주간 업무 사항 정리

작성자	제품팀 이민성 인턴
업무 일시	2024.01.22 – 2024.01.25

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. 인터럽트 핸들러 개발 - 중요 내용 요약 - IDT 구현 - ISR 구현 2. 키보드 드라이버 개발 - 키보드 드라이버 1 - 키보드 드라이버 2

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. 인터럽트 핸들러 개발

- 중요 내용 요약

인터럽트 : 프로세스 실행 도중 예기치 않은 상황이 발생할 때 발생한 상황을 처리한 후 실행중인 작업으로 복귀하는 것

- PIC : 인터럽트를 관리하는 하드웨어 장치 (인터럽트 결합, 우선순위 지정, 인터럽트 마스킹, 인터럽트 벡터 리매핑을 함)
- IDT : 인터럽트 처리를 관리하는 데 사용되는 데이터 구조
- ISR: 특정 인터럽트에 대한 처리를 수행하는 함수

1. PIC는 여러 인트럽트 중 CPU가 처리해야 할 인터럽트를 결정한 뒤, 그 인터럽트를 CPU에게 전달함.
2. CPU는 PIC로부터 인터럽트를 받으면 IDT를 참조하여 해당 인터업트에 대응하는 ISR 주소를 찾음.
3. 찾은 ISR을 실행하여 인터럽트를 처리하고, 처리가 완료되면 원래의 작업으로 돌아감.

∴ 인터럽트 실행 -> PIC -> **CPU** -> IDT -> ISR -> 인터럽트 처리 -> 원래 작업 복귀

- IDT 구현

작주에 interrupt.h 구현까지 하였으니, interrupt.c를 구현해 보겠습니다.

```c
#include "interrupt.h"
#include "function.h"

struct IDT inttable[3]; // ignore, timer, keyboard
struct IDTR idtr = {256 * 8 - 1, 0};

void init_intdesc() {}
void idt_ignore() {}
void idt_timer() {}
void idt_keyboard() {}
```

IDT와 IDTR의 선언

- inttable[3]: IDT(Interrupt Descriptor Table)는 인터럽트 서비스 루틴(ISR)의 주소를 저장하는 구조체 배열입니다. 여기서는 ignore, timer, keyboard에 대한 인터럽트를 처리하기 위해 3 개의 엔트리를 선언했습니다.
- idtr: IDTR(Interrupt Descriptor Table Register)는 IDT의 크기와 시작 주소를 담는 구조체입니다. 여기서는 256 개의 인터럽트를 위해 256 * 8 바이트의 크기를 설정하고, 시작 주소는 0x0 으로 설정했습니다. 이는 GDT(Global Descriptor Table)을 생성할 때 사용한 논리와 같습니다.

인터럽트 처리 함수의 선언

- init_intdesc(): 이 함수는 IDT의 각 엔트리를 초기화하는 역할을 합니다. 각 인터럽트에 대한 처리 함수를 IDT에 등록할 것입니다.
- idt_ignore(): 이 함수는 인터럽트를 무시할 때 호출됩니다.
- idt_timer(): 이 함수는 타이머 인터럽트가 발생했을 때 호출되며, 시간 관련 처리를 담당합니다.
- idt_keyboard(): 이 함수는 키보드 인터럽트가 발생했을 때 호출되며, 키보드 입력 처리를 담당합니다.

```
void init_intdesc()
{
    int i, j;
    unsigned int ptr;
    unsigned short *isr;

    { // 0x00 : isr_ignore
        ptr = (unsigned int)idt_ignore;
        inttable[0].selector = (unsigned short)0x08;
        inttable[0].type = (unsigned short)0x8E00;
        inttable[0].offsetl = (unsigned short)(ptr & 0xFFFF);
        inttable[0].offseth = (unsigned short)(ptr >> 16);
    }

    {  // 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);
    }
}
```

```
void init_intdesc()
{
    unsigned int ptr;
```

- 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.

```
    // 0x00 : isr_ignore
    ptr = (unsigned int)idt_ignore;
```

