주간 업무 사항 정리

작성자	제품팀 이민성 인턴
업무 일시	2024.01.31 – 2024.02.02

세부 사항

1. 업무 내역 요약 정리

Plan	To-do
1. 부트로더 개발 - 정의 - 간단한 부트로더 개발 - 문자를 출력하는 부트로더 개발 2. 하드디스트 읽기 모듈 개발 - 하드디스크의 실린더와 헤드 - 섹터 - 하드디스크 내 특정 섹터 읽기 3. 모드 전환 모듈 개발 - 리얼모드와 보호모드에 대한 기본 개념 - 리얼모드 환경에서의 세그먼트:오프셋 구조 - 리얼모드에서 보호모드로의 전환 4. 함수 만들기 - 어셈블리어로 함수 만들기 - C언어로 함수 만들기 - 개발의 편의를 위해 makefile 만들기 - C언어로 함수 만들기 2 5. 인터럽트 핸들러 개발 - PIC 세팅 - IDT 선언 - IDT 구현 6. 키보드 드라이버 개발 - 키보드 드라이버 1 - 키보드 드라이버 2 7. 쉘 개발 - 쉘과 cli의 차이점	1. 오류 발생

CLI는 사용자와 컴퓨터 시스템 간의 상호작용 방식을 일컫는 반면, 쉘은 그러한 상호작용을 가능하게 하는 구체적인 소프트웨어를 가리킵니다.
모든 쉘은 CLI를 제공하지만, 모든 CLI가 쉘은 아닙니다. 예를 들어, 애플리케이션 내부에 CLI 기능이 내장되어 있을 수 있지만, 그것이 운영 체제의 쉘이라고 할 수는 없습니다.
쉘은 사용자가 시스템과 상호작용하는 많은 방법 중 하나이며, CLI는 그러한 상호작용의 형태 중 하나입니다.

- 기초적인 Shell

8. 하드디스크 드라이버 개발
- 하드디스크 드라이버
- Qemu
- 읽기
- 쓰기

9. 파일 시스템(ext2) 개발
- printf() 가변인자 구현
- Superblock
- Groupblock
- Bitmap
- Inode & ls
- cd
- 현재 Directory Path
- cat
- Block alloc & free
- Inode alloc & free
- mkdir
- rm

2. 내용 세부 (업무 세부 내역 정리 및 기타 사항 정리)

1. 오류 발생

저번 주차에 아래와 같이 interrupt.c의 init_intdesc()와 idt_keyboard() 함수를 수정해서 키보드를 눌렀을 때 어떤 키를 눌렀는지 파악할 수 있도록 하려고 했습니다.

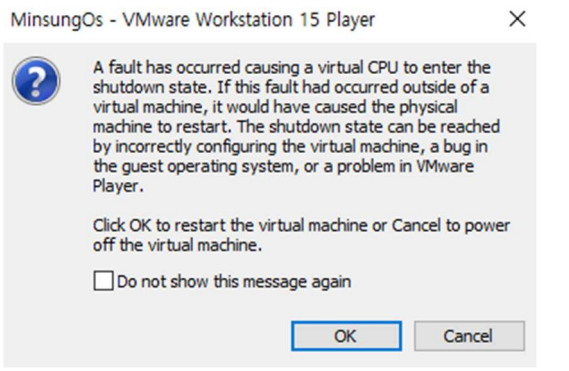
```
unsigned char keybuf;
```

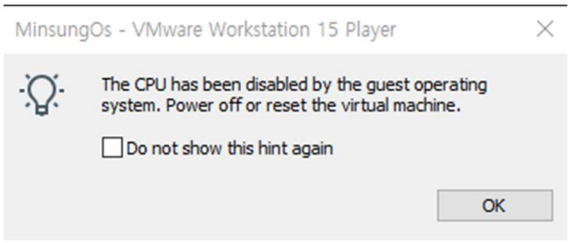
```
    // 키보드 작동
__asm__ __volatile__
(
    "mov al, 0xAE;"
    "out 0x64, al;"
);
//  인터럽트 작동 시작
```

