니다.

링커의 주된 역할은 오브젝트 파일을 모아 섹션을 통합하고, 그에 따라 어드레스를 조정하며, 외부 라이브러리에 있는 함수를 연결해주는 것 입니다. 하지만, 두드리기만 하면 금은 보화가 나오는 요술 방망이가 아니므로 링커가 실행 파일을 만들려면 파일 구성에 대한 정보가 필요합니다. 바로 이때 사용하는 것이 링커 스크립트(linker script)입니다.

링커 스크립트에는 각 센션의 배치 순서와 시작 어드레스, 섹션 크기 정렬 등의 정보를 저장해 놓은 텍스트 형태의 파일로 굉장히 복잡한 내용이 담겨 있습니다.

천만 다행인 것은 기존 링크 스크립트에서 섹션 배치와 섹션 크기 정렬에 대한 내용만 수정하면 된다는 것입니다. 따라서 섹션을 배치하고 정렬하는 데 필요한 기본적인 내용만 간단히 살펴보겠습니다.

[링커 스크립트의 기본 형식]

SECTIONS

{

SectionName Load Address: ; 섹션 이름과 메모리에 로드할 어드레스

{

\*(text) ; 오브젝트 파일의 섹션 중에 SectionName에 통합할 섹션 이름

… 생략 …

. = ALIGN (AlignValue) ; 현재 어드레스를 Align Value에 맞추어 이동, 다음 섹션의 시작은 AlignValue의 배수 . 은 현재 어드레스를 나타내는 기호

} =0x00000000 ; 섹션을 채울 기본 값

}

섹션 크기 정렬 부분은 ALIGN() 부분의 값을 수정함으로써 변경할 수 있습니다. 크기 정렬 값은 임의의 값으로 설정해도 괜찮지만, 편의상 데이터 섹션의 시작을 섹터 크기(512바이트)에 맞추겠습니다. 이후에 커널의 공간이 부족하다면 이 값ㅇ르 더 작게 줄임으로써 보호 모드 커널이 차지하는 비중을 줄일 수 있습니다.

[링커 스크립트를 직접 지정하여 오브젝트 파일을 링크하는 예]

x86\_64-pc-linux-ld –melf\_i386 –T elf\_i386.x –nostadlib main.o –o main.elf

-melf\_i386 : 2장에서 크로스 컴파일한 Bintuils가 기본적으로 64비트 코드를 생성하므로 32비트 실행 파일을 위해 설정한 옵션

-T elf\_i386.x : elf\_i386.x 링커 스크립트를 이용해서 링크 수행

-nostdlib : stdlib를 사용하지 않고 링크 수행

로딩할 메모리 어드레스와 엔트리 포인트 지정

어셈블리어로 작성된 부트 로더나 보호 모드 엔트리 포인트처럼 c 코드 역시 로딩될 메모리를 미리 예측하고 그에 맞춰 이미지를 생성하는 것이 중요합니다. 만약 이미지를 로딩할 어드레스에 맞춰서 생성하지 않는다면 전역 변수와 같이 선형 어드레스를 직접 참조하는 코드는 모두 잘못된 어드레스에 접근하기 때문입니다.

메모리에 로딩하는 어드레스를 지정하는 방법은 1. 링커 스크립트를 수정하는 방법과 2. 링커(ld) 프로그램의 명령줄 옵션으로 지정하는 방식 두 가지가 있습니다. 링커 스크립트를 통해 수정하려면 스크립트 파일의 ‘.text’ 섹션을 아래와 같이 수정합니다. ‘text’ 섹션의 어드레스를 수정하면 그 이후에 있는 ‘data’와 ‘bss’ 같은 섹션은 자동으로 ‘text’가 로딩되는 어드레스 이후에 계싼되며, 0x10000의 어드레스에는 512바이트 크기의 보호 모드 엔트리 포인트(EntryPoint.s) 코드가 있으니 c 코드는 0x10200 어드레스부터 시작할 것입니다. [그림 7-3]은 보호 모드 엔트리 포인트와 c언어 커널이 결합된 이미지가 로딩된 메모리의 배치를 나타낸 것입니다.

[그림 7-3] 새롭게 생성된 디스크 이미지의 메모리 배치

0x100000(1M) -------------------------------------------

0x010800 -------------------------------------------

보호 모드 c언어 커널 영역 (Kernel32.bin)

0x010200 -------------------------------------------

보호 모드 엔트리 포인트 영역(EntryPoint.bin)

0x010000 -------------------------------------------

부트 로더의 스택 영역

0x007E00 -------------------------------------------

부트 로더 영역

0x007C00 -------------------------------------------

[링커 스크립트를 수정하여 로딩될 메모리 어드레스를 지정하는 예]

…생략…

.text 0x10200 : .text 섹션을 0x10200에 로딩하도록 지정

{

…

}

[커멘드 라인 옵션을 통해 로딩될 메모리 어드레스를 지정하는 예]

x86\_64-pc-linux-ld –Ttext 0x10200 main.o –o main.elf

엔트리 포인트 역시 링커 스크립트 또는 커맨드 라인 옵션으로 지정할 수 있습니다. 링크 스크립트를 통해 지정하려면 스크립트 파일의 상단에 있는 ENTRY() 부분을 다음과 같이 수정합니다.

[링커 스크립트를 수정하여 엔트리 포인트 지정하는 예(main 함수를 엔트리 포인트로 지정)]

…생략…

OUTPUT\_ARCH(i386)

ENTRY(main)

SEARCH\_DIR(“/usr/cross/x86\_64-pc-linux/lib”);

…생략…

[커맨드 라인 옵션을 통한 엔트리 포인트 지정]

x86\_64-pc-linux-ld –e main main.o –o main.elf

사실 엔트리 포인틀르 링커에 지정하는 작업은 빌드의 결과물이 os에 의해 실행 가능한 파이 포맷(리눅스의 elf 파일 포맷, 윈도의 pe 파일 포맷 등)일 때만 의미가 있습니다. 실행 파일을 바이너리 형태로 변환하는 MINT64 OS의 경우는 엔트리 포인트를 정보가 제거되므로 엔트리 포인트는 큰 의미가 없으며, 단순히 링크 시에 발생하는 경고(warning)를 피하려고 설정한 것입니다. 하지만 앞서 설명했듯이 0x10000 어드레스에 존재하는 보호 모드 엔트리 포인트는 0x10200 어드레스로 이동(jmp)하므로 , c 코드의 엔트리 포인트를 해당 어드레스에 강제로 위치시킬 필요가 있습니다.

그럼 어떻게 해야 특정 함수를 가장 앞쪽에 위치시킬 수 있을까요? 특정 함수를 실행 파일의 가장 앞쪽에 두려면 두 가지 순서를 조작해야 합니다.

1. 오브젝트 파일 내의 함수 간의 순서입니다. 오브젝트 파일은 소스 파일로부터 생성되고, 컴파일러는 특별한 옵션이 없는 한 소스 파일에 정의도니 함수의 순서대로 오브젝트 파일의 내용을 생성합니다. 따라서 c 소스 파일을 수정하여 엔트리 포인트 함수를 가장 상위로 옮겨줌으로써 오브젝트 파일에 포함된 함수의 순서를 변경할 수 있습니다.

2. 실행 파일 내의 함수 간의 순서입니다. 컴파일러와 마찬가지로 실행 파일은 오브젝트 파일로부터 생성되고, 링커는 특별한 옵션이 없는 한 입력으로 주어진 오브젝트 파일의 순서대로 통합하여 실행 파일을 생성합니다. 따라서 엔트리 포인트가 포함된 오브젝트 파일을 가장 앞쪽으로 옮겨줌으로써 c 코드의 엔트리 포인트를 0x10200에 위치시킬 수 있습니다.

실행 파일을 바이너리 파일로 변환

컴파일과 링크 과정을 거쳐 생성된 실행 파일은 코드 섹션과 데이터 섹션 이외의 정보를 포함하므로 이를 제거하여 부트 로더나 보호 모드 엔트리 포인트와 같이 순수한 바이너리 파일 형태로 변환해야 합니다. 따라서 실행 파일에서 불필요한 섹션을 제외하고 꼭 필요한 코드 섹션과 데이터 섹셕만 추출해야 하는데, objcopy 프로그램을 사용하면 이러한 작업을 손쉽게 처리할 수 있습니다.

objcopy는 실행 파일 또는 오브젝트 파일을 다른 포맷으로 변환하거나 특정 섹션을 추출하여 파일로 생성해주는 프로그램으로 binutils에 포함되어 있습니다. objcopy는 옵션이 굉장히 많지만 섹션을 추출하여 바이너리로 바꾸는 작업만 수행하면 되므로 –j,-S,-O 옵션에 대해서만 알아보겠습니다.

-j 옵션은 실행 파일에서 해당 섹션만 추출하는 옵션이며, .text 섹션만 추출하려면 –j .text와 같이 사용하면 됩니다.

-S 옵션은 실행 파일에서 재배치 정보와 심볼을 제거하는 옵션이며, c언어 커널은 함수 이름이나 변수 이름을 사용할 일이 없으므로 제거합니다.

-O 옵션은 새로 생성할 파일의 포맷을 지정하는 옵션이며, 실행 파일을 바이너리 파일 포맷으로 변환하려면 –O binary처럼 사용합니다. 다음은 Kernel32.elf 파일에서 코드 섹션과 데이터에 관련된 섹션만 추출하여 바이너리 형태의 Kernel32.bin 파일을 만드는 예를 나타낸 것입니다.

[objcopy로 바이너리 파일 변환]

x86\_64-pc-linux-objcopy –j .text –j .data –j .rodata –j .bss –S –O binary Kernel32.elf Kernel32.bin

링커 스크립트의 OUTPUT\_FORMAT() 항목과 SectionName 항목을 잘 이용하면 objcopy 프로그램을 사용하지 않고도 바이너리 파일을 직접 생성할 수 있습니다. 아래와 같이 OUTPUT\_FORMAT()에 ‘binray’로 지정하고, .text .data .bss 섹션을 제외한 나머지 섹션에 이름을 /DISCARD/로 교체하면 됩니다.

링크 스크립트를 수정해서 직접 바이너리 파일을 생성하면 한 단계를 줄일 수 있어서 빌드 시간이 단축됩니다. 하지만, 중간 과정에서 생성된 elf 파일을 통해 여러 가지 정보를 얻을 수 있으며, elf 파일을 바이너리 파일로 변환하는데 시간이 오래 걸리지 않으므로 elf 파일을 거쳐서 생성하는 것이 좋습니다.

7.2 c 소스 파일 추가와 보호 모드 엔트리 포인트 통합

지금까지 오브젝트 파일을 링크하고 원하는 파일 포맷으로 변환하는 방법을 알아봤습니다. 이제는 보호 모드 커널 디렉터리에 c 소스 파일을 추가하고 자동으로 포함하여 빌드하는 방법을 살폅겠습니다. 그리고 환영 메시지를 출력하는 c 코드를 추가하여 보호 모드 엔트리 포인트와 통합해 보겠습니다.

7.2.1 c 소스 파일 추가

7.2.2 보호 모드 엔트리 포인트 코드 수정

7.2.3 makefile 수정

7장부터는 다수의 파일을 컴파일하고 링크해야 하므로 makefile을 좀 더 편리하게 수정할 필요가 있습니다. 따라서 make의 몇 가지 유용한 기능을 사용하여 Source 디렉터리에 .c 확장자의 파일만 추가하면 자동으로 포함하여 빌드하게 수정하겠습니다.

. c 파일을 자동으로 빌드 목록에 추가하려면, 매번 빌드를 수행할 때 마다 Source 디렉터리에 있는 \*.c파일을 검색하여 소스 파일 목록에 추가해야 합니다. make에서는 이러한 작업을 위해 디렉터리에 있는 파일을 검색하는 와일드카드 기능을 제공합니다. Source 디렉터리에 있는 \*.c 파일을 모두 검색해서 CSOURCEFILES이라는 변수에 넣고 싶다면 와일드 카드를 사용하여 다음과 같이 입력합니다.