- ptr = (unsigned int)idt_ignore;: idt_ignore 함수의 주소를 ptr 변수에 저장합니다. 여기서 idt_ignore는 특정 인터럽트에 대해 아무런 동작도 하지 않는 무시 처리 함수입니다.

```
    inttable[0].selector = (unsigned short)0x08;
```

- inttable[0].selector = (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;
```

- inttable[0].type = (unsigned short)0x8E00;: type 필드에는 인터럽트 게이트의 속성을 나타내는 값인 0x8E00 을 저장합니다. 0x8E는 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);
    inttable[0].offseth = (unsigned short)(ptr >> 16);
```

- inttable[0].offsetl = (unsigned short)(ptr & 0xFFFF);: ptr의 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);: ptr의 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.

동일한 로직으로 isr_timer와 isr_keyboard에 대해서도 설정을 합니다. 각각의 인터럽트 핸들러에 대해 selector, type, offsetl, offseth 값을 설정함으로써, 해당 인터럽트가 발생했을 때 처리할 함수의 주소와 인터럽트의 속성을 IDT에 등록하게 됩니다.
이렇게 설정된 IDT는 CPU가 인터럽트를 받았을 때, 어떤 함수를 실행해야 하는지를 알게 해주며, 시스템의 인터럽트 처리 메커니즘에 필수적인 역할을 합니다.

```
for (i = 0; i < 256; i++)
{
    isr = (unsigned short*)(0x0 + i * 8);
    *isr = inttable[0].offsetl;
    *(isr + 1) = inttable[0].selector;
    *(isr + 2) = inttable[0].type;
    *(isr + 3) = inttable[0].offseth;
}

{
    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;
}

{
    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 코드의 for 루프는 인터럽트 디스크립터 테이블(IDT)의 모든 엔트리를 초기화하는 데 사용됩니다. IDT는 CPU가 인터럽트 요청을 받았을 때 실행할 인터럽트 서비스 루틴(ISR)의 주소를 담고 있는 테이블입니다.

위 for 루프에서 수행되는 작업은 다음과 같습니다:

- for (i = 0; i < 256; i++): 이 루프는 0 부터 255 까지 256 번 반복합니다. 이는 IDT가 256 개의 인터럽트 벡터를 가질 수 있음을 의미하며, 각각의 인터럽트 벡터에 대한 엔트리를 설정할 것입니다.
- isr = (unsigned short*)(0x0 + i * 8);: 여기서 isr 포인터는 IDT 엔트리의 시작 주소를 가리키게 됩니다. 0x0 은 IDT의 시작 주소이며, i * 8 는 각 IDT 엔트리가 8 바이트 크기를 가지므로, 다음 인터럽트 엔트리 주소를 계산하기 위해 사용됩니다.
- *isr = inttable[0].offsetl;: IDT 엔트리의 첫 16 비트에 ISR의 하위 16 비트 주소를 저장합니다.

- *(isr + 1) = inttable[0].selector;: 다음 16 비트에는 세그먼트 셀렉터를 저장합니다. 이 값은 보통 코드 세그먼트를 가리키는 GDT(Global Descriptor Table)의 인덱스입니다.
- *(isr + 2) = inttable[0].type;: 그 다음 16 비트에는 게이트 타입과 속성을 저장합니다. 이 예에서는 0x8E00 이 사용되었는데, 이는 32 비트 인터럽트 게이트를 나타내며, DPL(Descriptor Privilege Level)이 0 이고, 세그먼트가 present 상태임을 나타냅니다.
- *(isr + 3) = inttable[0].offseth;: 마지막 16 비트에는 ISR의 상위 16 비트 주소를 저장합니다.

전체적으로 이 루프는 모든 인터럽트 벡터를 idt_ignore 함수로 초기화합니다. 즉, 어떠한 인터럽트가 발생하더라도 기본적으로 idt_ignore 함수가 호출되게 합니다. 그런 다음 특정 인터럽트 벡터에 대한 ISR을 별도로 설정할 수 있으며, 이 코드에서는 타이머 인터럽트(0x20)와 키보드 인터럽트(0x21)에 대해 별도의 ISR을 등록합니다.

인터럽트 디스크립터 테이블(IDT)의 설정을 완료한 후, 이제 CPU에게 새로운 IDT가 준비되었음을 알리는 단계에 대해 설명하겠습니다.