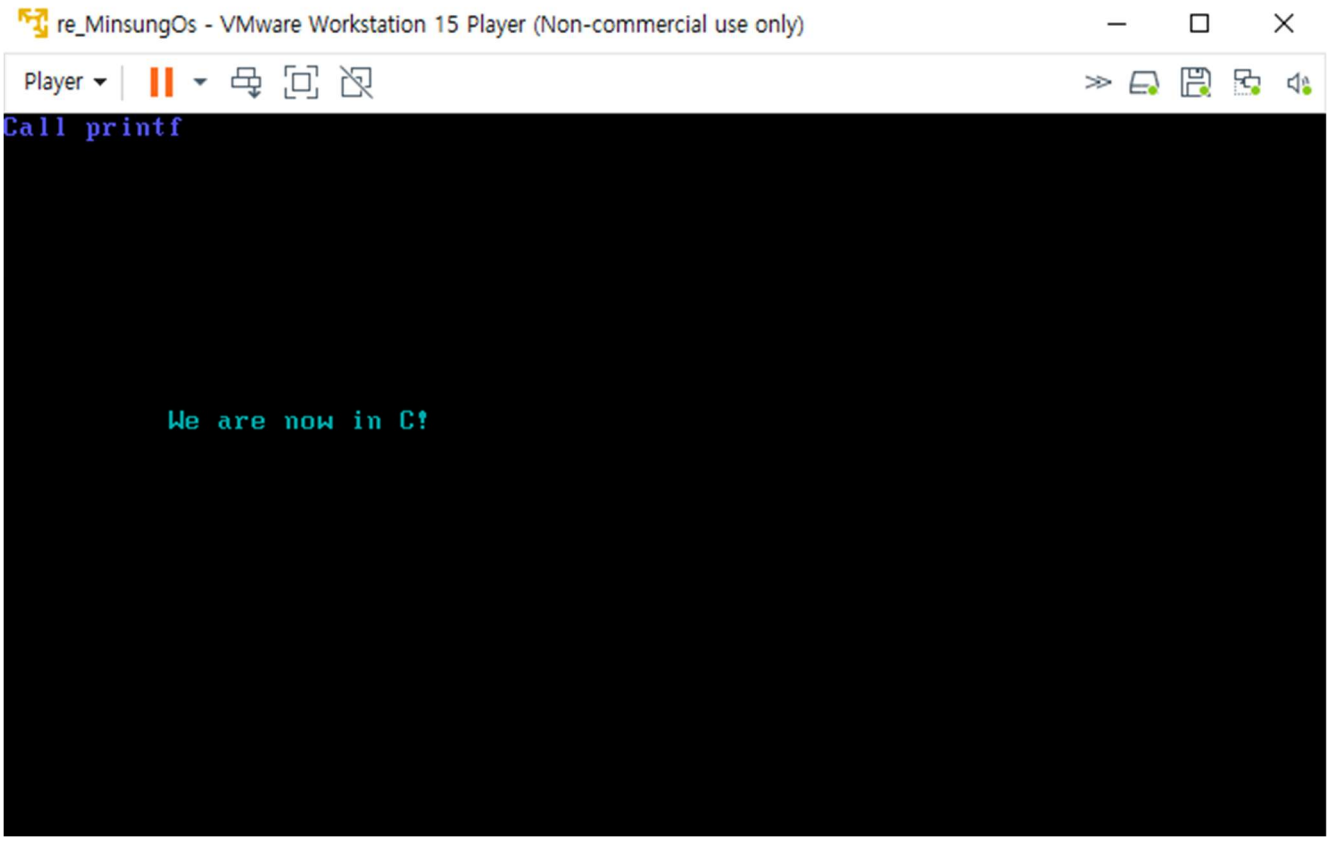
```
void idt_keyboard()
{
    __asm__ __volatile__
    (
        "push gs;"
        "push fs;"
        "push es;"
        "push ds;"
        "pushad;"
        "pushfd;"
        "xor al,al;"
        "in al, 0x60;"
    );

    __asm__ __volatile__("mov %0, al;" :"=r"(keybuf) );

    kprintf(&keybuf, 8, 40);

    __asm__ __volatile__
    (
        "mov al, 0x20;"
        "out 0x20, al;"
    );

    __asm__ __volatile__
    (
        "popfd;"
        "popad;"
        "pop ds;"
        "pop es;"
        "pop fs;"
        "pop gs;"
        "leave;"
        "nop;"
        "iretd;"
    );
}
```

```
minsung@ubuntu:~/Dev/Os/Keyboard$ make
nasm -f bin -o Boot.img Boot.asm
nasm -f bin -o Sector2.img Sector2.asm
gcc -c -masm=intel -m32 -ffreestanding main.c -o main.o
gcc -c -masm=intel -m32 -ffreestanding function.c -o function.o
gcc -c -masm=intel -m32 -ffreestanding interrupt.c -o interrupt.o
ld -melf_i386 -Ttext 0x10200 -nostdlib main.o function.o interrupt.o -o main.img
ld: warning: cannot find entry symbol _start; defaulting to 0000000000010200
objcopy -O binary main.img disk.img
cat Boot.img Sector2.img disk.img > final.img
```