[Source 디렉터리내의 모든 .c 파일을 CSOURCEFILES 변수로 지정하는 예]

CSOURCEFILES = $(wildcard Source/\*.c)

Source/\*.c 디렉터리에 확장자가 .c인 모든 파일을 의미

wildcard 디렉터리에 특정 패턴의 파일을 추출하는 wildcard 함수

디렉터리에 있는 모든 c파일을 검색했으니, 이제 이 파일들에 대한 빌드 룰만 정해주면 자동으로 빌드할 수 있습니다. 지금까지의 makefile은 각 파일에 대해 빌드 룰을 개별적으로 기술했습니다. 하지만, 빌드에 필요한 파일이 수백 개쯤 된다면 여러분이 아무리 부지런하다 해도 관리하기가 어렵습니다. 또한 팡리이 추가되고 삭제될 때마다 룰을 변경해야 하는데 실수하면 빌드 오류나 실행 도중 예기치 못한 오류가 발생할 수 있습니다.

이러한 문제는 파일 패턴에 대해 동일한 룰을 적용함으로써, 간단히 처리할 수 있습니다. 가령 모든 .c 파일은 gcc –c 라는 컴파일 과정을 통해 .o 파일로 변환된다면, 다음과 같이 써서 모든 .파일을 .o 파일로 컴파일할 수 있습니다.

[확장자가 .c인 파일을 .o로 변경하는 룰을 지정하는 예]

%.o : %.c

gcc –c $<

A pattern rule contains the character ‘%’ (exactly one of them) in the target; otherwise, it looks exactly like an ordinary rule. The target is a pattern for matching file names; the ‘%’ matches any nonempty substring, while other characters match only themselves.

For example, ‘%.c’ as a pattern matches any file name that ends in ‘.c’. ‘s.%.c’ as a pattern matches any file name that starts with ‘s.’, ends in ‘.c’ and is at least five characters long. (There must be at least one character to match the ‘%’.) The substring that the ‘%’ matches is called the stem.

‘%’ in a prerequisite of a pattern rule stands for the same stem that was matched by the ‘%’ in the target. In order for the pattern rule to apply, its target pattern must match the file name under consideration and all of its prerequisites (after pattern substitution) must name files that exist or can be made. These files become prerequisites of the target.

Thus, a rule of the form

%.o : %.c ; recipe…

specifies how to make a file n.o, with another file n.c as its prerequisite, provided that n.c exists or can be made.

와일드 카드와 패턴 룰 기능을 이용하면, Source 디렉터리 내의 모든 C 파일을 자동으로 컴파일할 수 있습니다. 그럼 이제 검색된 C 파일을 이용하여 링크할 파일 목록을 생성해 보겠습니다. 일반적으로 오브젝트 파일은 소스 파일과 같은 이름이며 확장자만 .o로 변경되므로 소스 파일 목록에 포함된 파일의 확장자를 .c에서 .o로 수정하면 됩니다. 특정 문자를 치환하려면 patsubst 기능을 사용하면 되고, patsubst는 $(patsubst 수정할 패턴, 교체할 패턴, 입력 문자열)의 형식으로 사용합니다.

[CSOURCEFILES 내의 목록의 확장자를 .c에서 .o로 수정하여 COBJECTFILES 변수로 지정하는 예]

COBJECTFILES = $(patsubst %.c,%.o,$(CSOURCEFILES))

$(patsubst pattern,replacement,text)

Finds whitespace-separated words in text that match pattern and replaces them with replacement. Here pattern may contain a ‘%’ which acts as a wildcard, matching any number of any characters within a word. If replacement also contains a ‘%’, the ‘%’ is replaced by the text that matched the ‘%’ in pattern. Only the first ‘%’ in the pattern and replacement is treated this way; any subsequent ‘%’ is unchanged.

하지만, 이것이 끝이 아닙니다. 우리는 c 커널 엔트리 포인트 함수를 가장 앞쪽에 배치하려면 엔트리 포인트 오브젝트 파일을 COBJECTFILES의 맨 앞에 둬야 합니다. 만일 c 커널의 엔트리 포인트를 포함하는 오브젝트 파일 이름이 main.o라고 가정하면, main.o 파일을 COBJECTFILES에서 맨 앞에 두려면 다음과 같이subst를 사용합니다.

[subst 기능으로 main.o 오브젝트 파일을 가장 먼저 링크 하는 예]

CENTRYPOINTOBJECTFILE = main.o

COBJECTFILES = $(patsubst %.c,%.o,$(CSOURCEFILES))

COTHEROBJECTFILES = $(subst main.o, , $(COBJECTFILES))

Kernel32.elf: $(CENTRYPOINTOBJECTFILE) $(COBJECTFILES)

x86\_64-pc-linux-ld –o $@ $^

이와 같은 규칙은 어셈블리어 파일에도 마찬가지로 적용할 수 있습니다. 보호 모드 커널과 IA-32e 모드 커널에서 사용할 어셈블리어 파일은 .asm으로 생성할 예정이므로 이를 고려하여 수정하겠습니다. 앞에서 설명한 makefile의 내용 중에서 크게 바뀌는 부분은 없으며 .c 부분만 .asm으로 수정하고 gcc 컴파일러 옵션 대신 nasm을 사용하게 변경하면 끝입니다. 단 , 컴파일된 어셈블리어 오브젝트 파일과 c언어 오브젝트 파일은 같이 링크되어야 하므로 이를 고러하여 컴파일 옵션을 설정해야 합니다. gcc의 오브젝트 파일은 elf32 파일 포맷 형태를 따르며 nasm의 오브젝트 파일 역시 동일한 포맷으로 생성되게 컴파일 옵션에 –f elf32를 추가합니다.

특정 오브젝트 파일을 가장 먼저 링크하는 방법은 커맨드 라인의 가장 앞쪽에 두는 방법도 있지만, 링커 스크립트의 .text 섹션 부분을 수정해도 결과가 같습니다. 다음은 기존의 링커 스크립트를 수정하여 main.o를 가장 먼저 링크하게 만든 예입니다.

.text 0x10200

{

main.o(text)

…

}

디렉터리에 있는 모든 c 소스 파일을 포함하는 작업은 make의 기능을 사용해서 간단히 처리할 수 있습니다. 하지만, 어디까지나 c 소스파일에만 해당되는 내용입니다. c언어는 헤더 파일(Header File)을 정의하여 소스 파일에서 공통으로 사용하는 데이터 타입이나 함수의 선언을 모아두고, 이를 참조할 수 있습니다. 이는 소스 파일의 내용뿐 아니라 헤더 팡리이 수정되어도 소스 파일을 다시 빌드해야 함을 의미합니다. 이를 위해서는 소스 파일을 모두 검사하여 포함하는 헤더 파일을 모두 makefile의 dependency에 기록해야 합니다. (내가 지금까지 해온 방법)

그렇다면 어떻게 해야 소스 파일에 관련된 헤더 파일을 찾을 수 있을까요? 간단히 프로그램을 작성해서 소스 파일의 #include 부분을 읽어서 처리해야 할까요? 다행히도 gcc의 옵션 중에 makefile용 규칙을 만들어 주는 전처리기 관련 옵션(-M)을 사용하면, 자동으로 헤더 파일을 추출할 수 있습니다. 그중에서 –MM 옵션을 사용하면 stdio.h와 같은 시스템 헤더 파일을 제외한 나머지 헤더 파일에 대한 의존 관계를 출력할 수 있습니다. 따라서 -MM옵션을 이용하여 소스 코드를 모두 검사하고 그 결과를 파일로 저장하면, 소스 파일별 헤더 파일의 의존 관계(Dependency)를 확인할 수 있습니다. 다음은 main.c와 test.c 소스 파일의 의존 관계를 구해 dependency.dep 파일로 저장하는 예입니다.

[main.c와 text.c의 의존 관계를 Dependency.dep 파일로 저장하는 예]

x86\_64-pc-linux-gcc –MM main.c test.c > Dependency.dep

이렇게 생성한 Dependency.dep 파일을 makefile에 포함해야 각 파일의 의존 관계를 분석하여 정확한 빌드를 수행할 수 있습니다. make는 수행 시 다른 makefile을 포함하는 기능을 제공하며, include 지시어가 바로 그러한 역할을 담당합니다. 그렇다면 무조건 include Dependency.dep를 수행하면 될까요? include 지시어는 해당 파일이 없으면 에러를 발생시킵니다. 따라서 최초 빌드 시나 오브젝트 파일을 정리하고 나서 다시 빌드할 때 dependency.dep 파일이 없으면 빌드 에러가 발생할 수 있습니다. 이를 피하려면 현재 디렉터리를 검사해서 dependency.dep 파일이 있을 때만 포함해야 합니다. 이러한 작업은 make의 조건문과 wildcard 함수를 조합하면 됩니다. 다음은 현재 디렉터리에 있는 dependency.dep 파일이 존재할 때만 include를 수행하는 makefile의 일부입니다.

[현재 디렉터리에 dependency.dep 파일이 있는 경우에만 include를 수행하는 예]

ifeq (dependency.dep, $(wildcard depdendency.dep))

include dependency.dep

endif

mint64 os의 커널 디렉터리는 소스 디렉터리(Source)와 임시 디렉터리(Temp)로 다시 구분되며, 커널 빌드 작업은 임시 디렉터리를 기준으로 수행합니다. 따라서 dependency.dep 파일의 내용과 경로를 같게 하려면 make를 수행하는 디렉터리를 변경하는 옵션 –C를 이용하여 임시 디렉터리로 변경한 후 makefile를 수행합니다. 최종 결과물인 보호 모드 커널 이미지는 컴파일과 링크 과정이 끝난 후에 보호 모드 엔트리 포인트와 바이너리로 변환된 c커널을 결합하여 생성합니다.

The include directive tells make to suspend reading the current makefile and read one or more other makefiles before continuing

4.11 Multiple Rules for One Target

One file can be the target of several rules. All the prerequisites mentioned in all the rules are merged into one list of prerequisites for the target. If the target is older than any prerequisite from any rule, the recipe is executed.

There can only be one recipe to be executed for a file. If more than one rule gives a recipe for the same file, make uses the last one given and prints an error message.

makefile:6: warning: ignoring old commands for target `f1.o'

단 %.o: %.c와 같은 패턴 매치의 경우 include가 있을 경우, include의 .o파일은 패턴 매치 되지 않는다. 따라서, include에 –MM 옵션 dependency 다음에 명령어가 나타나는 경우, %.o:%.c 의 명령어가 실행되지 않는다. 반면, include에 –MM 옵션 dependency 만 존재하는 경우 %.o:%.c의 명령어가 실행된다.

target이 없고 tab으로 안되있는 명령어의 경우 단순히 다른 dependency와 합쳐지게 된다.

$(notdir names…)

Extracts all but the directory-part of each file name in names. If the file name contains no slash, it is left unchanged. Otherwise, everything through the last slash is removed from it.

A coprocessor is a computer processor used to supplement the functions of the primary processor (the CPU). Operations performed by the coprocessor may be floating point arithmetic, graphics, signal processing, string processing, encryption or I/O Interfacing with peripheral devices. By offloading processor-intensive tasks from the main processor, coprocessors can accelerate system performance

x87 is a floating point-related subset of the x86 architecture instruction set. It originated as an extension of the 8086 instruction set in the form of optional floating point coprocessors that worked in tandem with corresponding x86 CPUs. These microchips had names ending in "87". This was also known as the NPX (Numeric Processor eXtension). Like other extensions to the basic instruction set, x87-instructions are not strictly needed to construct working programs, but provide hardware and microcode implementations of common numerical tasks, allowing these tasks to be performed much faster than corresponding machine code routines can. The x87 instruction set includes instructions for basic floating point operations such as addition, subtraction and comparison, but also for more complex numerical operations, such as the computation of the tangent function and its inverse, for example.

Most x86 processors since the Intel 80486 have had these x87 instructions implemented in the main CPU but the term is sometimes still used to refer to that part of the instruction set. Before x87 instructions were standard in PCs, compilers or programmers had to use rather slow library calls to perform floating-point operations, a method that is still common in (low-cost) embedded systems.

8장. A20 게이트를 활성화하여 1MB 이상 영역에 접근해보자.