```
__asm__ __volatile__("mov eax, %0"::"r"(&idtr));
__asm__ __volatile__("lidt [eax]");
__asm__ __volatile__("mov al, 0xFC");
__asm__ __volatile__("out 0x21, al");
__asm__ __volatile__("sti");

return;
```

IDT 레지스터 설정

```
__asm__ __volatile__("mov eax, %0"::"r"(&idtr));
```

mov eax, %0: Inline 어셈블리를 사용하여, idtr의 주소를 EAX 레지스터에 로드합니다. %0 은 인라인 어셈블리에서 첫 번째 입력으로 제공된 &idtr을 나타냅니다.

```
__asm__ __volatile__("lidt [eax]");
```

lidt [eax]: LIDT 명령어를 사용하여, IDTR(Interrupt Descriptor Table Register)에 EAX 레지스터에 저장된 주소값을 로드합니다. 이 작업으로 IDT가 CPU에 등록됩니다.

인터럽트 마스킹 해제

```
__asm__ __volatile__("mov al, 0xFC");
```

mov al, 0xFC: AL 레지스터에 0xFC 값을 로드합니다. 이 값은 PIC(Programmable Interrupt Controller)의 IMR(Interrupt Mask Register)를 설정하기 위한 값으로, 특정 인터럽트를 활성화하기 위해 사용됩니다.

```
__asm__ __volatile__("out 0x21, al");
```

out 0x21, al: OUT 명령어를 사용하여, AL 레지스터의 값을 포트 0x21 에 보냅니다. 이 포트는 PIC의 IMR에 해당하며, 이를 통해 마스크되었던 인터럽트를 해제하게 됩니다.

인터럽트 활성화

```
__asm__ __volatile__("sti");
```

sti: STI 명령어(Interrupts Enable)는 CPU의 인터럽트를 활성화합니다. 이 명령어가 실행되면, CPU는 외부 인터럽트를 받아들일 준비가 되었음을 의미합니다.

함수 종료

```
return;
```

return;: 이 함수의 실행이 끝났음을 나타내며, 제어를 호출한 부분으로 반환합니다.

위 과정을 통해 시스템은 새로 설정한 IDT를 사용하여 인터럽트를 처리할 준비가 됩니다. 이러한 설정은 운영체제 또는 부트로더 수준에서 매우 중요한 단계로, 실제 하드웨어와 상호작용하는 깊은 수준의 컨트롤을 가능하게 합니다.

하지만 코드를 이렇게 작성하면 gcc 컴파일러가 알아먹지 못합니다. 왜냐하면 지금까지 써온 어셈블리어의 문법이 intel 문법인데 gcc의 기본 세팅은 AT&T 문법을 따르기 때문입니다.
따라서 intel 문법을 따르면서 gcc로 컴파일하고 싶다면 -masm=intel이라는 컴파일 옵션을 넣어줘야합니다.

Makefile에서 -masm=intel 옵션을 목적파일(.o)을 생성하는 코드에 넣어주고, interrupt.o를 생성하는 코드도 추가합니다.

```
CC = gcc

final.img : Boot_c.img Sector2_c.img disk.img
    cat Boot_c.img Sector2_c.img disk.img > final.img

disk.img : main.img
    objcopy -O binary main.img disk.img

main.img : main.o function.o interrupt.o
    ld -melf_i386 -Ttext 0x10200 -nostdlib main.o function.o interrupt.o -o main.img

main.o : main.c
    gcc -c -masm=intel -m32 -ffreestanding main.c -o main.o

function.o : function.c
    gcc -c -masm=intel -m32 -ffreestanding function.c -o function.o

interrupt.o : interrupt.c
    gcc -c -masm=intel -m32 -ffreestanding interrupt.c -o interrupt.o

Boot_c.img : Boot_c.asm
    nasm -f bin -o Boot_c.img Boot_c.asm

Sector2_c.img : Sector2_c.asm
    nasm -f bin -o Sector2_c.img Sector2_c.asm

clean :
    rm *.o *.img
```

- ISR 구현

인터럽트 발생시 실행할 함수를 구현해봅니다.

먼저 idt_ignore() 함수를 작성해봅시다.

