CaOS

(Calcium Operating System)

프로세서 초기화

2007.9.5 v0.05 문서 작성

김기오

www.asmlove.co.kr

프로세서 초기화

bootsect.asm와 setup.asm 파일이 프로세서를 초기화하고 커널이 동작하기 위한 준비를 한다.

1. bootsect.asm

bootsect.asm 파일의 코드는 플로피 디스크의 첫번째 섹터에 저장되어 있으므로 바이오스가 스스로 플로피 디스크에서 메모리로 읽어주고 실행을 시킨다. 따라서 bootsect.asm 부터가 커널 개발의 시작이 되고 커널 이미지를 메모리로 읽어오는 기능을 구현해야 한다. 그런데 커널이 시작되기 위해서 미리 프로세서의 몇가지 기능을 동작시켜야 하는데 이런 기능을 모두 bootsect.asm에 넣으면 512바이트를 넘길 수 있으므로 커널과 부트로더의 중간 단계로 setup.asm을 실행한다.
결국 bootsect.asm은 setup.asm과 커널 이미지를 메모리로 읽어오는 일만 하고 setup.asm에서 프로세서의 초기 설정을 담당한다.

```
[org 0x0]

SETUP_SECT      equ 4
KERNEL_SECT     equ KERNEL_SIZE

BOOT_SEG        equ 0x7c0
SETUP_SEG       equ 0x900

; Kernel is copied to 0xA000 at first,
; and to 0x100000 in setup.asm
KERNEL_SEG      equ 0xA00

SETUP_ADDR      equ 0x9000


        jmp BOOT_SEG:start
```

org 0x0은 부트로더가 메모리에 저장되는 위치를 0번지로 지정한다. 그리고 jmp BOOT_SEG:start 명령은 세그먼트 주소를 0x7c0으로 바꾸고 오프셋 주소를 mov ax, cs 명령으로 바꿔서 계속 실행되도록한다. 바이오스가 자동으로 플로피디스크의 부트로더를 읽어서 물리 메모리 0x7c00 번지에 저장해주는데 왜 org 0x0을 해야 할까? 변수의 주소 계산에 세그먼트 주소와 오프셋이 같이 사용되기 때문에 어셈블러에게 프로그램의 시작 주소를 미리 알려줄 필요가 있다.

부트로더에서 메모리 0x7c0a 번지에 있는 변수를 읽고자 할 때를 생각해보자. 예를 들어 세그먼트 주소가 만약 0x0이라면 오프셋이 0x7c0a가 되야 한다. 마찬가지로 세그먼트 주소가 0x7c0 이라면 오프셋은 0xa가 되야 한다. 세그먼트 주소가 얼마인지에 따라 오프셋 계산이 틀려지는 것이다.

그럼 실제 PC에서 부트로더가 실행될 때 세그먼트 주소는 얼마가 될까? 이건 바이오스가 어떻게 부트로더를 실행시키느냐에 따라 다르다. 실제로 확인해 본 결과 몇몇 PC에서는 세그먼트 주소가 0x0이 되는 것을 알 수 있었지만 이것은 어떤 표준이나 약속이 있는 것이 아니다. 따라서 부트로더가 시작되면서 바로 세그먼트 주소를 명확하게 정의할 필요가 있게된다.

그래서 부트로더에서 세그먼트 주소를 0x7c0으로 맞추기 위해 jmp 명령을 사용하는 것이다. 그러면 이렇게 세그먼트 주소를 0x7c0으로 맞추면 오프셋 주소는 어떻게 계산해야 할까. 잠깐 생각하면 부트로더의 실제 시작 주소가 0x7c00 이므로 오프셋 주소를 0x7c00을 기준으로 계산해야하지 않을까 생각되지만 세그먼트 주소가 이미 0x7c0으로 정의되었으므로 오프셋은 0x7c0을 감안해서 계산되어야 한다.

0x7c0a에 있는 변수를 접근하기 위해 세그먼트 주소를 0x7c0으로 정했다면 오프셋은 0xa가 되야 한다. 부트로더가 시작되는 주소를 0x7c00이라고 어셈블러에게 정해주면 어셈블러는 오프셋을 계산할 때 항상 0x7c00에 변수의 주소를 더해서 계산하게 된다. 결국 모든 변수의 주소가 0x7c00+X 가 된다. 즉 세그먼트 주소에 상관없이 오프셋이 계산되므로 만약 세그먼트 주소가 0x7c0 이라면 물리 주소는 0x7c00 + 0x7c00 + X 가 되서 엉뚱한 주소로 계산된다.