위와 같이 컴파일 후 가상 머신에 올려보니 아래와 같은 오류가 발생하였고 VmWare에 idt_ignore(), idt_timer(), idt_keyboard()에 있는 kprintf가 작동을 하지 않아 출력을 안하는 오류가 발생하였습니다.

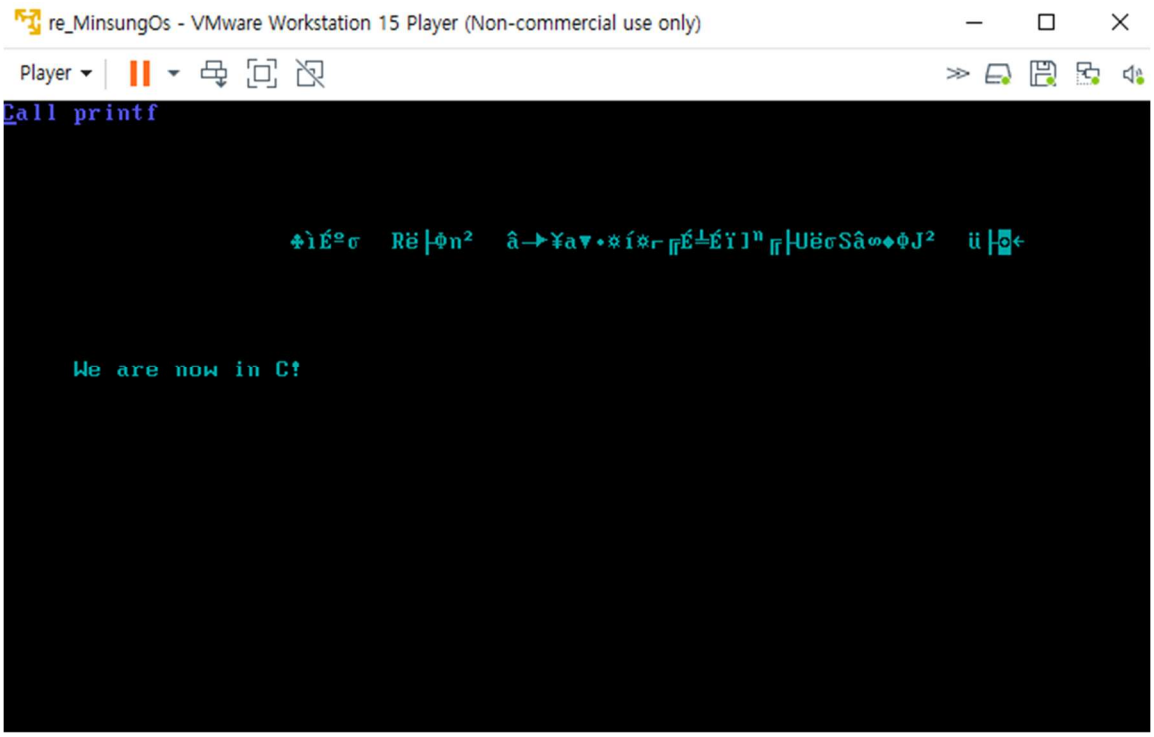






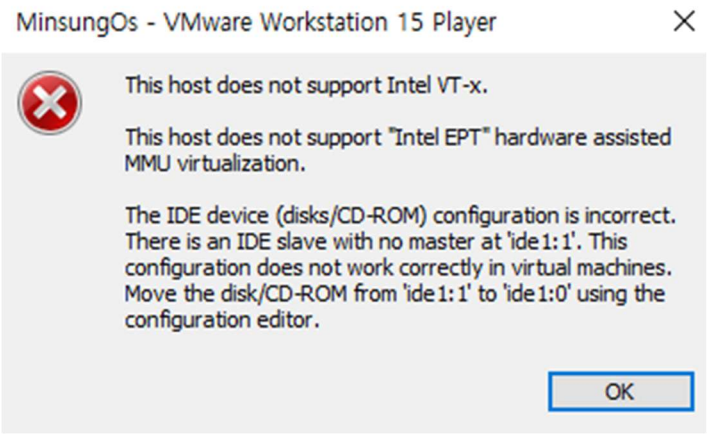
몇일 동안 이것저것 바꿔보고 살펴보고 해도 오류의 원인을 찾지 못하였습니다.
그래서 저번 주차에 idt를 구현한 부분부터 다시 보았는데 interrupt.c의 idt_ignore() 함수에서 kprintf("idt_ignore", 5, 40)을 하면 5 행 40 열에 idt_ignore이 나와야 하는데 아래 사진과 같이 이상한 문자가 뜬다는 것을 파악하였습니다.

그래서 코드에 실수가 있나 찾아보았는데 아무리 찾아봐도 없어서 블로그에 있는 코드를 그대로 복사하여 컴파일 후 실행해보았습니다.



그랬더니 위와 같이 더 이상한 문자가 쓰였습니다. 그래서 원인을 찾아보았습니다.

또 다른 가상머신을 만들어서 작동해보니 이번에는 아래와 같은 오류가 발생하였습니다.



블로그와 실행 환경(컴파일러 버전, 리눅스 버전, CPU 종류 등)이 달라서 오류가 발생하는 것으로 파악하였습니다. 스스로 고쳐서 실행을 할 수 없다고 판단하여 실행을 포기하고 블로그에 나와있는 내용으로 코드(수정한 interrupt.c)를 분석해보면 공부가 되겠다고 생각하여 분석해보겠습니다.
실행은 분석 후 블로그에 나와있는 img 파일로 하겠습니다.

```
// interrupt.c

#include "interrupt.h"
interrupt.h 라는 헤더파일을 현재 코드 파일에 포함시킵니다. 이 헤더 파일에는 인터럽트 처리에 필요한 데이터 타입이나 함수 등이 선언되어 있습니다.

#include "function.h"
function.h 라는 헤더 파일을 현재 코드 파일에 포함시킵니다. 이 헤더 파일에는 이 코드에서 사용되는 함수들의 선언이 포함되어 있을 수 있습니다.

struct IDT inttable[3];
IDT(Interrupt Descriptor Table)라는 구조체 타입으로 inttable 이라는 이름의 배열을 선언합니다. 이 배열은 3 개의 IDT 항목을 저장할 수 있습니다.

struct IDTR idtr = { 256 * 8 - 1,0 };
IDTR(Interrupt Descriptor Table Register)라는 구조체 타입으로 idtr 이라는 이름의 변수를 선언하고 초기화합니다. 이 구조체는 IDT 의 크기와 시작 주소를 저장하는 데 사용됩니다.

unsigned char keyt[2] = { 'A', 0 };
unsigned char 타입의 keyt 라는 이름의 배열을 선언하고 'A'와 0 으로 초기화합니다. 이 배열은 키보드 입력을 저장하는 데 사용됩니다.