```
void idt_ignore()
{
    __asm__ __volatile__
    (
        "push gs;"
        "push fs;"
        "push es;"
        "push ds;"
        "pushad;"
        "pushfd;"
        "mov al, 0x20;"
        "out 0x20, al;"
    );

    kprintf("idt_ignore", 5, 40);

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

```
void idt_ignore()
```

이 함수는 인터럽트를 무시하는 함수입니다. 여기서 void는 이 함수가 값을 반환하지 않음을 의미합니다.

```
    __asm__ __volatile__
```

이 부분은 어셈블리어 코드를 C 코드 내에 삽입하기 위한 일종의 매크로입니다. volatile는 컴파일러 최적화로 인해 해당 명령이 재정렬되거나 제거되는 것을 방지합니다.

```
        "push gs;"
```

gs는 segment register 중 하나로, 주로 사용자 모드 데이터에 접근할 때 사용됩니다. 여기서는 현재 gs register의 값을 스택에 저장(push)합니다.

```
        "push fs;"
```

fs는 segment register 중 하나로, 일반적으로 특정 시스템 데이터에 접근할 때 사용됩니다. 현재 fs register의 값을 스택에 저장합니다.

```
"push es;"
```

es는 segment register 중 하나로, 주로 데이터 세그먼트에 대한 추가적인 접근을 위해 사용됩니다. 현재 es register의 값을 스택에 저장합니다.

```
"push ds;"
```

ds는 segment register 중 하나로, 데이터 세그먼트에 대한 접근을 위해 사용됩니다. 현재 ds register의 값을 스택에 저장합니다.

```
"pushad;"
```

"pushad;" 명령은 x86 아키텍처에서 사용하는 어셈블리어 명령입니다. 이 명령은 모든 일반 목적 레지스터(general purpose register)의 값을 스택에 저장(push)하는 역할을 합니다.

일반 목적 레지스터(general purpose register)란 CPU에서 데이터를 임시로 저장하고 연산을 수행하는데 사용하는 레지스터를 말합니다. x86 아키텍처에서는 이러한 레지스터로 EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP 등이 있습니다.

따라서 "pushad;" 명령은 이러한 일반 목적 레지스터들의 현재 값을 모두 스택에 저장합니다. 이렇게 하면 함수나 인터럽트 처리 루틴 등이 레지스터 값을 변경하더라도, 나중에 원래의 레지스터 값으로 복원할 수 있습니다. 이는 프로그램의 실행 흐름을 정확하게 유지하는 데 중요한 역할을 합니다.

```
"pushfd;"
```

"pushfd;"는 x86 아키텍처의 어셈블리어 명령입니다. 이 명령은 현재 플래그 레지스터(flag register)의 값을 스택에 저장(push)하는 역할을 합니다.

플래그 레지스터(flag register)는 CPU의 상태를 나타내는 레지스터입니다. 플래그 레지스터에는 CPU의 각종 상태 정보가 비트 단위로 저장되어 있으며, 이 정보를 통해 CPU는 연산 결과가 양수인지, 음수인지, 0 인지, 오버플로우가 발생했는지 등을 판단합니다.

플래그 레지스터의 예로는 Carry Flag, Zero Flag, Sign Flag, Overflow Flag 등이 있습니다. 이 중 Carry Flag는 연산 결과에서 발생한 캐리(자릿수 올림)를 나타내고, Zero Flag는 연산 결과가 0 인지를 나타냅니다. Sign Flag는 연산 결과가 음수인지를 나타내고, Overflow Flag는 연산 결과에서 발생한 오버플로우를 나타냅니다.