그래서 오프셋 계산에서 0x7c00을 두번 더하지 않기 위해 org 0 으로 오프셋의 시작 지점을 설정하고 세그먼트 주소에 0x7c0 을 정하는 것이다.

그럼 setup.asm 파일도 열어서 확인해보자. setup.asm 파일의 경우에는 org 0x9000 으로 설정하고 있다. 그 이유는 bootsect.asm 의 끝 부분에서 setup.asm 파일로 넘어갈 때 jmp 0x0:0x9000을 실행하기 때문이다. 즉 setup.asm 이 실행되는 순간에는 세그먼트 주소가 0이므로 오프셋 주소가 곧 물리 주소로 그대로 변환된다. 따라서 어셈블러가 주소 계산을 할 때 0x9000을 기준으로 오프셋을 계산하도록 해주어야 한다.

왜 bootsect.asm과 setup.asm의 org 를 다르게 해야 할까. 여기 작성한 프로그램의 경우에는 lgdt [gdtr] 명령어 때문이다. lgdt 명령어는 특정 물리 메모리 주소에서 6바이트의 값을 읽어서 gdtr 레지스터에 저장하는 명령어이다. 근데 이 명령어는 오퍼랜드를 지정할 때 변수의 오프셋 주소만 사용되고 세그먼트 주소는 사용하지 않는다. lgdt [gdtr]은 되는데 lgdt [ds:gdtr]은 안된다는 것이다. 따라서 세그먼트 주소와 오프셋 주소를 더해서 물리 메모리 주소를 나타내는 주소 계산법이 소용이 없다. 오프셋 주소만으로 gdtr이라는 변수의 주소가 결정되야 한다. 그래서 setup.asm에서는 세그먼트 주소를 0x0으로 설정해서 사용하는 것이다. org 0x9000 으로 지정했으므로 어셈블러는 gdtr 변수의 주소를 계산할 때 프로그램의 시작 지점에서 변수가 어느정도 떨어져있나를 계산

하고 거기에 0x9000을 더해서 0x9000+X 로 오프셋을 결정하게 된다. 그리고 lgdt [gdtr] 명령을 lgdt 0x900X로 변환하게 된다.

bootsect.asm에서도 세그먼트 주소를 0x0으로 사용해도 된다. 어떻게든지 부트로더나 setup.asm이 시작되는 물리 주소를 생각하고 실제 어셈블 단계에서 어떻게 주소 계산이 될지를 잘 따져보아야 한다.

```
boot_drv                db 0      ; floppy drive A:
sect_per_track    dw 18   ; sectors per one track
head_per_drv      dw 2    ; floppy has 2 heads
current_sect      dw 1    ; current reading sector # (relative)
read_sect               db 0      ; it read how many sectors
```

플로피 드라이브의 하드웨어 스펙이다. 보통 플로피 드라이브는 A: 이고 번호는 0번이다. 플로피 디스크의 하나의 트랙에는 18개의 섹터가 있고 2개의 헤드를 가진다. 여기서 currentSect 변수는 각 트랙 내부의 섹터 번호가 아닌 전체 플로피 디스크를 놓고 봤을 때 섹터 번호이다. 즉 플로피 디스크의 모든 섹터를 일렬로 나열했을 때 섹터 번호이므로 상대적인 섹터 번호라고 생각하면 된다.

왜 이렇게 구성했냐면 만약 커널이 커져서 여러 개의 트랙에 저장될 경우 몇번 트랙의 몇번 섹터인지 구별해서 커널을 읽어야 하기 때문이다. 나중을 위해서 이렇게 상대적인 섹터 번호를 가지고 커널 위치를 파악하고, 상대 섹터 번호를 트랙과 헤드 번호로 변환하는 함수를 하나 만들어서 사용하게 된다.

```
start:
        mov ax, cs
        mov ds, ax              ; ds=es=0x7c0
        mov es, ax

        mov ss, ax              ; stack: 0x7c0:0 -> low memory
        mov sp, 0x0

        mov [boot_drv], dl      ; bios store drive number in dl
```