unsigned char keybuf;
unsigned char 타입의 keybuf 라는 이름의 변수를 선언합니다. 이 변수는 키보드에서 입력 받은 키 값을 임시로 저장하는 데 사용됩니다.
```

```
static unsigned char keyboard[160] = { 0, };
```
static 이라는 키워드를 사용하여 unsigned char 타입의 keyboard 라는 이름의 배열을 선언하고 0 으로 초기화합니다. 이 배열은 키보드에서 입력 받은 키 값들을 저장하는 데 사용됩니다.

```
static unsigned short index = 0;
```
static 이라는 키워드를 사용하여 unsigned short 타입의 index 라는 이름의 변수를 선언하고 0 으로 초기화합니다. 이 변수는 keyboard 배열에 대한 현재 인덱스를 저장하는 데 사용됩니다.

```
void init_intdesc()
{
    int i,j;
    unsigned int ptr;
```
ptr 변수는 ISR(Interrupt Service Routine)의 주소를 저장하기 위한 32 비트 정수형 변수입니다. 이 변수는 함수의 주소를 임시로 담는 데 사용됩니다.

```
    unsigned short *isr;
```
ISR 의 주소를 저장하기 위한 포인터 변수 isr 를 선언합니다. 하지만 ptr 과 달리 isr 은 IDT(Interrupt Descriptor Table) 내의 특정 항목을 가리키는 데 사용됩니다. 이를 통해 IDT 내의 각 항목에 ISR 의 주소를 저장할 수 있습니다.

```
    {  // 0x00 : isr_ignore
        ptr = (unsigned int)idt_ignore;
```
idt_ignore 함수의 주소를 ptr 변수에 저장합니다. 여기서 idt_ignore 는 특정 인터럽트에 대해 아무런 동작도 하지 않는 무시 처리 함수입니다.

```
        inttable[0].selector = (unsigned short)0x08;
```
(unsigned short)0x08;: selector 필드에는 세그먼트 셀렉터 값을 저장합니다. 여기서 0x08 은 코드 세그먼트의 셀렉터를 나타냅니다. 이는 GDT(Global Descriptor Table)에서 코드 세그먼트를 참조하는 인덱스를 의미하며, 보호 모드에서 사용되는 세그먼트 기반 메모리 관리의 일부입니다.

```
        inttable[0].type = (unsigned short)0x8E00;
```
type 필드에는 인터럽트 게이트의 속성을 나타내는 값인 0x8E00 을 저장합니다. 0x8E 는 이진수로 10001110 을 의미하며, 각 비트는 다음과 같은 의미를 가집니다:

1: 세그먼트 존재 비트(Present Bit), 인터럽트가 사용 가능함을 나타냅니다.

00: Descriptor Privilege Level(DPL), 인터럽트 게이트가 호출될 수 있는 최소 권한 레벨을 나타냅니다. 여기서는 00 이므로 권한 레벨 0(커널 모드)에서만 호출 가능합니다.

01110: Gate Type, 인터럽트 게이트임을 나타내며, 이는 인터럽트가 발생했을 때 사용되는 게이트 타입입니다.

```
        inttable[0].offsetl = (unsigned short)(ptr & 0xFFFF);
        ptr 의 하위 16 비트를 추출하여 offsetl 필드에 저장합니다. & 0xFFFF 연산은 32 비트
        주소 중 하위 16 비트만을 마스킹하여 추출하는 데 사용됩니다.

        inttable[0].offseth = (unsigned short)(ptr >> 16);
        ptr 의 상위 16 비트를  추출하여 offseth 필드에 저장합니다. >> 16 연산은 32 비트
        주소를 16 비트 오른쪽으로 시프트하여 상위 16 비트를 하위 16 비트 위치로 이동시키는
        데 사용됩니다.
    }

    {  // 0x01 : isr_timer
        ptr = (unsigned int)idt_timer;
        inttable[1].selector = (unsigned short)0x08;
        inttable[1].type = (unsigned short)0x8E00;
        inttable[1].offsetl = (unsigned short)(ptr & 0xFFFF);
        inttable[1].offseth = (unsigned short)(ptr >> 16);
    }

    {  // 0x02 : isr_keyboard
        ptr = (unsigned int)idt_keyboard;
        inttable[2].selector = (unsigned short)0x08;
        inttable[2].type = (unsigned short)0x8E00;
        inttable[2].offsetl = (unsigned short)(ptr & 0xFFFF);
        inttable[2].offseth = (unsigned short)(ptr >> 16);
    }

동일한 로직으로 isr_timer 와 isr_keyboard 에 대해서도 설정을 합니다. 각각의 인터럽트
핸들러에 대해 selector, type, offsetl, offseth 값을 설정함으로써, 해당 인터럽트가
발생했을 때 처리할 함수의 주소와 인터럽트의 속성을 IDT 에 등록하게 됩니다.
이렇게 설정된 IDT 는 CPU 가 인터럽트를 받았을 때, 어떤 함수를 실행해야 하는지를 알게
해주며, 시스템의 인터럽트 처리 메커니즘에 필수적인 역할을 합니다.

    for (i = 0; i < 256; i++)
    이 코드는 i 가 0 부터 255 까지 변화하면서 반복되는 for 반복문을 시작합니다. 이 반복문은
    모든 인터럽트 벡터에 대해 처리를 설정합니다.
    {
        isr = (unsigned short*)(0x0 + i * 8);
        인터럽트 서비스 루틴(ISR)의 주소를 가리키는 포인터 변수 isr 에 현재 인터럽트
        벡터의 IDT 항목 위치를 저장합니다.

        *isr = inttable[0].offsetl;
```

```
        IDT 항목의 오프셋 하위 부분에 무시 처리를 하는 핸들러의 오프셋 하위 부분을
        저장합니다.

        *(isr + 1) = inttable[0].selector;
        IDT 항목의 세그먼트 선택자 부분에 무시 처리를 하는 핸들러의 세그먼트 선택자를
        저장합니다.

        *(isr + 2) = inttable[0].type;