따라서 "pushfd;" 명령은 이러한 플래그 레지스터의 현재 값을 스택에 저장합니다. 이렇게 하면 함수나 인터럽트 처리 루틴 등이 레지스터 값을 변경하더라도, 나중에 원래의 레지스터 값으로 복원할 수 있습니다. 이는 프로그램의 실행 흐름을 정확하게 유지하는 데 중요한 역할을 합니다.

```
"mov al, 0x20;"
```

이 명령은 AL 레지스터에 16 진수 값 0x20 을 저장하는 명령입니다. 여기서 AL 레지스터는 x86 아키텍처의 일반 목적 레지스터 중 하나인 EAX 레지스터의 하위 8 비트를 나타냅니다. 따라서 이 명령은 AL 레지스터에 0x20 이라는 값을 저장하라는 의미입니다.

```
"out 0x20, al;"
```

이 명령은 AL 레지스터의 값을 포트 0x20 에 출력하는 명령입니다. 여기서 '포트'는 컴퓨터의 하드웨어 장치와

CPU가 서로 통신하는 인터페이스를 말합니다. 특히 포트 0x20 은 PIC(Programmable Interrupt Controller)에 연결되어 있습니다. 따라서 이 명령은 PIC에게 앞서 발생한 인터럽트 처리가 완료되었음을 알리는 의미입니다.

"mov al, 0x20"과 "out 0x20, al"을 통해, CPU는 하드웨어 장치로부터 발생한 인터럽트를 처리하고, 그 처리가 끝났음을 해당 장치에 알릴 수 있습니다. 이는 컴퓨터 시스템에서 중앙 처리 장치(CPU)와 주변 장치들 사이의 복잡한 상호작용을 가능하게 합니다.

al에 0x20 을 저장하고 0x20 에 al의 값을 출력하는게 대체 무슨 소리인가?

- "mov al, 0x20;" : 이 부분은 AL 레지스터에 16 진수 값인 0x20 을 저장하는 명령입니다.
- "out 0x20, al;" : 이 부분은 앞서 AL 레지스터에 저장된 값을 I/O 포트 번호 0x20 에 출력하는 명령입니다.

첫 번째 명령에서는 AL 레지스터에 값을 '저장'하는 것입니다. AL 레지스터는 CPU의 일반 목적 레지스터 중 하나로, 여기에서는 특정 명령을 나타내는 0x20 이라는 값을 저장합니다.
두 번째 명령에서는 이 AL 레지스터의 값을 I/O 포트 0x20 에 '출력'하는 것입니다. 이는 인터럽트가 끝났음을 인터럽트 컨트롤러에게 알리는 명령이며, 여기서의 포트 0x20 은 인터럽트 컨트롤러와 연결된 I/O 포트를 가리킵니다.
따라서 이 두 명령은, "AL 레지스터에 0x20 값을 저장하고, 그 값을 인터럽트 컨트롤러와 연결된 포트 0x20 에 출력하여 인터럽트 처리가 끝났음을 알리는" 과정을 나타냅니다.

```
    kprintf("idt_ignore", 5, 40);
```

kprintf 함수를 호출하여 화면에 "idt_ignore"라는 문자열을 출력합니다.

```
        "popfd;"
```

스택에서 값을 꺼내 flag register에 복원합니다.

```
        "popad;"
```

스택에서 값을 꺼내 general purpose register에 복원합니다.

```
        "pop ds;"
```

스택에서 값을 꺼내 ds register에 복원합니다.

```
        "pop es;"
```

스택에서 값을 꺼내 es register에 복원합니다.

```
        "pop fs;"
```

스택에서 값을 꺼내 fs register에 복원합니다.

```
        "pop gs;"
```

스택에서 값을 꺼내 gs register에 복원합니다.

```
        "leave;"
```

함수의 종료 준비를 합니다. 스택 프레임을 해제하는 역할을 합니다.

```
        "nop;"
```

아무런 연산도 수행하지 않는 no operation 명령을 수행합니다. 프로그램의 실행을 잠시 멈추거나 디버깅 용도로 사용됩니다.

```
        "iretd;"
```

인터럽트 반환 명령을 수행합니다. 이는 CPU의 상태를 인터럽트가 발생하기 전 상태로 복원하고 인터럽트 루틴을 종료합니다.