8.1 IA-32e 모드 커널과 메모리 맵

7장에서 c언어로 커널을 작성하는 방법에 대해서 알아보았습니다. 또한, makefile을 수정하여 Source디렉터리의 소스 파일을 자동으로 검색한 뒤, 이를 바탕으로 빌드하는 방법도 알아보았습니다. 이 장에서는 IA-32e 모드 커널을 실행하기 위한 준비 작업으로 pc에 설치된 메모리가 64MB 이상인지 검사하고, IA-32e 모드 커널이 위치할 영역을 모두 0으로 초기화하는 작업을 수행하겠습니다. 또한, 부팅 과정을 완료하고 나서 1MB 이상의 메모리에 정상적으로 접근되는지 확인하고, 이를 위해 어떤 작업이 필요한지 살펴보겠습니다.

우선 왜 1MB 이상의 메모리에 접근해야 하는지부터 살펴보겠습니다. 7장에서 설명했듯이 부트 로더에 의해 커널 이미지가 메모리에 로딩되는 어드레스는 0x10000입니다. 만약 1MB 이하의 어드레스중에서 비디오 메모리가 위치하는 0xA0000 이하를 커널 공간으로 사용한다고 가정하면 보호 모드 커널과 IA-32e 모드 커널의 clheo 크기는 0xA0000-0x10000이 되어 576KB 정도가 됩니다. 커널 이미지(보호 모드 커널과 IA-32e 모드 커널 포함)에는 초기화되지 않는 (.bss 섹션)이 포함되지 않으므로 커널 이미지의 크기로 환산하면 이보다 더 작은 크기여야 합니다. 지금처럼 커널이 별다른 기능을 하지 않는다면 문제가 안되지만, 이후 멀티 태스킹과 파일 시스템과 같은 기능이 추가되어 커널이 커진다면 576KB의 공간이 부족할 수 있습니다.

MINT64 OS는 이러한 문제를 해결하려고 커널 이미지를 모두 0x10000 어드레스에 복사하되, 덩치가 큰 IA-32e 모드 커널은 2MB의 어드레스로 복사하여 2MB~6MB의 영역을 별도로 할당했습니다. 따라서 IA-32e 모드의 커널 영역은 모든 섹션을 포함하여 총 4MB의 크기가 되며, 이는 MINT64 OS에게 충분한 크기라고 할 수 있습니다. [그림 8-1]은 이러한 메모리 맵을 나타낸 것입니다.

[그림 8-1] MINT64 OS의 메모리 맵

0x700000(7M) -----------------------------------------------

IA-32e 모드의 커널 스택 영역

0x600000(6M) -----------------------------------------------

IA-32e 모드의 c언어 커널 영역

(Kernel63.bin)

0x200000(2M) ----------------------------------------------- 커널 이미지 복사와 실행

IA-32e 모드의 커널 자료 구조 영역

0x100000(1M) -----------------------------------------------

IA-32e 모드의 C언어 커널 이미지 영역

(Kernel64.bin)

보호 모드의 C언어 커널 영역

(Kernel32.bin)

0x010200 -----------------------------------------------

보호 모드의 엔트리 포인트 영역

(EntryPoint.bin)

0x010000 -----------------------------------------------

보호 모드의 커널 및 부트 로더의 스택 영역

0x007E00 -----------------------------------------------

부트 로더 영역

0x007C00 -----------------------------------------------

0x000000 -----------------------------------------------

그렇다면 왜 IA-32e 모드 커널이 위치할 영역을 0으로 초기화할까요? 1MB 이하의 공간에서 IA-32e 모드 커널 이미지를 2MB의 어드레스로 복사한다면, 애써 초기화 한 부분이 커널 이미지로 덮어써 질 텐데 말입니다. 그럼에도 미리 초기화하는 이유는 IA-32e 커널 이미지가 초기화되지 않은 영역을 포함하고 있지 않기 때문입니다. 비록 커널은 초기화되지 않은 영역을 사용하지않지만, 커널 이미지에는 초기화 되지 않은 영역이 제외되어 있습니다. 따라서 커널 이미지를 옮길 때도 이 영역은 해당되지 않으며, 이미지를 옮길 영역을 미리 0으로 초기화하지 않는다면 어떤 임이의 값이 들어 있을 거십니다. 이러한 상태에서 IA-32e 모드 커널이 실행되면 0으로 참조되어야 할 변수들이 0이 아닌 값으로 설정되어, 루프를 빠져 나오지 못한다든지 잘못도니 조건문이 실행된다든지 하는 문제가 발생할 수 있습니다. 이러한 사태를 미연에 방지하려고 먼저 0으로 초기화하는 것입니다.

지금부터 IA-32e 모드 커널의 메모리 공간을 0으로 초기하는 방법을 살펴보겠습니다.

8.2 IA-32e 모드 커널을 위한 메모리 초기화

이번 절에서는 IA-32e 모드 커널이 위치할 영역을 초기화하는 방법을 살펴보겠습니다. 그리고 커널 영역을 초기화하는 코드를 추가하여 빌드한 뒤 os를 실행했을 때 발생하는 문제와 그 원인에 대해서 알아보겠습니다.

8.2.1 메모리 초기화 기능 추가

7장의 [예제 7-2]에서 main.c(01.Kernel32/Source) 파일을 추가하여 c언어로 작업할 수 있는 환경을 마련했으니, 1MB부터 6MB 영역까지를 모두 0으로 초기화하는 기능은 c코드로 구현해보겠습니다. 사실 1MB~2MB 영역은 IA-32e 모드 커널이 위치할 곳은 아니지만, IA-32e 모드 커널을 위한 자료구조가 위치할 영역이므로 같이 초기화하겠습니다. 초기화 코드를 main() 함수 내부에 직접 추가해도 되지만 main() 함수는 앞으로도 해야 할 일이 많으므로 초기화를 수행하는 kInitializeKernel64Are() 함수를 추가하여 이를 호출하겠습니다. 아래는 변경 사항이 반영도니 main.c 파일입니다. InitializeKernel64Address() 함수는 루프르르 수해앟면서 1MB~6MB 영역을 모두 0으로 채우는 단순한 코드이므로 이해하는데 어려움이 없을 것입니다.

8.2.2 빌드와 실행

이제 코드가 완성되었으니 make를 입력하여 os 이미지를 생성하고 나서 qemu와 실제 pc에서 이를 실행해 보겠습니다.

실행하면 qemu처럼 정상적으로 실행되거나 [그림 8-2]의 아래와 같이 메시지가 출력되지 않거나(IA-32e Kernel Area Initialization Complete 메시지) 또는 pc가 리부팅될 것입니다.

qemu에서 정상적으로 동작한다면 실제 pc에서도 정상적으로 동작해야 할 텐데, 왜 이런 문제가 발생하는 것일까요? 이것은 qemu가 실제 pc처럼 동작하지 않음을 보여주는 좋은 예입니다. 이는 qemu 한계라고도 볼 수 있으므로 수시로 실제 pc를 사용하여 테스트하여 정확하게 확인해야 합니다.

그렇다면 실제 pc에서 이런 증상이 나타나는 이유가 무엇일까요? 이 문제의 원인은 pc가 하위 기종에 대한 호환성을 유지하기 위해 어드레스 라인을 비활성화했기 때문입니다. 어드레스 라인에 대한 내용은 그리 간단하지 않으므로 다음 절에서 설명하겠습니다.

8.3 1MB 어드레스와 A20 게이트

A20 게이트는 pc의 역사와 관계가 있다 해도 과언이 아닙니다. A20은 어드레스 라인의 번호를 의미하지만, 실제로는 이름 이상으로 중요한 내용이 담겨 있습니다. 이번 절에서는 A20 게이트를 간단히 살펴본 뒤 시스템 포트와 BIOS 서비스를 이용하여 A20 게이트를 활성화해 보겠습니다.

8.3.1 A20 게이트의 의미와 용도

초창기 XT PC는 어드레스를 최대 1MB 어드레스까지 접근할 수 있었습니다. 하지만, 리얼 모드에서 세그먼트와 오프셋으로 접근할 수 있는 최대 어드레스는 0xFFFF:0xFFFF로 1MB가 넘는 0x10FFEF까지 접근할 수 있습니다. 하드웨의 한계로 1MB가 넘는 어드레스로 접근하는 경우 하위 어드레스만 남아 실제로는 0xFFEF로 인식되었습니다

XT PC가 주로 사용되던 시절에는 이러한 어드레스의 한계를 이용하여 작성된 프로그램이 문제없이 동작했습니다. 하지만 이후에 16MB 어드레스까지 접근할 수 있는 AT PC가 탄생하면서 기존의 XT PC용 프로그램을 AT PC에서 실행하는데 문제가 생겼습니다. 기존의 XT PC 프로그램 중에서 XT PC의 특수한 어드레스 계산법(1MB 이상의 어드레스를 1MB 이하의 어드레스에 매핑)을 이용하는 프로그램들 때문이었습니다. 이러한 호환성의 문제를 해결하려고 궁여지책으로 도입된 것이 A20게이트입니다.

A20 게이트에서 A20의 의미는 어드레스의 21번째 비트를 뜻하며, A20의 역할은 어드레스 21번째 비트를 활성화하거나 비활성화하여 XT PC의 어드레스 계산 방식과 호환성을 유지시킵니다. A20 게이트가 비활성화되면 어드레스 라인의 21번째(1M의 위치)가 항상 0으로 고정되므로 선형 주소가 0x10FFEF가 되더라도 0xFFEF로 처리할 수 있습니다.

AT PC는 부팅 과정을 완료하고 나서 A20 게이트를 무조건 0으로 설정하여 XT PC와 호환성을 유지했으며, A20 게이트를 활성화했을 떄만 21번째 어드레스 비트가 정상적으로 동작하게 했습니다. 이것이 지금까지 이어져 이전 절에서 보았던 문제를 일으킨 것이니다. [그림 8-3]은 A20 게이트의 비활성화에 따른 선형 주소의 변화에 대해서 나타낸 것입니다.

[그림 8-3] A20 게이트 비활성화에 따른 선형 주소의 변화

3MB + 64KB = 0x310000 코드에서 접근하는 선형주소

0011 0001 0000 0000 0000 0000

A2- 게이트가 비활성화된 경우 비트 21 어드레스 라인을 0으로 고정

0010 0001 0000 0000 0000 0000 실제로 접근하는 선형 주소

2MB + 64KB = 0x210000

A20 게이트가 비활성화된 상태에서는 어드레스 라인의 21번째 비트가 항상 0으로 설정되므로 홀수 MB에는 접근할 수 없습니다. 최초 부팅된 이후의 상태는 A20 게이트가 비활성화된 상태이므로 1MB~4MB까지의 어드레스를 초기화하면 홀수 MB 영역을 제외한 0~1MB와 2~3MB영역을 초기화하게 됩니다. 하지만, 0~1MB 영역은 BIOS와 보호 모드 커널 영역으로 사용하고 있으므로 현재 커널이 수행 중인 부분이 초기화하게 되어 문제가 생깁니다. [그림 8-4]는 A20 게이트 비활성화에 따라 접근 가능한 메모리 공간의 변화를 나타낸 것입니다.

그렇다면 문제 없이 실행되는 경우는 어떤 상황일까요? 이는 간혹 최신 BIOS중에서 A20게이트를 사용하지 않거나 기본 값으로 1로 설정하기 때문입니다. 이러한 경우는 문제없이 실행됩니다.

8.3.2 A20 게이트 활성화 방법

A20 게이트를 활성화하는 방법은 크게 세 가지가 있습니다.

키보드 컨트롤러로 활성화하는 방법

시스템 컨트롤 포트로 활성화하는 방법

BIOS 서비스로 활성화하는 방법

키보드 컨트롤러로 활성화하는 방법은 AT PC의 초창기 시절부터 사용되던 방법입니다. 과거 AT PC는 키보드 컨트롤러에 A20 게이트를 연결하여 키보드 컨트롤러를 통해 제어하도록 했습니다. 키보드 컨트롤러를 통하는 방법은 속도가 느리고 코드가 복잡하지만, PS/2 방식의 키보드/마우스를 지원하는 pc라면 어디서나 사용가능하다는 장점이 있습니다.

시스템 컨트롤 포트를 통해 활성화하는 방법은 키보드 컨트롤러의 대안으로 나왔으며, 시스템 제어에 관련된 I/O 포트를 통해 A20 게이트를 활성화하는 방법입니다. 시스템 포트를 통한 방법은 키보드 컨트롤러를 통하는 것보다 속도가 빠르고 소스 코드가 간략하다는 장점이 있습니다.

