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. C언어로 인터럽트 핸들러 개발 2. 오류 수정

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. C언어로 인터럽트 핸들러 개발

인터럽트 핸들러 개발, 키보드 드라이버 개발에서는 CPU 가상화 설정을 변경해도, 코드의 수정을 해봐도 작주에 발생하였던 오류가 고쳐지지 않아서 개념 공부와 C언어로 간단한 인터럽트 핸들러를 개발해 보았습니다.

- 인터럽트 개념

인터럽트: CPU에게 시스템 내에서 중요한 사건이 발생했음을 알리는 메커니즘.
이 중요한 사건은 입출력 요청, 하드웨어 오류, 사용자 입력 등 다양할 수 있음. 인터럽트가 발생하면 CPU는 현재 처리 중인 작업을 일시 중단하고, 대신 인터럽트 서비스 루틴(ISR)이라는 특별한 코드를 실행하도록 요청함.

1. 인터럽트 핸들러 (Interrupt Handler) || 인터럽트 서비스 루틴 (ISR): 인터럽트가 발생했을 때 CPU가 호출하여 실행하는 코드.
인터럽트 핸들러는 인터럽트가 발생한 원인을 파악하고 이를 적절히 처리하는 로직을 포함함. 인터럽트 핸들러가 작업을 마치면, CPU는 인터럽트 전에 중단된 작업을 계속 수행합니다.

2. 인터럽트 벡터 테이블 (Interrupt Vector Table): 특정 인터럽트에 대한 ISR의 주소를 저장하는 테이블. 인터럽트가 발생하면, CPU는 이 테이블을 참조하여 해당 인터럽트를 처리할 ISR을 찾아냅니다.

3. 프로그램 가능 인터럽트 컨트롤러 (Programmable Interrupt Controller, PIC): 하드웨어 인터럽트를 받아들이고 이를 CPU에 전달하는 역할을 하는 장치.
PIC는 여러 인터럽트 요청을 우선순위에 따라 정렬하여 CPU에 전달. 이는 CPU가 여러 인터럽트 중에서 가장 중요한 것을 먼저 처리할 수 있도록 도움.

4. 인터럽트 요청 (Interrupt Request, IRQ): 하드웨어 장치가 CPU에게 인터럽트를 요청하는 신호.
각 하드웨어 장치는 고유한 IRQ 번호를 가지며, 이 번호를 통해 CPU는 어떤 장치가 인터럽트를 발생시켰는지 파악할 수 있음.

5. 인터럽트 마스킹 (Interrupt Masking): 인터럽트 마스킹은 특정 인터럽트의 처리를 일시적으로 중단하는 기능. 인터럽트 마스킹을 사용하면, CPU는 마스킹된 인터럽트를 무시하고 다른 작업을 계속 수행할 수 있음.

- C언어로 간단한 인터럽트 핸들러 개발

```c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

// 인터럽트 핸들러 함수
void interrupt_handler(int signum) {
    printf("Interrupt signal received!\n");
    // 실제 인터럽트 핸들링 코드를 여기에 작성합니다.
}

int main() {
    // SIGINT(일반적으로 Ctrl+C로 발생시키는 시그널)에 대한 핸들러 설정
    signal(SIGINT, interrupt_handler);

    while(1) {
        printf("Running...\n");
        sleep(1); // 1초 동안 프로그램 실행을 잠시 멈춥니다.
    }

    return 0;
}
```

```
#include <stdio.h>
```

stdio.h는 표준 입력/출력 라이브러리를 포함하는 헤더 파일입니다. 이를 통해 printf와 같은 함수를 사용할 수 있습니다.

```
#include <signal.h>
```

signal.h는 시그널 처리 관련 함수와 매크로를 포함하는 헤더 파일입니다. 이를 통해 signal 함수와 'SIGINT'와 같은 시그널 매크로를 사용할 수 있습니다.

시그널: 운영체제에서 프로세스나 스레드에게 어떤 이벤트가 발생했음을 알리는 방법 중 하나
시그널은 비동기적으로 동작하며, 특정 시그널이 전달되면 운영체제는 해당 프로세스가 미리 등록해둔 핸들러 함수를 호출하여 시그널을 처리합니다.
시그널은 다양한 종류가 있으며, 각각이 특정 이벤트를 나타냅니다. 예를 들어, 'SIGINT'는 인터럽트 시그널을 나타내며, 사용자가 Ctrl+C를 눌렀을 때 발생합니다. 'SIGALRM'은 알람 시그널을 나타내며, alarm 함수에 의해 설정된 시간이 경과했을 때 발생합니다. 'SIGSEGV'는 세그멘테이션 오류 시그널을 나타내며, 프로세스가 메모리를 잘못 접근했을 때 발생합니다.
프로세스는 signal 함수를 사용하여 각 시그널에 대한 핸들러를 설정할 수 있습니다. 이 핸들러는 해당 시그널이 발생했을 때 운영체제에 의해 자동으로 호출됩니다. 시그널 핸들러 내에서는 시그널의 원인을 파악하고 적절히 대응하는 코드를 작성할 수 있습니다.
시그널 메커니즘은 프로세스 간 통신(IPC), 예외 처리, 프로세스 제어 등 다양한 용도로 사용됩니다.

```
#include <unistd.h>
```

unistd.h는 POSIX 운영체제 API를 포함하는 헤더 파일입니다. 이를 통해 sleep 함수와 같은 POSIX 함수를 사용할 수 있습니다.