바이오스가 실행을 끝내고 부트로더로 넘어갈 때 바이오스는 dl 레지스터에 부트로더가 저장된 드라이브의 번호를 저장해놓는다. 크게 필요한 것은 아니지만 나중에 혹시 하드디스크에서 부팅할 경우를 위해 저장해놓았다.

```
mov ax, SETUP_SEG        ; read 512bytes of setup.asm
mov es, ax
mov si, 0                        ; store image at es:si
mov cx, SETUP_SECT       ; how many sectors?
call read_sectors

mov ax, KERNEL_SEG
mov es, ax
mov si, 0
mov cx, KERNEL_SECT
call read_sectors
```

물리 메모리 0x9000 에 setup.asm을 저장한다. 하나의 섹터를 읽는 함수가 read_one_sect 인데 이 함수를 이용해서 여러 개의 섹터를 읽는 read_sect 함수를 만들었다. 커널이 몇 섹터 크기인지를 나타내는 KERNEL_SEG 상수는 Makefile에서 setup.asm을 어셈블할 때 계산된다.

```
call a20_try_loop ; activate A20 line
```

프로세서의 주소 버스 중 A20 핀을 활성화시킨다. 그래야 1MB 이상의 메모리에 접근할 수 있다. 1MB는 16진수로 0x100000 이고 첫번째 자리가 20번 비트이기 때문이다.

```
mov ax, 0xb800
mov es, ax
mov bx, 0x0
mov al, byte [hello_msg]   ; Greeting!
mov byte [es:bx], al
```

부트로더가 시작되면 C 라고 화면에 출력한다. 부트로더가 정상적으로 시작되었는지 확인할 수 있다. 왜 그냥 아스키 코드 'C'를 출력하지 않고 변수를 이용했냐면, 주소 지정이 제대로 되었는지 확인하기 위해서이다. 그냥 비디오 메모리에 바로 'C'를 쓰면 변수들의 접근이 제대로 되고 있는지 확인할 수가 없다.

```
mov ax, KERNEL_SECT

;=========================================
```

```
        ; segment addres = 0x0
        ; offset = 0x9000
        ; Segment must be 0x0000.
        ;=========================================
        jmp 0x0:SETUP_ADDR      ; execute setup

        jmp $                           ; or die!
```

setup.asm이 실행될 때는 세그먼트가 0x0 이고 오프셋 주소가 0x9000 이다. 오프셋 주소가 바로 물리 주소로 변환될 수 있도록 하기 위해 점프 명령의 오퍼랜드에 0x0:0x9000 으로 지정했다.

2. setup.asm

```
swapper_pg_dir   equ 0x3000
pg0                             equ 0x4000

GDT_ADDR                equ 0x1000
TSS_ADDR                equ 0x2000

KERNEL_REAL_ADDR        equ 0xA000
KERNEL_VIRT_ADDR        equ 0x100000


KSECT equ KSIZE
```

swapper_pg_dir은 커널 부팅에서 임시로 사용할 페이지 디렉토리이다. pg0도 임시로 사용할 페이지 테이블이고 각각 물리주소 0x3000, 0x4000에 저장된다. 추후에 커널 부팅이 끝나고 유저 태스크가 실행될 때 유저 태스크의 페이지 디렉토리에 모두 복사되고 직접 사용되지 않는다. 커널과 유저 태스크가 4GB의 가상 메모리 공간을 나눠서 공존하게 되므로 유저 태스크의 페이지 디렉토리에도 커널이 사용할 공간에 대한 설정이 있어야 한다.

```
[bits 16]
[org 0x9000]

setup_start:

        mov ax, cs              ; cs=ds=es=0x0
        mov ds, ax
        mov es, ax
```

```
mov ax, 0xb800
mov es, ax
mov bx, 0x2
mov al, byte [setup_msg]
mov byte [es:bx], al
```

bootsect.asm에서 오프셋 주소를 0x9000으로 설정하고 세그먼트 주소를 0x0으로 설정했으므로 org 0x9000을 선언했다. 오프셋 주소를 계산할 때 소스 코드내의 오프셋 주소에 0x9000을 더하게 된다. 그리고 setup.asm이 시작되었음을 알리기 위해 'a' 문자를 화면에 출력한다.

```
; clear TSS
; 0x6000 ~ 0x6FFF is allocated for TSS
mov edi, TSS_ADDR
xor eax, eax
mov ecx, 26
cld
rep stosd

;
; make TSS segment descriptor
;
xor eax, eax
mov eax, TSS_ADDR
mov [descriptor4+2], ax
shr eax, 16
mov [descriptor4+4], al
mov [descriptor4+7], ah
```