마지막 방법인 BIOS 서비스를 통해 활성화하는 방법은 486 프로세서를 지원하는 bios에서 처음 도입되었으며, bios 서비스 중에 시스템에 관련된 서비스를 통해 A20 게이트를 활성화하는 방법입니다. pc의 전반적인 정보를 관리하는 BIOS의 서비스를 사용하므로 세 가지 방법 중에서 가장 확실한 방법이라고 할 수 있습니다.

pc에 있는 프로세서의 코어의 수가 2개 이상이거나 64비트를 지원하는 프로세서라면 위의 세 가지 방법 중 최소 두 가지는 지원할 것입니다. 같은 기능을 하는 코드를 굳이 세 가지나 구현할 필요는 없으므로 8장에서 시스템 컨트롤 포트와 bios 서비스를 사용하여 A20 게이트를 활성화하겠습니다. 키보드 컨트롤러를 제어하는 예는 11장에서 자세하게 다룰 예정이니 그때 다시 살펴보겠습니다.

시스템 컨트롤 포트로 A20 게이트 활성화하기

시스템 컨트롤 포트는 I/O 포트 어드레스의 0x92에 위치하며, A20 게이트부터 하드 디스크 LED에 이르기까지 여러 가지 시스템 옵션을 담당합니다. [표 8-1]은 시스템 컨트롤 포트의 각 비트와 그 의미를 정리한 것입니다.

주변 장치 I/O를 수행하는 방식은 크게 메모리 맵 I/O 방식과 포트 맵 I/O 방식으로 나뉩니다.

메모리 맵 I/O 방식은 물리 메모리 어드레스를 I/O 용도로 할당하는 방식으로, I/O 관련 명령어가 별도로 필요하지 않고 메모리 접근 명령어(mov)로 I/O를 수행할 수 있는 것이 특징입니다. 메모리 어드레스를 I/O로 사용하므로 메모리에 접근하는 시점에 즉시 물리 메모리에 접근하여 작업을 수행해야 정상적으로 동작합니다. 만일 프로세서의 캐시가 활성화되어 있으면 캐시 정책에 따라 물리 메모리에 접근하는 시점에 차이가 있으므로 해당 메모리 어드레스는 캐시를 사용하지 않게 설정해야 합니다.

포트 맵 I/O 방식은 메모리 어드레스와 별개로 I/O 전용 어드레스를 할당하는 방식으로, 해당 포트 어드레스에 접근하려면 특수한 명령어 (in/out)가 필요합니다. 포트 맵 I/O 방식은 포트 어드레스를 별도로 할당하므로 메모리 어드레스를 보다 효율적으로 사용할 수 있는 것이 특징입니다.

x86 계열의 프로세서는 메모리 맵 I/O 방식과 포트 맵 I/O 방식을 모두 사용합니다.

[표 8-1] 시스템 컨트롤 포트의 각 비트와 의미

비트 읽기/쓰기 모드 설명

7 읽기와 쓰기 하드디스크 LED 제어

6 모두 0으로 설정하면 LED가 꺼지며, 그 외의 경우는 LED가 켜짐

5 - 사용 안함

4

3 읽기와 쓰기 부팅 패스워드 접근 제어

1로 설정하면 전원을 다시 인가할 때까지 CMOS 레지스터(0x38 ~ 0x3F)에 설정된 부팅 패스워드를 삭제하지 못하며, 0으로 설정하면 부팅 패드 워드를 삭제할 수 있음

2 읽기 전용 사용 안함

1 읽기와 쓰기 A20 게이트 제어

1로 설정하면 A20 게이트를 활성화하며, 0으로 설정하면 A20 게이트를 비활성화 함

0 쓰기 전용 빠른 시스템 리셋

1로 설정하면 시스템 리셋(리얼 모드로 전환)을 수행하며, 0으로 설정하면 아무 변화 없음

우리의 목적은 A20 게이트를 활성화하는 것이므로 시스템 컨트롤 포트의 비트 1만 1로 설정하겠습니다. 시스템 컨트롤 포트는 I/O 포트에 있으므로, 이에 접근하려면 별도의 명령어를 사용해야 합니다. 4장의 어셈블리어 목록에서 살펴보았듯이 x86 프로세서는 I/O 포트에 접근하는 in/out 명령어를 제공합니다. 다음은 in/out 명령어로 시스템 컨트롤 포트에 접근하여 A20 게이트를 활성화하는 코드입니다.

시스템 컨트롤 포트를 통해 A20 게이트를 활성화하는 코드

in al, 0x92 ; 시스템 컨트롤 포트(0x92)에서 1 바이트를 읽어 Al 레지스터에 저장

or al, 0x02 ; 읽은 값에 A20 게이트 비트(비트 1)를 1로 설정

and al, 0xFE ; 시스템 리셋 방지를 위해 0xFE와 AND 연산하여 비트0을 0으로 설정

out 0x92, al ; 시스템 컨트롤 포트(0x92)에 변경된 값을 1바이트로 설정

BIOS 서비스로 A20 게이트 활성화 방법

BIOS 서비스로 A20 게이트를 활성화하는 방법은 시스템 컨트롤 포트를 사용하는 것보다 훨씬 간단합니다. A20 게이트 관련 설정 값을 AX 레지스터에 넣고 나서 A20 게이트를 활성화하는 BIOS의 시스템 서비스(인터럽트 벡터 0x15)를 호출하는 것이 전부입니다.

BIOS 서비스 중에서 A20 게이트 관련 기능은 BIOS의 시스템 서비스에 포함되어 있으며, 시스템 서비스는 인터럽트 벡터 테이블의 0x15에 있습니다.[표 8-2]는 BIOS의 시스템 서비스 중에 A20 게이트 관련 기능을 나타낸 것입니다.

[표 8-2] BIOS 시스템 서비스(0x15) 중 A20 게이트 관련 기능 목록

AX 레지스터 값 설명 상태

0x2400 A20 게이트 비활성화 성공이면 EFLAGS 레지스터의 CF 비트 =0, AH 레지스터=0

0x2401 A20 게이트 활성화 실패이면 EFLAGS 레지스터의 CF 비트 =1 AH레지스터=0x80 or0x86

0x2402 A20 게이트 상태 읽기 성공이면 EFLAGS 레지스터의 CF 비트=0 AH 레지스터=0 AL 레지스

터=활성화(1) or 비활성화(0)

0x2403 A20 게이트 지원 유무 성공이면 EFLAGS 레지스터의 CF 비트=0 AH 레지스터=0 BX 레지스

터- 키보드 컨트롤 포트 지원(1) or 시스템 컨트롤 포트 지원(2)

실패이면 EFLAGS 레지스터의 CF 비트=1 AH 레지스터=0x80 or

0x86

8.4 A20 게이트 적용과 메모리 크기 검사

지금까지 A20 게이트를 활성화하는 방법과 코드까지 모두 살펴봤으므로 MINT64 OS에 적용하여 정상적으로 동작하는지 확인해 보겠습니다. 그리고 메모리 부족으로 발생할 수 있는 문제를 피하기 위해 pc에 설치된 메인 메모리가 최소 크기를 만족하는지 검사하는 기능을 추가한 후 이번 절을 마무리 하겠습니다.

8.4.1 A20 게이트 활성화 코드 적용

8.3절에서 시스템 컨트롤 포트와 BIOS 서비스를 사용하여 A20 게이트를 활성화하는 방법에 대해서 살펴보았습니다. 두 가지 방법 중에 어느 한 가지 방법만 사용해도 A20 게이트를 활성화할 수 있지만, BIOS마다 구현의 차이가 있으므로 두 가지를 모두 사용하겠습니다. 순서는 먼저 가장 믿을만한 BIOS 서비스를 실행하고, BIOS 서비스가 실패할 경우 시스템 컨트롤 포트를 사용하는 순으로 하겠습니다. BIOS 서비스를 사용하려면 리얼 모드여야 하므로 A20 게이트 활성화 코드는 부트 로더 또는 보호 모드 커널 엔트리 포인트에 추가해야 합니다. 어느 쪽에 추가해도 좋지만, 의미상 보호 모드 커널과 관계가 있으므로 보호 모드 엔트리 포인트에 추가하겠습니다.

이제 완성된 MINT64 OS 이미지를 플로피 디스크로 옮긴 다음 실제 PC로 부팅해 보겠습니다. 지금까지 내용이 정확히 반영되었다면 qemu와 같은 실행 결과가 표시됩니다.

보호 모드 커널에서는 최대 4GB 메모리까지 밖에 접근할 수 없으므로 4GB 이상의 메모리가 설치된 경우 이를 정확하게 판단하려면 IA-32e 모드 커널에서 검사해야 합니다. 따라서 메모리의 전체 크기를 계산하는 코드는 IA-32e 모드 커널에서 하게 하고, 이번 장에서는 MINT64 OS 실행에 필요한 메모리가 충분한지 정도만 검사하겠습니다.

8.4.2 메모리 크기 검사 기능 추가

사용 가능한 멤모리를 검사하는 가장 확실한 방법은 메모리에 특정 값을 쓰고 다시 읽어서 같은 값이 나오는지 확인하는 것입니다. 만일 해당 어드레스가 진짜 물리 메모리라면 쓴 값이 그대로 읽힙니다. 해당 어드레스가 진짜 물리 메모리가 아니라면 쓴 값은 저장되지 않았으므로 임의의 값이 읽히게 됩니다. 이런 특징을 이용하면 실제 메모리 크기가 얼마인지 쉽게 계산할 수 있습니다. 물론 메모리의 크기를 BIOS 데이터 영역에서 읽어올 수도 있지만, BIOS마다 구현에 차이가 있을 수 있으므로 직접 검사하여 측정하겠습니다.

검사하는 방법은 1MB 단위로 어드레스를 증가시키면서 각 MB의 첫 번째 4바이트에 0x12345678를 ㅆ,고 읽어 보는 것으로 하겠습니다. 물론 전체 영역을 바이트 단위로 검사할 수 있지만, pc에 설치된 메모리의 크기가 GB 단위라면 꽤 많은 시간이 걸립니다. 우리가 사용하는 PC의 메모리 크기가 MB 또는 GB 단위로 증가함을 고려할 때 MB 단위로 첫 번째 4바이트를 검사하는 것으로 충분합니다.

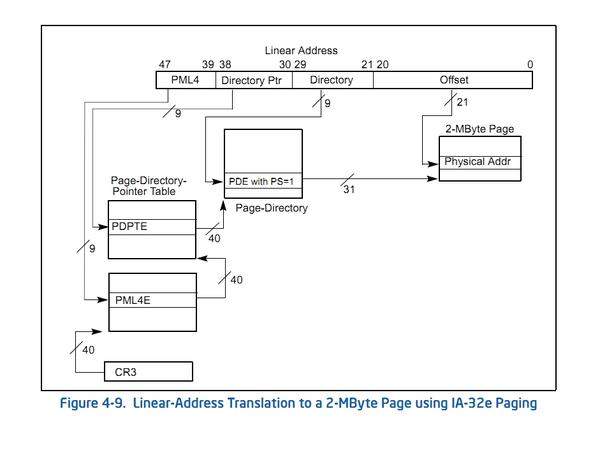
9장 페이징 기능을 활성화하여 64비트 전환을 준비하자

IA-32e 모드 커널을 위해 64GB까지 매핑하는 페이지 테이블을 생성하고, 생성된 페이지 테이블을 프로세서에 설정하여 페이징 기능을 활성화하는 방법에 대해서 살펴봅니다.

9.1 선형 주소와 4단계 페이징 기법

3장의 IA-32e 모드 페이징에 대한 3.3.3절에서 4KB 페이지를 사용하는 5단계 페이징 기법과 어드레스 변환 과정에 대해서 잠깐 설명했습니다. 2MB 페이지를 사용하는 페이징 역시 이와 크게 다르지 않으며, 차이점이라면 페이지 테이블이 사라지고 페이지 디렉터리가 직접 해당 페이지의 시작 어드레스를 가리킨다는 점입니다. [그림 9-1]은 IA-32e 모드의 4단계 페이징 기법과 어드레스 변환 과정입니다.