        IDT 항목의 타입 속성 부분에 무시 처리를 하는 핸들러의 타입 속성을 저장합니다.

        *(isr + 3) = inttable[0].offseth;
        IDT 항목의 오프셋 상위 부분에 무시 처리를 하는 핸들러의 오프셋 상위 부분을
        저장합니다.
    }

    {
        isr = (unsigned short*)(0x0 + 8 * 0x20);
        *isr = inttable[1].offsetl;
        *(isr + 1) = inttable[1].selector;
        *(isr + 2) = inttable[1].type;
        *(isr + 3) = inttable[1].offseth;
    }
이 부분은 0x20(타이머 인터럽트) 벡터에 대해 핸들러를 설정합니다. 핸들러의 정보를 해당
인터럽트 벡터의 IDT 항목에 저장합니다. 이를 통해 해당 인터럽트가 발생하면 설정된 핸들러가
호출되도록 합니다.

    {
        isr = (unsigned short*)(0x0 + 8 * 0x21);
        *isr = inttable[2].offsetl;
        *(isr + 1) = inttable[2].selector;
        *(isr + 2) = inttable[2].type;
        *(isr + 3) = inttable[2].offseth;
    }
이 부분은 0x21(키보드 인터럽트) 벡터에 대해 핸들러를 설정합니다. 핸들러의 정보를 해당
인터럽트 벡터의 IDT 항목에 저장합니다. 이를 통해 해당 인터럽트가 발생하면 설정된 핸들러가
호출되도록 합니다.

    __asm__ __volatile__
    __asm__은 C 나 C++ 코드 내에서 어셈블리 코드를 작성할 수 있게 해주는 키워드입니다.
    __volatile__은 컴파일러에게 이 코드를 최적화하지 말고 원래의 순서대로 유지하도록
    지시하는 키워드입니다.
    (
```

```
        "mov al, 0xAE;"
        이 부분은 AL 레지스터(AX 레지스터의 하위 8 비트)에 0xAE 값을 저장하는 어셈블리
        코드입니다.

        "out 0x64, al;"
        이 부분은 AL 레지스터의 값을 0x64 포트에 출력하는 어셈블리 코드입니다.

이 코드는 키보드 컨트롤러의 커맨드 포트에 값을 보내는 역할을 합니다. 0xAE 커맨드는 키보드
인터페이스를 활성화하는 데 사용됩니다. 이를 통해 키보드 인터럽트를 받을 수 있게 됩니다.
    );

    __asm__ __volatile__("mov eax, %0"::"r"(&idtr));
    이 코드는 idtr 의 주소를 EAX 레지스터에 저장하는 어셈블리 코드입니다. 여기서 %0 는 첫
    번째 입력 인자(이 경우 &idtr)를 나타냅니다. "r"은 일반 레지스터를 나타냅니다.

    __asm__ __volatile__("lidt [eax]");
    이 코드는 EAX 레지스터가 가리키는 주소(즉, idtr 의 주소)에 있는 IDT 를 CPU 에 로드하는
    어셈블리 코드입니다. lidt 는 Load Interrupt Descriptor Table 의 약자입니다.

    __asm__ __volatile__("mov al,0xFC");
    이 코드는 AL 레지스터(AX 레지스터의 하위 8 비트)에 0xFC 값을 저장하는 어셈블리
    코드입니다.

    __asm__ __volatile__("out 0x21,al");
    이 코드는 AL 레지스터의 값을 0x21 포트에 출력하는 어셈블리 코드입니다. 이 코드는
    PIC(Programmable Interrupt Controller)의 IMR(Interrupt Mask Register)에 값을
    보내는 역할을 합니다. 0xFC 는 특정 인터럽트를 활성화하거나 비활성화하는 데 사용됩니다.