유저 모드 태스크를 만들고 컨텍스트 스위칭을 하기 위해서는 TSS 영역을 만들어야 한다. 물리 메모리 0x2000 에서 시작하는 한 페이지를 0으로 클리어하고, GDT에 TSS 영역을 가르키는 디스크립터를 만든다. 이 TSS 영역은 나중에 스케줄러에서 태스크 스위칭에 사용된다.

```
lea si, [gdt]         ; source : ds:si=0x0:gdt
xor ax, ax
mov es, ax                    ; destination : es:di=0x0:0x1000
mov di, GDT_ADDR
```

```
        mov cx, 8*7         ; 7 descriptors

        rep movsb

        cli
        lgdt [gdtr]
```

setup.asm 파일의 끝부분에 GDT가 정의되어 있는데 이것을 물리 메모리 0x1000 에 복사한다. 0x1000은 시스템의 물리 메모리에서 2번째 페이지 프레임이 된다. 첫번째 페이지 프레임, 즉 0x0 번지에는 IDT를 저장한다. gdtr 이라는 변수에는 미리 GDT의 크기 0x1000과 GDT가 시작하는 물리 주소 0x1000을 정의해놓고 lgdt 명령으로 gdtr 변수의 내용을 프로세서 레지스터 GDTR에 저장한다.
다음 코드가 setup.asm에서 만들어진 gdt의 정의이다.

```
gdtr:
        dw 0x1000
        dd GDT_ADDR

gdt:
        dd 0x00000000, 0x00000000
        dd 0x0000FFFF, 0x00CF9A00 ; kernel code segment
        dd 0x0000FFFF, 0x00CF9200 ; kernel data segment
        dd 0x8000FFFF, 0x0040920B ; video memory segment



        ;dd 0x00000000, 0x00000000          ; for TSS descriptor
descriptor4:
        dw 104
        dw 0
        db 0
        db 0x89
        db 0
        db 0

        dd 0x0000FFFF, 0x00cfFA00    ; user code segment
        dd 0x0000FFFF, 0x00cfF200    ; user data segment
        dd 0x00000000, 0x00000000
gdt_end:
```

다음으로 보호모드로 진입하고 세그먼테이션 기능을 시작한다.

```
        mov eax, cr0
        or eax, 0x0000001
        mov cr0, eax

        jmp $+2
        nop
        nop

        jmp dword KERNEL_CODE_SEL:protect_start
```

GDT 영역을 설정하고 GDTR 레지스터에 값을 저장했으면 cr0의 0번 비트를 1로 셋팅해서 보호모드를 시작한다. 그런데 보호모드에 진입한 후에도 프로세서 내부의 명령어 캐쉬에는 16비트 크기의 명령어 코드가 들어있을 것이다. 프로그램 초기에 [bits 16] 지시어를 사용했기 때문이다. 따라서 보호모드로 진입한 직후에는 [bits 32]로 선언한 코드가 실행되도록 해야 한다. 위의 코드와 같이 세그먼트 셀렉터와 오프셋 주소를 지정한 long jump를 실행하면 명령어 캐쉬가 비워지면서 세그먼트 레지스터의 값과 EIP 값을 모두 바꿀 수 있다.

```
[bits 32]
protect_start:
        mov ax, KERNEL_DATA_SEL
        mov ds, ax
        mov es, ax
        mov fs, ax
        mov gs, ax
        mov ss, ax

        mov edi, 0xb8004
        mov al, byte [protect_msg]
        mov byte [edi], al
```

이제부터 보호모드가 시작되었으므로 32비트 코드 크기로 실행된다. bits 32 지시어로 어셈블러가 32비트 코드를 생성하도록 한다. fd, es, fs, gs, ss는 데이터 세그먼트로 사용되므로 커널 데이터 세그먼트 셀렉터 값을 저장한다. 그리고 보호모드 진입이 이루어진 것을 알 수 있도록 'O' 문자를 출력한다.

```
        ; check physical memory size
```

```
        mov esi, 0x100000
        mov eax, 0x1234abcd
        mov ecx, 0
check_mem_size:
        inc ecx
        mov dword [ds:esi], eax
        mov ebx, dword [ds:esi]
        add esi, 0x100000
        cmp eax, ebx
        je check_mem_size
        mov [memory_size], ecx
```