[그림 9-1] IA-32e 모드의 4단계 페이징과 어드레스 변환 과정



페이징에 사용하는 각 테이블은 512(2^9)개의 엔트리로 구성되며, 다음 레벨에서 사용할 테이블의 기준 주소(Base Address)를 포함합니다. 그리고 가장 마지막 레벨인 페이지 디렉터리의 엔트리는 2MB 페이지의 기준 주소를 포함합니다. 그리고 가장 마지막 레벨인 페이지 디렉터리의 엔트리는 2MB 페이지 기준 주소를 포함합니다. [그림 9-1]과 같이 선형 주소에 9비트씩 할당된 영역은 해당 테이블 내의 엔트리 오프셋을 나타내며, 이를 통해 최종 엔트리와 테이블을 찾아 실제 물리 주소로 변환합니다. 선형 주소를 실제 물리 주소로 변환하려면 CR3 레지스터에 설정된 PML4 테이블의 기준주소로부터 PML4 엔트리 -> 페이지 디렉터리 포인터 엔트리 -> 디렉터리 엔트리의 순으로 진행하여 2MB 페이지의 기준 주소를 찾은 다음 기준 주소에 선형 주소의 하위 21비트 오프셋을 더하면 구할 수 있습니다.

[그림 9-1]에서도 볼 수 있듯이 4단계 페이징 기법을 사용하려면 페이지 맵 레벨 4 테이블(PML4 Table), 페이지 디렉터리 포인터 테이블, 페이지 디렉터리 등 세 가지 자료구조를 생성해야 합니다. 각 테이블은 8바이트 크기의 엔트리로 구성되며, 각 엔트리는 다음 레벨 테이블의 기준 주소 외에 다양한 필드를 포함합니다.

[표 9-1] 페이지 테이블을 구성하는 엔트리의 각 필드와 의미

필드 설명

EXB Execute-Disable 비트의 약자로 관련된 페이지를 데이터 전용으로 설정하는 것을 의미

1로 설정하면 해당 페이지는 데이터 전용으로 설정되면 해당 영역에서 코드를 실행하면 페이지 폴트 예외가 발생함

0으로 설정하면 제약 사항 없음

IA32\_EFER 레지스터의 NXE 비트를 1로 설정할 경우 유효함

Avail Available의 약자로 OS에서 임의의 용도로 사용할 수 있는 영역을 의미

임의의 용도로 사용할 수 있음

기준 주소 다음 레벨 테이블 또는 페이지의 기준 주소를 의미

(Base Address)

A Accessed의 약자로 해당 페이지가 접근(읽기 또는 쓰기)되었음을 의미

프로세서가 페이지를 감시하여 접근이 있으면 해당 비트를 설정함

1로 설정하면 접근되었음을 나타내고, 0으로 설정되면 접근되지 않았음을 나타냄

PCD Page-level cache Disable의 약자로 해당 페이지의 캐시 활성화 여부를 의미

CR0 레지스터의 CD 비트가 1로 설정되어 전체 캐시가 비활성화된 경우, PCD 비트는 무시됨

1로 설정하면 해당 페이지의 캐시를 비활성화하며, 0으로 설정하면 해당 페이지의 캐시를 활성화함

PWT Page-Level Write\_through의 약자로 해당 페이지의 캐시 정책을 의미

PCD 비트와 마찬가지로, CR0 레지스터의 CD 비트가 1로 설정되어 전체 캐시가 비활성화된 경우 PWT 비트는 무시됨

1로 설정하면 해당 페이지는 write-through 정책이 적용되며, 0으로 설정하면 write-back 정책이 적용됨

U/S User/Supervisor의 약자로 해당 페이지의 권한을 의미

1로 설정하면 유저 레벨 권한(Ring 3)임을 나타내며, 모든 레벨에서 해당 페이지에 접근할 수 있음

0으로 설정하면 특권 레벨 권한(Ring 0~2)을 나타내며, 유저 레벨에서 해당 페이지에 접근하면 페이지 폴트 예외가 발생함

R/W Read/Write의 약자로 해당 페이지의 읽기/쓰기 정책을 의미

1로 설정하면 읽기/쓰기 모두 가능

0으로 설정하면 읽기만 가능하며, 쓰기를 시도하면 페이지 폴트 예외 발생

P Present의 약자로 해당 엔트리가 유효함을 의미

1로 설정하면 유효함을 나타내며, 0으로 설정하면 유효하지 않음을 나타냄

0으로 설정된 페이지에 접근하면 페이지 폴트 예외 발생

PAT Page Attribute Table Index의 약자로, PAT 선택에 사용되는 최상위 비트를 의미

프로세서가 PAT를 지원할 경우 PAT,PCD,PWT 3비트를 사용하여 PAT를 선택

프로세서가 PAT를 지원하지 않으면 0으로 예약됨

G Global의 약자로 CR3 레지스터의 값이 바뀌더라도 해당 페이지를 페이지 테이블 캐시인 TLB(Translation lookaside buffer)에서 교체하지 않음을 의미

CR4 레지스터의 PGE 비트를 1로 설정할 때만 유효함

1로 설정하면 CR3 레지스터 교체 시에 해당 페이지를 TLB에서 교체하지 않으며, 0으로 설정하면 T LB에서 교체함

PS Page Size의 약자로 페이지 크기를 의미

1로 설정했을 때 CR3 레지스터의 PAE 비트가 0이면 4MB 페이지를 나타내며, PAE 비트가 1이면 2MB 페이지를 나타냄

0으로 설정하면 4KB 페이지를 나타냄

D Dirty의 약자로 관련된 페이지에 쓰기가 수행되었음을 의미

A 비트와 마찬가지로 프로세서가 페이지를 감시하여 해당 비트를 설정함

1로 설정되면 쓰기가 수행되었음을 나타내고, 0으로 설정되면 쓰기가 수행되지 않았음을 나타냄

페이지 엔트리도 컨트롤 레지스터만큼이나 복잡하고 다양한 필드가 있습니다. 이렇게 복잡한 필드를 설정해야 할 때는 무작정 필드 값을 고민하기보다 우리가 사용하고자 하는 목적을 먼저 정리하고 나서 그에 맞게 필드 값을 설정하는 것이 더 좋습니다.

그럼 페이지 엔트리에 설정한 값을 고민하기 전에, 우리가 사용하고자 하는 페이징의 역할에 대해서 정리해 보겠습니다. MINT64 OS를 실행하는 데 필요한 페이지의 역할은 다음과 같습니다.

선형 주소와 물리 주소를 1:1로 매핑하여 직관적인 어드레스 변환을 수행해야함

2MB 페이지를 사용하여 최대 64GB의 물리 메모리를 매핑해야함

물리 메모리 전체 영역에 대해 캐시를 활성화하여 수행 속도 향상시켜야 함

위에서 언급한 기능 외에 다른 기능은 사용하지 않음

9.2 페이지 테이블 구성과 공간 할당

페이징 기능을 사용하려면 메모리 영역의 정보를 담고 있는 페이지 테이블을 생성하여 프로세서에 수정해야 합니다. 이번 절에서는 64GB의 메모리 공간을 관리하는 데 필요한 페이지 테이블의 크기를 알아보고 페이지 엔트리의 속성 필드를 설정하는 방법을 살펴보겠스니다.

9.2.1 64GB의 물리 메모리 관리를 위한 메모리 계산

엔트리의 필드 값을 고민하기 전에 먼저 전체적인 그림을 한 번 그려보겠습니다. 이전 절에서 MINT64 OS는 2MB 크기의 페이지를 사용하여 최대 64GB의 물리 메모리를 매핑한다고 했습니다. 그렇다면, 64GB 메모리를 관리하려면 최소한 몇 개의 테이블이 필요할까요?

우선 가장 마지막에 있는 페이지 디렉터리의 개수부터 계산해 보겠습니다. 페이지 디렉터리는 8바이트 크기인 엔트리 512(2^9)개로 구성되며, 각 엔트리는 2MB 페이지에 대한 정보를 담고 있습니다. 따라서 페이지 데릭터리 하나로 관리할 수 있는 메모리 영역은 (2MB \* 512)가 되어 1GB까지이며, 페이지 디렉터리 하나가 차지하는 메모리 크기는 8바이트 \* 512가 되어 4KB가 됩니다. 우리는 64GB 영역을 매핑해야 하므로 필요한 페이지 디렉터리의 수는 64개이고 총 256KB의 메모리가 필요합니다.

두 번째에 있는 페이지 디렉터리의 포인터 테이블 역시 페이지 디렉터리와 마찬가지로 8바이트 크기인 엔트리 512(2^9)개로 구성됩니다. 페이지 디렉터리 포인터의 각 엔트리는 하위 레벨인 페이지 디렉터리에 대한 정보를 포함하며, 앞서 계산했던 페이지 디렉터리 64개를 관리하려면 엔트리가 총 64개 필요합니다. 페이지 디렉터리 포인테 테이블 한 개로 관리할 수 있는 페이지 디렉터리의 수는 512개이므로 1개의 페이지 디렉터리 포인터 테이블로 충분하며, 페이지 디렉터리 포인터 테이블을 위해서는 총 4KB의 메모리가 필요합니다.

첫 번째에 있는 페이지 맵 레벨 4 테이블(이하 PML4 테이블)도 다른 테이블처럼 8바이트 크기인 PML4 테이블 엔트리 512개로 구성됩니다. 각 엔트리는 하위 레벨인 페이지 디렉터리 포인트 테이블의 정보를 담고 있으며, 앞서 계산했던 페이지 디렉터리 포인터 테이블 1개를 관리하려면 엔트리 1개만 있으면 됩니다. PML4 테이블 한 개는 페이지 디렉터리 포인터 테이블을 최대 512까지 관리할 수 있으므로 테이블 하나면 충분하고, PML4 테이블을 위해 필요한 메모리는 총 4KB가 됩니다.

이것으로 64GB 물리 메모리를 매핑하는 데 필요한 테이블의 개수가 총 66개(64개의 페이지 디렉터리 + 1개의 페이지 디렉터리 포인터 테이블 + 1개의 PML4 테이블)이며, 필요한 메모리는 총 264KB라는 것을 알았습니다. 지금까지 진행한 MINT64 OS의 이미지 크기가 5KB가 안 되는 것을 생각하면 264KB는 매우 큰 크기입니다. 이러한 공간을 커널 이미지에 포함시켜서 이미지 크기를 늘리면 부팅 시간을 지연시키므로 별로 바람직하지 못합니다. 또한, 페이지 테이블이 4KB로 정렬된 메모리에 있어야 한다는 제약 사항을 고려할 떄 OS 이미지에 포함시켰다가는 메모리 정렬을 위해 불필요한 공간이 낭비될 수 있습니다.

그렇다면 어디에 공간을 할당하는 것이 좋을까요?

9.2.3 페이지 테이블을 위한 공간 할당

264K나 되는 영역을 OS 이미지나 IA-32e 모드 커널 어드레스 공간(0x200000~0x600000)에 두는 것은 좋은 방법이 아니므로 별도의 공간을 할당할 필요가 있습니다. 8장의 내용 중에서 IA-32e 모드용 커널 이미지가 왜 0x200000(2MB)에서 시작하는지 궁금해 했던 분이 있을 것입니다. 이것은 0x100000(1MB)~0x200000(2MB) 영역을 IA-32e 모드용 커널 자료구조 영역으로 사용하기 때문이며, 이 공간의 가장 앞부분에 페이지 테이블이 위치합니다. 페이지 테이블의 순서는 최상위 레벨인 PML4 테이블부터 페이지 디렉터리 포인터 테이블, 페이지 디렉터리의 순이며 0x100000(1MB)~0x14200(1MB+264KB)의 어드레스에 위치합니다.

9.2.3 공통 속성 필드 설정

PML4 테이블 엔트리, 페이지 디렉터리 포인터 테이블 엔트리, 페이지 디렉터리 엔트리는 공통적인 속성 필드를 가지고 있으며, A,PCD,PWT,U/S,R/W,P 필드가 바로 그것입니다. 지금부터 해당 속성에 대해서 하나하나 살펴보겠습니다.