    __asm__ __volatile__("sti");
    이 코드는 AL 레지스터의 값을 0x21 포트에 출력하는 어셈블리 코드입니다. 이 코드는
    PIC(Programmable Interrupt Controller)의 IMR(Interrupt Mask Register)에 값을
    보내는 역할을 합니다. 0xFC 는 특정 인터럽트를 활성화하거나 비활성화하는 데 사용됩니다.

    return;
}

void idt_ignore()
{
    __asm__ __volatile__
    이 키워드로 시작하는 부분은 인라인 어셈블리 코드입니다. 이 코드는 인터럽트가 발생했을
    때 CPU 상태를 저장하고 인터럽트를 처리한 후 원래 상태로 복구하는 역할을 합니다.
    (
```

```
        "push gs;"
        "push fs;"
        "push es;"
        "push ds;"
        "pushad;"
        "pushfd;"
        이 부분에서는 현재 CPU의 상태를 스택에 저장합니다. 이는 나중에 인터럽트 처리가
        끝나면 원래 상태로 복구하기 위한 것입니다.

        "mov al, 0x20;"
        "out 0x20, al;"
        이 부분에서는 인터럽트가 처리되었음을 인터럽트 컨트롤러에 알립니다. 0x20은
        인터럽트 컨트롤러의 커맨드 포트 주소이며, 0x20 값을 보내는 것은 현재 인터럽트
        처리가 완료되었음을 의미합니다.
    );

    kprintf("idt_ignore", 5, 40);
    kprintf 함수를 호출하여 "idt_ignore"라는 문자열을 출력합니다.

    __asm__ __volatile__
    (
        "popfd;"
        "popad;"
        "pop ds;"
        "pop es;"
        "pop fs;"
        "pop gs;"
        "leave;"
        "nop;"
        "iretd;"
        이 부분에서는 앞서 스택에 저장했던 CPU의 상태를 복구합니다. 마지막의 iretd
        명령은 인터럽트 처리가 끝났음을 CPU에 알리고 원래의 코드 실행으로 돌아가게 합니다.
    );
}

void idt_timer()
{
    __asm__ __volatile__
    (
        "push gs;"
        "push fs;"
        "push es;"
```

```
        "push ds;"
        "pushad;"
        "pushfd;"
        "mov al, 0x20;"
        "out 0x20, al;"
    );

    kprintf(keyt, 7, 40);
    keyt[0]++;

    __asm__ __volatile__
    (
        "popfd;"
        "popad;"
        "pop ds;"
        "pop es;"
        "pop fs;"
        "pop gs;"
        "leave;"
        "nop;"
        "iretd;"
    );
}

void idt_keyboard()
{
    __asm__ __volatile__
    (
        "push gs;"
        "push fs;"
        "push es;"
        "push ds;"
        "pushad;"
        "pushfd;"
        "xor al,al;"
        AL 레지스터를 0으로 초기화

        "in al, 0x60;"
        키보드 컨트롤러의 데이터 포트(0x60)에서 스캔 코드를 읽어 AL 레지스터에 저장
    );

    __asm__ __volatile__("mov %0, al;" :"=r"(keybuf) );
```