이 함수는 인터럽트가 발생했을 때 아무런 동작을 수행하지 않고 인터럽트를 무시하도록 만들어진 함수입니다. 이러한 함수는 시스템의 안정성을 유지하고 예기치 않은 인터럽트로부터 시스템을 보호하는 역할을 합니다.

같은 방법으로 타이머와 키보드 인터럽트 또한 똑같이 구현해봅시다.

그 전에 interrupt.c 맨 상단에 전역변수로 다음과 같이 만들어줍니다.

```
unsigned char keyt[2] = {'A', 0};
unsigned char key[2] = {'A', 0};
```

```
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;"
        "in al, 0x60;"
        "mov al, 0x20;"
        "out 0x20, al;"
    );

    kprintf(key, 8, 40);
    key[0]++;

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

타이머와 키보드 모두 입력이 올 때마다 keyt, key를 각각 해당 위치에 출력을 하고 하나를 더해줍니다.
타이머는 시도때도 없이 인터럽트가 발생할 것이니 알 수 없는 문자들이 바뀌어 가며 찍혀 나갈 것이고, 키보드는 타이핑을 할 때만 문자가 찍혀 나올 것입니다.

최종적인 코드는 다음과 같습니다.

```
#include "interrupt.h"
#include "function.h"

struct IDT inttable[3]; // ignore, timer, keyboard
struct IDTR idtr = {256 * 8 - 1, 0};

unsigned char keyt[2] = {'A', 0};
unsigned char key[2] = {'A', 0};

void init_intdesc()
{
    int i, j;
    unsigned int ptr;
```

```
unsigned short *isr;

{ // 0x00 : isr_ignore
    ptr = (unsigned int)idt_ignore;
    inttable[0].selector = (unsigned short)0x08;
    inttable[0].type = (unsigned short)0x8E00;
    inttable[0].offsetl = (unsigned short)(ptr & 0xFFFF);
    inttable[0].offseth = (unsigned short)(ptr >> 16);
}

{  // 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);
}

for (i = 0; i < 256; i++)
{
    isr = (unsigned short*)(0x0 + i * 8);
    *isr = inttable[0].offsetl;
    *(isr + 1) = inttable[0].selector;
    *(isr + 2) = inttable[0].type;
    *(isr + 3) = inttable[0].offseth;
}

{
    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;
}
```

```
    {
        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;
    }

    __asm__ __volatile__("mov eax, %0"::"r"(&idtr));
    __asm__ __volatile__("lidt [eax]");
    __asm__ __volatile__("mov al, 0xFC");
    __asm__ __volatile__("out 0x21, al");
    __asm__ __volatile__("sti");

    return;
}

void idt_ignore()
{
    __asm__ __volatile__
    (
        "push gs;"
        "push fs;"
        "push es;"
        "push ds;"
        "pushad;"
        "pushfd;"
        "mov al, 0x20;"
        "out 0x20, al;"
    );

    kprintf("idt_ignore", 5, 40);

    __asm__ __volatile__
    (
        "popfd;"
        "popad;"
        "pop ds;"
        "pop es;"
        "pop fs;"
        "pop gs;"
        "leave;"
```

```
        "nop;"
        "iretd;"
    );
}

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;"
        "in al, 0x60;"
        "mov al, 0x20;"
        "out 0x20, al;"
    );

    kprintf(key, 8, 40);
    key[0]++;

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

이제 컴파일을 하고 가상머신에 돌려봅니다.

```
minsung@ubuntu:~/Dev/Os/함수 만들기/IDT$ 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
```

결과를 보면 알 수 없는 254 개 중 하나의 ignore 인터럽트가 걸렸고 타이머 인터럽트는 계속 걸려서 문자가 끊임없이 바뀌어서 출력됩니다. 키보드는 누를 때마다 문자가 바뀌어 출력됩니다.