물리 메모리의 크기를 알아내기 위해서 1MB 단위로 0x1234abcd 값을 쓰고 읽는다. 현재까지 0x100000 번지보다 상위 메모리에는 어떤 데이터도 쓰지 않았으므로 0x100000, 0x200000, ... 번지마다 0x1234abcd 값을 쓰고 읽어본다. 만약 같은 값이 읽혀지면 해당 번지에 메모리가 존재하는 것이므로 값이 안읽혀질 때까지 반복하면 메모리 크기를 알 수 있다.

```
        ;
        ; kernel exists at 0x100000~
        ;
        mov ax, KSECT
        mov dx, 512                 ; sector = 512byte
        mul dx
        mov cx, ax

        mov edx, 0

        mov esi, KERNEL_REAL_ADDR
        mov edi, KERNEL_VIRT_ADDR

copy_kernel:
        inc edx
        mov al, byte [ds:esi]
        mov byte [es:edi], al
        inc esi
        inc edi
        dec cx
        jnz copy_kernel
```

```
        mov [kernel_size], edx
```

커널을 물리메모리 1MB 지점으로 복사한다. bootsect.asm에서는 메모리 주소가 16비트까지만 지정되므로 커널 이미지를 0xA000 에 저장할 수 밖에 없었다. 하지만 현재는 32비트 주소 지정이 가능하므로 0x100000 에 복사한다. 한 바이트씩 복사하고 복사된 크기를 edx 레지스터에 기록했다가 kernel_size 변수에 저장해놓는다.

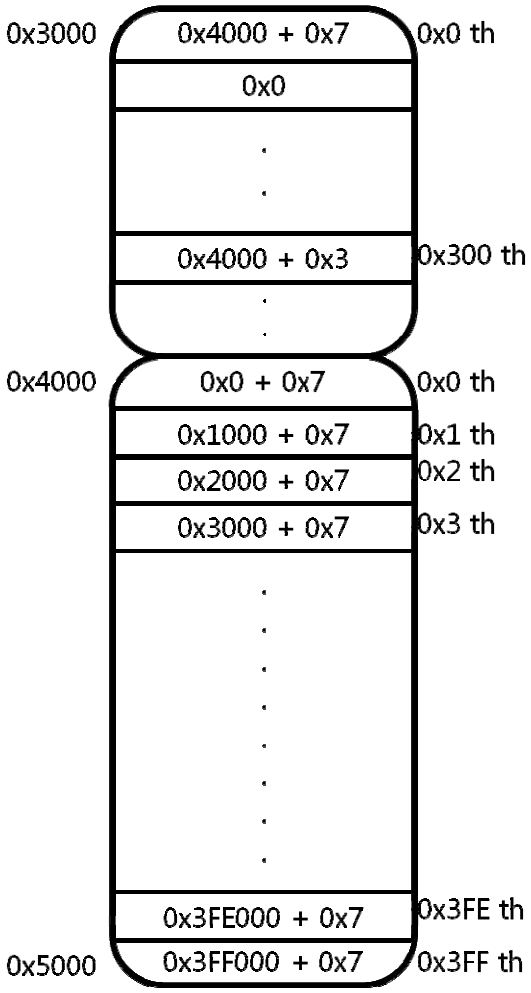
```
        ; set PAGE DIRECTORY

        mov edi, swapper_pg_dir+0x0*4               ; 0th table <- 0x3000
        mov eax, pg0
        or eax, 0x7                             ; 0~4MB is not accesable to user task
        mov [es:edi], eax

        mov edi, swapper_pg_dir+0x300*4 ; 0x300th table <- 0x3000
        mov eax, pg0
        or eax, 0x3
        mov [es:edi], eax

        ;
        ; set PAGE TABLE for global tables
        ; physical memory 0x0~0x400000 is mapped to
        ; virtual memory 0x0~0x400000(4MB) & 0xC0000000 ~ 4MB
        ;
        mov edi, pg0                ; first kernel page table at 0x4000
        mov eax, 0x00000007
        mov ecx, 1024
set_tbl_table:
        mov [es:edi], eax
        add eax, 0x1000
        add edi, 4
        dec ecx
        jnz set_tbl_table
```

페이지 디렉토리와 페이지 테이블을 만든다. 다음 그림과 같은 모양이 된다.