PCD 필드와 PWT 필드

MINT64 OS의 IA-32e 모드 커널은 보호 모드 커널과 달리 실제 OS를 구성하는 핵심역할을 합니다. 따라서 속도 향상을 위해 캐시를 사용하는 것이 좋습니다. 캐시 정책에는 write-Through 방식와 write-back 방식이 있으며, 두 가지 방법 중에 write-back 방식이 효율이 더 좋습니다. 따라서 write-back 방식의 캐시를 사용하도록 PCD 비트와 PWT 비트 값을 설정하겟습니다. 이를 위해 [표 9-1]을 참고하여 PCD 비트와 PWT 비트 값을 찾아보며 PCD=0 PWT=0이 됩니다.

U/S 필드와 R/W 필드

MINT64 OS의 최종 목표 중에 한 가지는 유저 레벨과 커널 레벨을 구분하여 잘못된 접근으로부터 보호(Protection)하는 것입니다. 하지만, 이는 유저 레벨 에플리케이션이 등장하여 유저/커널 레벨이 구분된 이후의 이야기이므로, 지금은 유저 레벨과 커널 레벨을 굳이 구분할 필요가 없스니다. 코드 영역과 데이터 영역 역시 읽기와 읽기/쓰기 속성을 따로 부여할 이유가 없으며, 속성을 따로 부여하려면 코드와 데이터를 페이지 단위(2MB)로 정렬해야 하는 제약 사항이 있습니다. 따라서 지금은 모든 페이지를 커널 레벨 영역으로 지정하고 읽기와 쓰기가 가능하게 설정하는 것으로 충분합니다. 이를 위해 [표 9-1]을 참고하여 U/S 비트와 R/W 비트 값을 찾아보면 U/S=0, R/W=1이 됩니다.

EXB 필드, A 필드. P 필드, Avail 필드

MINT64 OS에서는 페이징의 기본 기능 외에 다른 기능은 사용하지 않습니다. 따라서 EXB 필드와 같이 해당 페이지 내에서 코드 실행을 막는 기능은 사용하지 않으므로 0으로 설정합니다. 그리고 코드 실행 도중에 특정 페이지에 접근(읽기 또는 쓰기)했는지 여부도 참조하지 않으므로 임의의 값(0)으로 설정합니다. Avail 필드 역시 MINT64 OS에서 사용하지 않으므로 임의의 값(0)으로 설정하면 됩니다.

P필드는 해당 엔트리가 유효하다는 것을 나타내는 필드이므로 다른 필드와 달리 반드시 1로 설정하여 해당 엔트리에 접근했을 때 페이지 폴트 예외가 발생하지 않게 해야 합니다.

이로서 공통으로 사용되는 속성 필드의 값에 대해서 모두 알아보았습니다. 다음은 페이지 디렉터리 엔트리에서만 사용하는 속성 필드에 대해서 살펴보겠습니다.

9.2.4 페이지 디렉터리 엔트리용 속성 필드 설정

페이지 디렉터리 엔트리는 위에서 설명한 공통 필드 외에 PAT 필드와 G 필드, D 필드가 더 있습니다. 하지만, MINT64 OS에서 중요한 역할을 하지 않으니 잠시 살펴보고 다음으로 넘어가겠습니다.

PAT 필드, G 필드, D 필드

9.2.3절에서 잠시 언급했듯이, MINT64 OS는 페이지의 기본 기능 외에 다른 기능은 거의 사용하지 않습니다. 따라서 PAT처럼 페이지 별로 특수한 옵션을 지정하는 필드는 사용하지 않으므로 0으로 설정합니다.

또한 태스크 별로 페이지 매핑을 따로 구성하지 않으므로 페이지 테이블이 고정되어 있으니, 페이지 테이블 교체와 관련된 G 필드 역시 0으로 설정합니다. 마지막으로 D 필드는 A필드와 마찬가지로 MINT64 OS에서 참조하지 않으므로 임의의 값(0)으로 설정하면 됩니다.

9.3 페이지 테이블 생성과 페이징 기능 활성화

이번 절에서는 페이지 테이블을 다루는데 필요한 자료구조와 매크로를 정의하고, 이를 사용해서 페이지 테이블을 생성해 보겠습니다. 그리고 CR0 컨트롤 레지스터와 CR3 컨트롤 레지스터, CR4 컨트롤 레지스터를 제어하여 페이징 기능을 활성화해 보겠습니다.

9.3.1 페이지 엔트리를 위한 자료구조 정의와 매크로 정의

페이지 테이블은 각 엔트리의 집합이므로 페이지 엔트리를 나타내는 자료 구조를 먼저 정의해야 합니다. 세 종류의 페이지 엔트리는 내부 필드가 거의 유사하므로 개별적으로 엔트리를 정의하지 않고 한 가지 형태만 정의하여 공용으로 사용하겠습니다.

9.3.2 페이지 엔트리 생성과 페이지 테이블 생성

이제 자료구조아 매크로를 사용하여 페이지 엔트리와 페이지 테이블을 생성하는 예를 살펴보겠습니다.

9.3.3 프로세서의 페이징 기능 활성화

페이징 기능을 활성화하는 방법은 페이지 테이블을 생성하는 것보다 훨씬 간단합니다. 6장에서 사용했던 CR0 레지스터의 PG 비트와 CR3 레지스터, CR4 레지스터의 PAE 비트만 1로 설정하면 페이징 기능을 사용할 수 있습니다.

PG 비트는 CR0 레지스터의 최상위 비트에 위치하며, 프로세서의 페이징 기능을 제어하는 비트입니다. PG 비트를 1로 설정하는 순간 그 즉시 프로세서의 페이징 기능이 활성화되므로 1로 설정하기 전에 미리 CR3 레지스터에 최상위 페이지 테이블 PML4 테이블의 어드레스를 설정해야 합니다. CR3 레지스터는 페이지 디렉터리 베이스 레지스터(PDBR,Page-Directory Base Register)라고도 불리며, 최상위 페이지 테이블의 어드레스를 프로세서에 알리는 역할을 합니다. [그림 9-4]의 위는 CR3 컨트롤 레지스터의 구조를 나타낸 것입니다. 페이지 엔트리와 같은 역할을 하는 PC 비트와 PWT비트를 포함하는 것을 볼 수 있습니다.

[그림 9-4] CR3 컨트롤 레지스터의 구조(위)와 CR2 컨트롤 레지스터 구조(아래)

31(63) 12 11 9 8 6 5 4 3 2 1 0

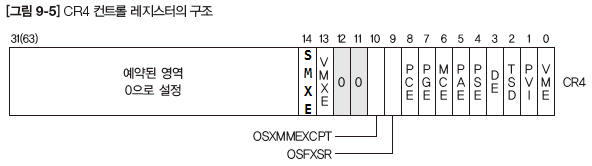
페이지 디렉터리 기준주소 예약된 영역 PCD PWT 예약된 영역 CR3(PDBR)

31(63) 0

페이지 폴트 선형 주소 CR2

사실 페이징 기능와 밀접하게 관련된 컨트롤 레지스터에는 CR3 레지스터만 있는 것이 아닙니다. 페이징 기능을 활성화한 후, 잘못된 페이지에 접근하거나 페이징 기능에 오류가 발생하면 페이지 폴트 예외(Page-Fault Exception)가 발생합니다. 이때 페이지 폴트 예외를 처리하려면 예외가 발생한 선형 주소가 필요하며, CR2 레지스터가 이러한 역할을 담당합니다.

보호 모드에서 페이지 크기가 4KB인 3단계 페이징 기능을 사용하는 것이 목적이었다면, CR0 레지스터와 CR3 레지스터를 설정하는 것으로 충분합니다. 하지만, 우리의 최종 목적은 IA-32e 모드에서 동작하며 2MB의 크기를 가지는 페이징을 활성화하는 것이므로 이를 프로세서에 알려줘야 합니다. 이러한 작업은 CR3 레지스터의 PAE(Physical Address Extensions) 비트와 페이지 디렉터리 엔트리의 PS 비트를 1로 설정함으로써 처리할 수 있습니다. 페이지 디렉터리 엔트리의 PS 비트는 이전 섹션에서 1로 설정했으므로, 이번 섹션에서는 CR4 레지스터의 PAE 비트만 1로 설정하겠씁니다.



[표 9-2] CR4 컨트롤 레지스터의 필드와 의미

필드 설명

SMXE safe mode extensions enable의 약자로 SMX 명령어를 사용할지 여부를 설정

1로 설정하면 SMX 명령어를 사용함을 나타내며, 0으로 설정하면 사용하지 않음을 나타냄

VMXE virtual machine extensions enable의 약자로 VMX 명령어를 사용할지 여부를 설정

1로 설정하면 VMX 명령어를 사용함을 나타내며, 0으로 설정하면 사용하지 않음을 나타냄

OSXMMEXCPT Operating System Suport for Unmasked SIME Floating-Point exceptions를 의미하며, SIME 관련 실수 연산 시 마스크 되지 않은 예외가 발생했을 때 예외 처리 방법을 설정

1로 설정하면 예외가 SIME Floating-Point Exception로 발생하면, 0으로 설정하면 예외가 Invalid Opcode Exception으로 발생함

실수 연산을 사용하는 경우 정확한 예외 처리를 위해 1로 설정하는 것을 권장

OSFXSR operating system support for FXSAVE and FXRSTOR instruction을 의미하며, os가 FXSAVE/FXRSTOR 명령 및 실수 연산 관련 명령을 지원하는지 여부를 설정

1로 설정하면 실수 연산 관련 명령을 지원함을 나타내며, 0으로 설정하면 실수 연산 관련 명령을 지원하지 않음을 나타냄

0으로 설정하면 실수 연산할 때마다 invalid Opcode Exception이 발생하므로 1로 설정하는 것을 권장

PCE Performance\_monitoring counter enable의 약자로 RDPMC 명령어를 사용할 수 있는 권한 레벨을 설정

1로 설정하면 모든 레벨에서 사용 가능함을 나타내며, 0으로 설정하면 최상위 레벨(0)에서만 사용 가능함을 나타냄

PGE page global enable의 약자로 global page feature를 사용할지 여부를 설정

1로 설정하면 global page featur를 사용함을 나타내며, CR3 레지스터가 교체되어 페이지 테이블이 바뀌는 경우 페이지 엔트리의 PG 비트가 1로 설정된 페이지는 TLB에서 교체안됨

0으로 설정하면 global page feature 기능을 사용하지 않음을 나타냄

MCE machine-check enable의 약자로 machine-check 예외를 사용할지 여부를 설정

1로 설정하면 machine-check 예외를 사용함을 나타내며, 0으로 설정하면 사용하지 않음을 나타냄

PAE physical address extensions의 약자로 36비트 이상의 물리 메모리를 사용할지 여부를 설정

1로 설정하면 36비트 이상의 물리 메모리를 사용함을 나타내며, 0으로 설정하면 사용하지 않음을 나타냄

IA-32e 모드에서는 필수적으로 1로 설정해야함

PSE Page size extensions의 약자로 4KB 또는 그 이상의 페이지 크기를 사용할지 여부를 설정

1로 설정할 경우 2MB 또는 4MB 페이지를 사용함을 나타내며, 0으로 설정할 경우 4KB 페이지를 사용함을 나타냄

PAE가 1로 설정될 경우 PSE 비트는 무시되며 페이지 디렉터리 엔트리의 PS 비트에 의해 페이지 크기가 결정됨

DE debugging extensions의 약자로, DR4와 DR5 레지스터에 접근을 허락할지 여부를 설정

1로 설정하면 DR4, DR5 레지스터는 프로세서에 의해 예약(reserved)되며, 해당 레지스터에 접근할 경우 undefined opcode exception이 발생

0으로 설정하면 DR4,DR5 레지스터는 각각 DR6,DR7 레지스터의 다른 이름(alias) 역할을 함

TSD Time stamp disable의 약자로 RDTSC 명령어를 사용할 수 있는 권한 레벨을 설정

1로 설정하면 최상위 레벨(0)에서만 사용 가능함을 나타내며, 0으로 설정함녀 모든 레벨에서 사용가능함을 나타냄

PVI Protected-mode virtual interrupts의 약자로 virtual interrupt flag를 사용할지 여부를 설정

1로 설정하면 VIF를 사용함을 나타내고, 0으로 설정하면 VIF를 사용하지 않음을 나타냄

VME virtual 8086 mode extensions의 약자로 가상 8086 모드에서 interrupt and exception-handling extensions 사용 여부를 설정

1로 설정하면 interrupt and exception-handling extensions을 사용함을 나타내고, 0으로 설정하면 사용하지 않음을 나타냄

기존의 바이트를 스택에 push하고 1바이트 단위로 처리한다음 끝나는 부분에 원래의 바이트 단위를 pop해주는 코드이다. 보통은 이렇게 사용하면 되겠다.