이렇게 해서 C언어로 인터럽트를 구현해봤습니다.

2. 키보드 드라이버 개발

- 키보드 드라이버 1

위에서 키보드 인터럽트를 받을 수 있게 되었으니, 이제 어떤 키를 눌렀는지를 알아봅시다.

Interrupt.c에 들어가서 버퍼용 전역변수를 하나 선언합니다. 여기에 어떤 문자를 입력했는지 저장할 겁니다.

```
unsigned char keybuf;
```

그리고, init_intdesc() 함수 끝에 다음과 같은 어셈블리 코드를 추가합니다. 이는 키보드 입력을 받겠다는 일종의 선업입니다.

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

주의할 점은 인터럽트 작동 시작 전에 작성해야 한다는 점입니다. 왜냐하먄 키보드 입력 받는다고 선언을 안했는데 인터럽트를 시작하면 안되기 때문입니다.

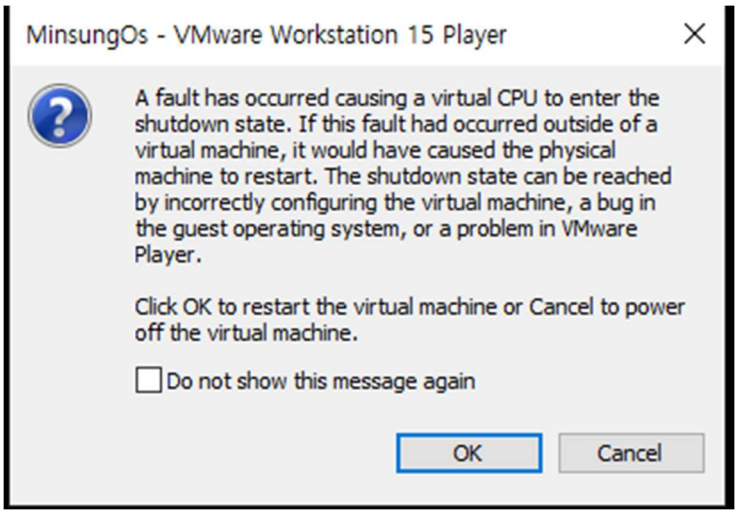
이제 세팅이 끝났으니 본격적으로 idt_ketboard() 함수를 수정해서 키보드를 눌렀을 때 어떤 키를 눌렀는지 파악해보도록 하겠습니다.

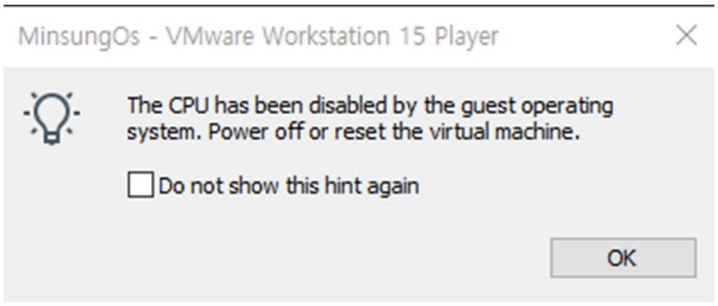
```
void idt_keyboard()
{
    __asm__ __volatile__
    (
        "push gs;"
        "push fs;"
        "push es;"
        "push ds;"
        "pushad;"
        "pushfd;"
        "xor al,al;"
        "in al, 0x60;"
    );

    __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;"
    );
}
```

위의 idt_keyboard() 함수와의 차이는 __asm__ __volatile__("mov %0, al;" :"=r"(keybuf)); 을 통해 실제로 어떤 값이 al에 저장되어있는지 keybuf로 옮기는 것과 kprintf 로 어떤 값인지 확인하는 과정뿐입니다.

이제 컴파일 후 가상머신에 올려보겠습니다.

```
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
```





오류가 발생하여 차주에 다시 살펴보도록 하겠습니다.