페이지 디렉토리에는 0번째와 0x300번째 엔트리에만 페이지 테이블의 주소를 써넣는다. 그리고 페이지 테이블에는 물리 메모리 0번지부터 0x3FF000 번지 (0번 페이지 ~ 0x3FF번 페이지)의 주소를 써넣는다. 이것의 의미는 물리 메모리의 처음 4MB를 가상주소 0x0~0x3FFFFF 번지에도 매핑하고, 또한 가상주소 0xC000 0000 ~ 0xC03F FFFF에도 매핑한다는 것이다. 만약 가상 메모리 0xC000 10000 번지에 데이터를 쓰면 물리 메모리 0x1000 번지에 값을 쓰게 되고 또한 가상 메모리 0x1000 번지에 데이터를 써도 같은 물리 메모리 0x1000 번지에 값이 써진다는 것이다.

이렇게 페이징을 설정하는 이유는 0xB8000 번지에 있는 비디오 메모리나 GDT, TSS 영역 등을 커널 모드뿐 아니라 유저 태스크에서도 접근할 수 있도록 하기 위해서이다. 만약 유저 태스크가 하위 4MB 영역에 접근할 수 없도록 하기 위해서는 페이지 디렉토리나 페이지 테이블의 특성 값으로 0x3을 써넣으면 된다. 또한 이 하위 4MB 영역에는 GDT, TSS, IDT, 비디어 메모리가 있고 커널이 이 영역에 계속 값을 쓰고 읽으므로 가상 주소를 물리 주소로 바꾸기가 쉬워야 한다. 그래서 페이지 순서를 그대로 메모리 주소 순으로 배치한 것이다. 추후에는 모든 물리 메모리 영역을 0xC000 0000 에 매핑해서 가상 메모리 주소만 있어도 모든 물리 메모리에 접근할 수 있도록 할 것이다.

```
mov eax, swapper_pg_dir
mov cr3, eax

mov eax, cr0
or eax, 0x80000000
mov cr0, eax

; stack space is grow down from 0x9ffff
;mov esp, 0xc009ffff

mov edx, [kernel_size]
mov ebx, [memory_size]


jmp 0xc0100000
```

페이징 기능을 사용하기 위해서 cr3 레지스터에 페이지 디렉토리의 주소를 저장하고 cr0의 31 번 비트를 1로 셋팅한다.

커널 이미지의 크기와 물리 메모리의 크기를 커널에 전달하기 위해서 edx, ebx 레지스터에 값을 저장한다. 그러면 setup.asm 직후에 실행될 head.asm 에서 이 레지스터의 값을 읽고 변수에 저장한다. 커널이 물리 메모리 0x100000 에 저장되어 있으므로 가상 주소 0xC010 0000 으로 점프한다.

```
tss: times 104 db 0

times 2048-($-$$) db 0
```

TSS 영역은 항상 104 바이트 이상이 되어야 한다. 만약 이보다 작으면 프로세서가 예외를 발생하고 보호모드로 진입을 할 수 없다. setup.asm 파일에는 gdt 등 데이터가 많이 저장되므로 이미지 크기를 2048 바이트로 만들었다.

3. head.asm

```
[bits 32]
```

```
%include "selector.inc"

extern start_kernel
extern phy_mem_size
extern kernel_size

section .text
global _start

_start:

        mov dword [kernel_size], edx
        mov dword [phy_mem_size], ebx

        mov edi, 0xC00b8006
        mov al, byte [kernel_msg]
        mov byte [edi], al

        ; stack space is grow down from 0x9ffff
        mov esp, 0xc009fffc

        call start_kernel

        jmp $

kernel_msg      db "S", 0
```

edx와 ebx 레지스터의 값을 변수에 저장한다. 그리고 커널에 진입했음을 알 수 있도록 'S' 문자를 화면에 출력하고 스택 포인터 값을 사용하지 않는 메모리 영역 중에서 적당한 곳으로 설정한다. C 함수를 사용하기 위해서 스택이 반드시 사용되므로 꼭 스택 포인터 값을 적당하게 설정해야 한다. 그리고 head.asm 부터 C 소스의 데이터나 함수를 사용할 수 있으므로, extern 지시어를 사용해서 원하는 함수나 변수에 접근한다. head.asm은 C 로 작성된 함수가 실행되기 전에 스택 포인터를 설정하는 일을 주로 한다.

start_kernel() 함수부터는 프로세서의 초기화가 끝났으므로 프로세서의 레지스터에 직접적으로 접근할 경우가 적으므로 대부분의 코드를 C 언어로 작성한다.