아래와 같은 코드를 사용하면 구조체 크기는 1바이트가된다.

#pragma pack(push, 1)

typedef struct DATA{

char cData;

}DATA;

#pragma pack(pop)

10장 64비트 모드로 전환하자

IA-32e 모드 커널을 작성하고 페이징 기능과 IA-32e 기능을 활성화하여 대망의 64비트 모드로 전환합니다.

9장에서는 IA-32e 모드의 필수 요소인 페이징 기능을 활성화하는 방법에 대해서 알아보았습니다. 10장에서는 준비된 페이지 테이블을 바탕으로 활성화하고 IA-32e 모드로 전환하여, 우리의 최종 목표인 64비트 모드로 전환해 보겠습니다.

10장은 다른 장보다 많은 페이지를 할애했습니다. 64비트로 전환하는 마지막 단계인 만큼 이것저것 마무리할 것이 많기 떄문입니다. 하지만, IA-32e 모드로 전환하는 데 핵심적인 부분ㅁ나 추리면 그렇제 많지 않으니, 그외에 소스 통합과 관련된 내용은 가볍게 읽어 내려가면 됩니다.

본격적인 내용에 들어가기에 앞서 IA-32e 모드로 전환하려면 거쳐야 하는 작업들의 순서에 대해 알아보겠습니다. 보호 모드에서 IA-32e 모드로 전환하려면 [그림 10-1]과 같은 순서의 7단계를 거쳐야합니다.

[그림 10-1] IA-32e 모드로 전환하는 7단계

세그먼트 디스크립터 추가

(IA-32e 모드 코드와 데이터용 세그먼트 디스크립터 추가)

↓

CR4 컨트롤 레지스터 설정

(CR4 컨트롤 레지스터 PAE 비트=1)

↓

CR3 컨트롤 레지스터 설정

(CR3 컨트롤 레지스터에 PML4 테이블 어드레스 설정) 페이지 활성화 및 모드 전환

↓

IA32\_EFER 레지스터 설정

(IA32\_EFER 레지스터(MSR 레지스터)의 LME 비트=1)

↓

CR0 컨트롤 레지스터 설정

(CR0 컨트롤 레지스터의 PG 비트=1)

↓

jmp 명령으로 CS 세그먼트 셀렉터 변경 및 IA-32e 모드로 전환

(jmp 0x18:IA-32e 모드 커널의 시작 어드레스)

↓ 32비트 보호 모드

---------------------------------------------------------------------------------------------------------------------------------------

각종 세그먼트 셀렉터와 스택 초기화 64비트 IA-32e 모드

(DS,ES,FS,GS,SS 세그먼트 셀렉터와RSP, RBP 레지스터 초기화)

↓

IA-32e 모드 커널 실행

[그림 10-1]을 얼핏 보면 굉장히 복잡한 과정을 수행해야 하는 것처럼 보이지만, 사실은 그렇지 않습ㄴ다. 위의 과정을 다시 디스크립터와 셀렉터를 갱신하는 부분과 페이징을 활성화하는 부분 그리고 IA-32e 모드로 전환하는 부분의 세 가지로 나눌 수 있습니다.

그럼 먼저, IA-32e 모드로 전환하는 방법을 알아보기 전에, 프로세서가 IA-32e 모드를 지원하는지부터 확인해 보겠습니다. i386 이상의 프로세서라면 기본으로 보호 모드를 지원하므로, 현재 동자하는 거의 모든 프로세서는 보호 모드를 지원한다고 봐도 무방하지만, IA-32e 모드는 조금 사정이 다릅니다. IA-32e 모드는 근래에 추가된 프로세서 확장 기능(Processor Extended Feature)으로 모든 프로세서에서 지원하다고 보기 어렵습니다. 프로세서가 IA-32e 모드를 지원하는지 여부를 확인하지 않는다면, 모드 전환 과정에서 문제가 발생했을 때 의미 없는 디버깅에 시간을 낭비하게 될지도 모릅니다. 이 같은 상황을 미연에 방지하려면 가장 먼저 IA-32e 모드 지원 여부를 검사해야 합니다.

10.1 프로세서의 제조사와 IA-32e 지원 여부 검사

프로세서는 발전을 거듭하면서 수많은 기능이 추가되었습니다. IA-32e 모드도 그중에 하나로 최근에 추가된 기능 중에 하나입니다. 프로세서마다 지원하는 기능이 조금씩 다르기 때문에 프로세서 제조사는 지원하는 기능을 확인할 수 있는 방법을 제공하며, 이것이 바로 CPUID 명령어입니다. 이번절에서는 CPUID 명령어를 이용하여 프로세서 제조사를 확인하는 방법과 IA-32e 모드를 지원하는지 확인하는 방법을 살펴보겠습니다.

10.1.1 CPUID를 사용하여 프로세서 정보 확인 방법

x86\_64 계열에는 프로세서 종류가 무수히 만흡니다. 프로세서를 생산하는 주 업체인 인텔과 AMD에서는 매년 새로운 프로세서를 선보이며, 각 프로세서는 강력한 속도와 기능을 무기로 PC 업그레이드 욕구를 자극합니다. 그런데 새로운 프로세서가 나올 때마다 바빠지는 사람이 마케팅 담당자나 소비자만은 아닙니다. OS 개발자 역시 프로세서의 새로운 기능을 지원하려면 발 빠르게 움직여야 하기 때문입니다. 그렇다면 OS는 어떠한 방법을 통해 프로세서가 지원하는 기능을 확인할까요?

x86 계열 프로세서는 프로세서에 대한 정보를 확인할 수 있는 CPUID 명령어를 제공하고 있습니다. CPUID 명령어는 CPU Identification의 약자로 프로세서어 대한 다양한 정보를 제공합니다. CPUID는 EAX 레지스터에 설정된 값에 따라 해당 정보를 조회하며, 범용 레지스터 EAX,EBX,ECX,EDX를 통해 그 결과를 넘겨줍니다. CPUID는 크게 기본 CPUID 정보(basic CPUID information)와 확장 기능 CPUID 정보(extended feature CPUID information)를 제공합니다. 두 정보는 다시 여러 가지 세부 정보로 나누어집니다. 우리의 목적은 프로세서가 64비트 모드를 지원하는지 확인하는 것이므로 IA-32e 모드 부분을 위주로 살펴보겠습니다.

[표 10-1] 제조사와 64비트 지원 기능에 관련된 CPUID 기능 목록

EAX 값 실행결과와 설명

0x00000000 기본 CPUID 정보 조회

EAX: 해당 프로세서가 지원하는 기본 CPUID 정보의 입력 최대값(EAX)

EBX,EDX,ECX의 순서, 하위 바이트에서 상위 바이트의 순서로 12바이트 제조사 이름이 저장됨

인텔은 GenuineIntel로 표시되며, AMD는 AuthenticAMD로 표시됨

0x80000001 확장 기능 CPUID 정보 조회

EAX : 제조사마다 차이가 있음

EBX : 제조사마다 차이가 있음

ECX : 비트 0 - 64비트 모드에서 LAHF/SAHF 명령 지원 여부 그 외 나머지 비트 – 제조사마다 차이가 있음

EDX : 비트 11- 64비트 모드에서 SYSCALL/SYSRET 명령 지원 여부

비트 20 – Execute Disable 비트 지원 여부

비트 29 - 64비트 모드 지원 여부

그 외 나머지 비트 - 제조사마다 차이가 있음

CPUID 명령어를 실행했을 때, 반환되는 정보는 프로세서가 생상된 제조사(인텔,AMD 등)에 따라 다릅니다. 하지만, 여러 프로세서 간의 호환을 위해 공통적인 필드를 가지고 있으며, [표 10-1]은 프로세서 대표 제조사인 인텔과 AMD의 공통 필드를 위주로 정리했습니다. 제조사별 세부 정보가 궁굼한 분은 제조사의 홈페이지나 이 책의 부록 CD에 담긴 CPUID 관련 문서를 참고하기 바랍니다.

10.1.2 프로세서 제조사와 IA-32e 모드 지원 여부 확인

CPUID는 EAX, EBX, ECX, EDX 레지스터로 값을 전달하므로 CPUID 명령어를 사용하려면 어셈블리어 코드로 직접 레지스터를 제어해야 합니다. 따라서 이전 장에서 보호 모드 엔트리 포인트를 위해 작성했던 PRINTMESSAGE 함수의 형태를 응용하여 C 언어에서 호출할 수 있는 CPUID 어셈블리어 함수를 만들어 보겠습니다.

CPUID 명령어는 EAX 레지스터에 조회할 정보를 넘겨받고, 조회된 정보를 EAX, EBX, ECX, EDX 레지스터에 담아 넘겨줍니다. 만일 EAX 레지스터 값을 고정하여 특정 정보만 조회하는 함수를 만든다면 범용으로 사용할 수 없으므로 조회하는 목록의 수만ㅋ므 어셈블리어 함수를 작성해야 합니다. 이는 상당히 비효율적이므로 우리는 5개의 파라미터로 구성된 kReadCPUID() 함수를 작성하고, C 코드에서 이를 호출하여 부가 작업을 처리하게 하겠습니다.

사실 어셈블리어 코드로 함수를 작성한다고 해서 C 코드에서 이를 호출할 수 있는 것은 아닙니다. 이를 위해서는 어셈블리어 코드에서 해당 함수가 외부에서 사용된다는 것을 알려야 하며, 이러한 역할을 하는 것이 global 지시어입니다. 어셈블리어 파일 상단에 ;global kReadCPUID’와 같이 추가하면, 링크 단계에서 kReadCPUID() 함수를 호출하는 부분과 어셈블리어 함수를 서로 연결해 줍니다.

다음은 DWORD 타입의 EAX 값과 EAX,EBX,ECX,EDX의 어드레스를 넘겨받아 CPUID 명령어를 실행하는 어셈블리어 코드와 그 함수의 선언부를 나타낸 것입니다. 주의해서 볼 부분은 스택에 들어 있는 두 번째 파라미터부터 다섯 번째 파라미터까지는 ESI 레지스터에 값을 옮긴 후, 다시 ESI 레지스터가 가리키는 어드레스에 값을 저장한다는 것입니다. 이는 kReadCPUID()의 파라미터가 변수의 포인터이기 때문입니다. 따라서 스택에 있는 값은 실제 변수의 어드레스이므로, ESI 레지스터에 어드레스를 옮긴 후 다시 ESI 레지스터가 가리키는 어드레스에 값을 저장하는 것입니다. 이는 C언어의 \*pdwEAX = eax 와 같은 역할을 합니다. 나머지 코드는 5장과 6장에서 작성했던 PRINTSTRING 함수, PRINTMESSAGE 함수와 같으므로 해당 함수를 참고하면 이해하는데 도움이 될 것입니다.