이 코드는 AL 레지스터의 값을 keybuf 변수에 저장합니다.

```
keybuf = transScan(keybuf);
```

transScan 함수를 호출하여 스캔 코드를 키보드 입력 값으로 변환하고, 그 결과를 keybuf 에 저장합니다.

```
if (keybuf == 0x08 && index != 0)
    keyboard[--index] = 0;
else if (keybuf != 0xFF && keybuf != 0x08)
    keyboard[index++] = keybuf;
```

이 부분은 키보드 입력을 처리하는 코드입니다. Backspace 키(스캔 코드 0x08)가 눌렸을 때는 입력 문자열에서 마지막 문자를 제거하고, 그 외의 키가 눌렸을 때는 해당 키의 값을 입력 문자열에 추가합니다.

```
kprintf_line_clear(8);
kprintf(keyboard,8,0);
```

이 부분에서는 화면의 8 번째 줄을 지우고, 키보드 입력 문자열을 그 위치에 출력합니다.

```
__asm__ __volatile__
(
    "mov al, 0x20;"
    "out 0x20, al;"
);

__asm__ __volatile__
(
    "popfd;"
    "popad;"
    "pop ds;"
    "pop es;"
    "pop fs;"
    "pop gs;"
    "leave;"
    "nop;"
    "iretd;"
);
}
```

transScan 함수는 키보드에서 입력받은 스캔 코드를 해당하는 문자로 변환하는 역할을 합니다.

```
unsigned char transScan(unsigned char target)
```

transScan 이라는 이름의 함수를 선언하며, 입력 인자로 target 이라는 이름의 unsigned char 타입의 변수를 받습니다. 이 함수는 변환된 문자를 unsigned char 타입으로 반환합니다.

```
{
    unsigned char result;
    변환된 문자를 저장할 변수 result 를 선언합니다.

    switch (target)
    switch 문을 사용하여 target 값에 따라 다른 작업을 수행합니다.
    {
    case 0x1E: result = 'a'; break;
    case 0x30: result = 'b'; break;
    case 0x2E: result = 'c'; break;
    case 0x20: result = 'd'; break;
    case 0x12: result = 'e'; break;
    case 0x21: result = 'f'; break;
    case 0x22: result = 'g'; break;
    case 0x23: result = 'h'; break;
    case 0x17: result = 'i'; break;
    case 0x24: result = 'j'; break;
    case 0x25: result = 'k'; break;
    case 0x26: result = 'l'; break;
    case 0x32: result = 'm'; break;
    case 0x31: result = 'n'; break;
    case 0x18: result = 'o'; break;
    case 0x19: result = 'p'; break;
    case 0x10: result = 'q'; break;
    case 0x13: result = 'r'; break;
    case 0x1F: result = 's'; break;
    case 0x14: result = 't'; break;
    case 0x16: result = 'u'; break;
    case 0x2F: result = 'v'; break;
    case 0x11: result = 'w'; break;
    case 0x2D: result = 'x'; break;
    case 0x15: result = 'y'; break;
    case 0x2C: result = 'z'; break;
    case 0x39: result = ' '; break;
    case 0x0E: result = 0x08; break;
    각 case 문에서는 target 값이 해당 스캔 코드일 때 result 값을 해당하는 문자로
    설정합니다. 예를 들어, 스캔 코드 0x1E 는 'a' 키를 의미하므로 result 를 'a'로
    설정합니다. 마찬가지로 스캔 코드 0x0E 는 Backspace 키를 의미하므로 result 를 0x08 로
    설정합니다.

    default: result = 0xFF; break;
    default 문에서는 target 값이 어떤 case 문에도 해당하지 않을 때 result 값을 0xFF 로
```

```
    설정합니다. 이는 target 값이 알려진 스캔 코드가 아닐 때의 처리 방법을 나타냅니다.
    }

    return result;
    result 값을 반환하여 함수를 종료합니다. 이 result 값은 변환된 문자를 나타내거나,
    알려진 스캔 코드가 아닐 경우 0xFF 를 나타냅니다.
}
```

스캔코드로 바꾸는 이유: 키보드 인터럽트가 발생하면 키보드 컨트롤러는 키보드에서 눌린 키에 해당하는 스캔 코드를 생성합니다. 스캔 코드는 키보드의 각 키에 할당된 고유한 숫자입니다. 이 스캔 코드는 키보드 컨트롤러의 데이터 포트를 통해 CPU로 전송됩니다.
CPU는 이 스캔 코드를 받아서 해당하는 키 입력을 처리하는데, 이 때 스캔 코드를 실제 문자나 키 입력 값으로 변환해야 합니다. 이 변환 과정이 필요한 이유는 스캔 코드 자체는 키보드의 물리적인 키 위치를 나타내는 것이므로, 이것을 사용자가 이해할 수 있는 문자나 키 입력 값으로 바꿔주어야 하기 때문입니다.
따라서 transScan 함수와 같은 스캔 코드 변환 함수는 이런 변환 과정을 수행하여, 스캔 코드를 실제 키 입력 값으로 바꾸는 역할을 합니다. 이렇게 변환된 키 입력 값은 운영체제나 응용 프로그램에서 사용자의 키 입력을 처리하는 데 사용됩니다.