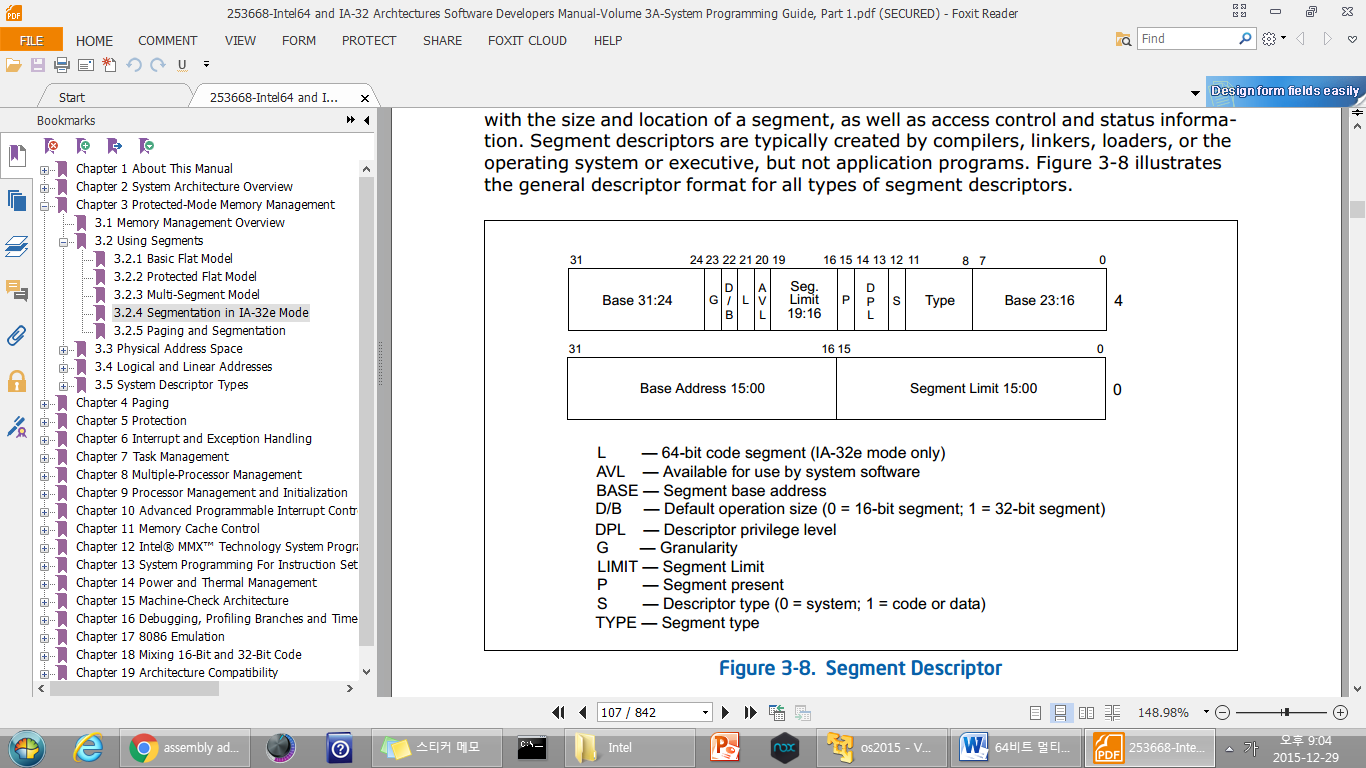
10.2 IA-32e 모드용 세그먼트 디스크립터 추가

리얼 모드에서 보호 모드로 전환하려면 보호 모드용 세그먼트 디스크립터가 필요했듯이, 보호 모드에서 IA-32e 모드로 전환하기 위해서는 IA-32e 모드용 세그먼트 디스크립터가 필요합니다. IA-32e 모드 커널을 위해 1MB 이상의 영역에 IA-32e 모드용 GDT를 새로 생성하는 것도 좋은 방법이지만, 이미 보호 모드 커널 엔트리 포인트에 GDT를 생성했으므로 간단히 여기에 추가하겠습니다. (1MB 이상의 영역에 GDT를 새로 생성하는 작업은 IA-32e 모드 전환이 완료된 다음에 할 것입니다.)

10.2.1 보호 모드 커널 엔트리 포인트에 디스크립터 추가

IA-32e 모드 커널용 코드 디스크립터와 데이터 디스크립터를 생성하는 방법은 보호 모드와 거의 유사합니다. 왜냐하면 IA-32e 모드의 세그먼트 디스크립터는 보호 모드의 세그먼트 디스크립터와 거의 같은 크기와 필드로 되어 있기 때문입니다. 차이점이라면 IA-32e 모드에서는 디스크립터의 기준 주소와 세그먼트 크기 값에 상관없이 64GB 전체 영역으로 설정된다는 것과 세그먼트 디스크립터의 L 비트가 IA-32e 서브 모드 중 호환 모드 또는 64비트 모드를 선택하는 데 사용된다는 것입니다.

[그림 10-2] IA-32e 모드용 세그먼트 디스크립터의 구조



우리는 IA-32e 모드의 서브 모드 중에서 64비트 모드를 사용할 것이므로, 보호 모드용 세그먼트 디스크립터를 기반으로 L 비트를 1로 D 비트를 0으로 설정하면 됩니다. D 비트를 0으로 설정하는 이유는 L 비트와 D 비트가 모두 1인 경우를 다른 목적으로 예약해두었기 때문입니다. QEMU에서는 이를 확인하지 않았지만, 실제 PC로 테스트할 때 L 비트와 D 비트를 모두 1로 설정하면 리부팅될 수 있으니 주의해야 합니다. 다음 코드는 보호 모드 엔트리 포인트 코드에 IA-32e 모드 커널용 코드 세그먼트 디스크립터와 데이터 세그먼트 디스크립터를 추가한 것입니다. IA-32e 모드로 전환한 후에는 보호 모드용 세그먼트 디스크립터를 사용하지 않으므로 IA-32e 모드용 세그먼트 디스크립터를 앞족으로 배치했습니다.

10.3 IA-32e 모드 전환과 1차 정리

지금까지 IA-32e 모드를 활성화하는데 필요한 페이징 기능과 세그먼트 디스크립터에 대해서 알아봤습니다. 이제 실제 코드를 작성하여 IA-32e 모드로 전환해보고 IA-32e 모드 커널로 들어가기 전에 코드르 정리하는 시간을 갖겠습니다.

10.3.1 물리 메모리 확장 기능 활성화와 페이지 테이블 설정

물리 메모리 확장 기능을 활성화하고 프로세서에 페이지 테이블을 설정하는 과정은 이미 9장에서 다뤘습니다. 따라서 이에 대한 자세한 내용은 9장의 9.3절을 참고하고, 이번 절에서는 복습하는 차원에서 실제 코드 위주로 간단히 살펴보겠습니다.

물리 메모리 확장(PAE,Physical Address Extensions) 기능은 CR4 레지스터의 PAE 비트(비트5)가 담당하고 있으며, PAE 비트를 1로 설정해서 물리 메모리 확장 기능을 사용할 수 있습니다. 프로세서에 페이지 테이블을 설정하려면 CR3 레지스터에 PML4 테이블의 어드레스를 저장하면 됩니다.

10.3.2 64비트 모드 활성화와 페이징 활성화

IA-32e 모드를 활성화하는 실질적인 최종 관문은 IA32\_EFER 레지스터의 LME 비트를 1로 설정하는 작업입니다. IA-32e\_EFER 레지스터의 비트를 활성화하지 않으면 IA-32e 모드용 세그먼트 레지스터로 교체한다 해도 32비트 보호 모드로 동작할 뿐입니다. IA32-EFER 레지스터는 이름에서도 알 수 있듯이 범용 또는 컨트롤 레지스터가 아니며, MSR(Model-Specific Register)라고도 불리는 특수한 용도의 레지스터입니다.

MSR 레지스터는 이름 그대로 프로세서 모델에 따라 특수하게 정의된 레지스터로 크기는 64비트이며 각 모델에 따라 차이가 있습니다. MSR 레지스터의 종류에는 크게 6가지가 있고 다음과 같습니다.

1. 디버깅 및 성능 측정 (Debugging and performance monitoring)

2. 하드웨어 에러 검사(Machine-check)

3. 메모리 범위와 메모리 타입 설정(memory type range registers,MTRRs)

4. 온도와 전력 관리(Thermal and power management)

5. 특수 명령어 지원(instruction-specific support)

6. 프로세서 특징과 모드 지원(processor feature/mode support)

IA-32e 모드 전환에 사용할 IA32\_EFER(extended feature enable register) 레지스터는 프로세서 특성과 모드 지원에 속하는 MSR 레지스터로 프로세서의 확장 기능을 제어할 수 있는 레지스터입니다. IA32\_EFER 레지스터로 제어할 수 있는 항목에는 SYSCALL/SYSRET 사용, IA-32e 모드 사용, Execute Disable(EXB) 사용 등이 있으며, 제어 기능 외에 현재 운영 중인 모드가 IA-32e 모드인지 확인하는 기능도 포함하고 있습니다.

MSR 레지스터는 개별적으로 할당된 레지스터 어드레스가 있고 해당 어드레스를 통해서만 접근할 수 있습니다. MSR 레지스터는 다른 레지스터와 달리 데이터 이동 명령(mov)으로 접근할 수 없으며, MSR 레지스터용 명령어인 RDMSR(read from model specific register)과 WRMSR(write to model specific register)을 사용해야 합니다. RDMSR 명령어는 MSR 레지스터에서 값을 읽어오는 역할을 하며, ECX,EDX 그리고 EAX 레지스터를 사용하니다. ECX 레지스터에 읽을 MSR 레지스터 어드레스르 넘겨주면 MSR 레지스터의 상위 32비트는 EDX 레지스터에 하위 32비트는 EAX 레지스터에 저장해줍니다. WRMSR 명령어는 MSR 레지스터에 값을 쓰는 역할을 하며, 역시 RDMSR 명령어와 같은 레지스터를 사용합니다. ECX 레지스터에 쓸 MSR 레지스터 어드레스를 넘겨주면, 상위 32비트는 EDX 레지스터의 값으로 하위 32비트는 EAX 레지스터의 값으로 MSR 레지스터에 저장해줍니다.

[표 10-2] IA32-EFER 레지스터의 비트 구성

비트 필드 읽기/쓰기 모드 설명

64~12 - - 제조사마다 차이가 있음

11 NXE 읽기,쓰기 No-execute enable의 약자로 execute disable 비트를 사용할지 여부를 표시

1로 설정하면 페이지 엔트리의 EXB 비트 값에 따라 실행 불가 기능이 활성화/비활성화 됟ㅁ

0으로 설정하면 페이지 엔트리의 EXB 비트가 무시됨

10 LMA 읽기 전용 long mode active의 약자로 현재 동작 중인 모드가 IA-32e 모드인지 여부를

표시

1로 설정하면 프로세서 모드가 IA-32e 모드(호환모드 또는 64비트 모드)임을 나타내며, 0으로 설정하면 프로세서 모드가 기타 모드임을 나타냄

9 - 예약됨 예약된 영역

8 LME 읽기,쓰기 long mode enable의 약자로 IA-32e 모드를 활성화함을 의미

1로 설정하면 프로세서의 IA-32e 모드를 활성화함을 나타내며, 0으로 설정하면 비활성화함을 나타냄

7~1 - 예약됨 예약된 영역

0 SCE 읽기,쓰기 system call enable의 약자로 syscall,sysret 명령어를 사용할지 여부를 의미

1로 설정하면 syscall,sysret 명령어를 사용함을 나타내며, 0으로 설정하면 사용하지 않음을 나타냄

MSR 레지스터는 CPUID와 마찬가지로 프로세서가 생상된 제조사(인텔,AMD 등)에 따라 차이가 있습니다. 그래서 [표 10-2]도 프로세서의 양대 제조사인 인텔과 AMD가 공통으로 지원하는 필드에 대해서만 정리했습니다. 제조사별 세부적인 정보가 궁금한 분은 제조사의 홈페이지나 부록 CD에 포함된 아키텍처 관련 문서를 참고하기 바랍니다.

10.3.3 IA-32e 모드로 전환과 세그먼트 셀렉터 초기화

IA-32e 모드로 전환하는 마지막 작업은 CR0 레지스터를 변경하여 캐시와 페이징을 활성화하고서 세그먼트 셀렉터를 IA-32e 커널용으로 교체하는 것입니다.

9장에서 초기화한 페이지 테이블의 PCD 비트와 PWT 비트는 페이징을 활성화했을 때만 유효합니다. x86 계열의 프로세서에는 페이지의 캐시 설정보다 우선하는 캐시 관련 비트가 있으며, 그것은 바로 CR0 컨트롤 레지스터의 NW비트(비트 29)와 CD 비트(비트 30)입니다. NW비트와 CD비트는 각각 Not write-through와 cache disable의 약자로 두 비트를 조합하여 여러가지 캐시 mode를 설정할 수 있습니다. 두 비트 모두 0으로 설정하면 normal cache mode로 동작합니다.