POSIX:
POSIX는 Portable Operating System Interface의 약자로, UNIX 운영 체제의 API를 표준화한 것입니다. POSIX 표준은 UNIX 뿐만 아니라 다른 운영 체제들에서도 널리 사용되며, 이를 따르는 운영 체제들은 서로 호환성을 유지할 수 있습니다.
POSIX 표준에는 시스템 호출, 라이브러리 함수, 셸 유틸리티, 명령어 구문 등 다양한 요소가 포함되어 있습니다. 이 중 시스템 호출과 라이브러리 함수는 C 언어 API의 형태로 제공되며, 이를 통해 프로그래머들은 파일 시스템, 프로세스, 스레드, 시그널, 타이머 등의 운영 체제 기능을 사용할 수 있습니다.
unistd.h 헤더 파일은 POSIX 표준에 정의된 여러 함수와 매크로를 포함하고 있습니다. 예를 들어, sleep 함수는 프로그램을 지정된 시간 동안 잠시 멈추게 하는 함수로, 이는 unistd.h 헤더 파일에 정의되어 있습니다. 이 외에도 fork, pipe, read, write 등의 함수도 unistd.h에 포함되어 있습니다. 이러한 함수들은 UNIX 계열 운영 체제 뿐만 아니라 POSIX 표준을 따르는 대부분의 운영 체제에서 사용할 수 있습니다.

```
void interrupt_handler(int signum) { … }
```

이 부분은 'SIGINT' 시그널을 처리하기 위한 인터럽트 핸들러 함수를 정의하는 부분입니다. 이 함수는 'SIGINT' 시그널이 발생하면 자동으로 호출됩니다.

```
printf("Interrupt signal received!\n");
```

'SIGINT' 시그널이 발생하면 인터럽트 핸들러 함수 내에서 이 코드가 실행되어 "Interrupt signal received!"라는

메시지를 출력합니다.

```
int main() { … }
```

main 함수는 프로그램의 시작점입니다. C 프로그램은 항상 main 함수에서 시작하여 main 함수에서 종료합니다.

```
signal(SIGINT, interrupt_handler);
```

signal 함수는 특정 시그널에 대한 핸들러를 설정하는 함수입니다. 이 코드는 'SIGINT' 시그널에 대한 핸들러로 interrupt_handler 함수를 설정합니다.

```
while(1) { … }
```

이 부분은 무한 루프를 생성합니다. 이 루프 내에서는 계속해서 "Running..."이라는 메시지를 출력하고 1 초 동안 대기합니다.

```
printf("Running...\n");
```

이 코드는 "Running..."이라는 메시지를 출력합니다.

```
sleep(1); // 1초 동안 프로그램 실행을 잠시 멈춥니다.
```

sleep 함수는 프로그램을 지정된 시간(초 단위) 동안 잠시 멈추게 하는 함수입니다. 이 코드는 프로그램을 1 초 동안 잠시 멈춥니다.

```
return 0;
```

main 함수가 0 을 반환하면 프로그램은 성공적으로 종료되었음을 운영체제에 알립니다. 이 코드는 main 함수의 마지막 부분에 위치합니다.

이 코드를 실행하면, 프로그램은 무한히 "Running..."이라는 메시지를 출력하면서 실행됩니다. 이 상태에서 사용자가 Ctrl+C를 누르면, 'SIGINT' 시그널이 발생하고 이에 대응하는 인터럽트 핸들러인 interrupt_handler 함수가 호출됩니다. 이 때 "Interrupt signal received!"라는 메시지가 출력됩니다. 이는 인터럽트가 프로그램의 일반적인 실행 흐름을 중단하고 특별한 작업을 수행하도록 만드는 인터럽트 메커니즘을 시뮬레이션한 것입니다.

2. 오류 수정

쉘 개발, 하드디스크 드라이버 개발 등 다음으로 넘어가려고 하였는데 뒷 내용에도 인터럽트 핸들러가 쓰여 오류를 찾아서 고쳐야겠다고 결심하였습니다. 그래서 인터럽트 구현부터 다시 살펴 보았습니다.

interrupt.c 코드에서 `idt_ignore` 함수에서 `kprintf("idt_ignore", 5, 40);` 부분은 문자열 "idt_ignore"을 화면의 좌표 (5, 40)에 출력하려는 의도를 가지고 있습니다. 그러나 원본 코드에서는 해당 좌표를 직접 사용하는 대신에 메모리 주소를 계산하여 선형 주소로 변환하는 것이 빠르고 효율적인 방법입니다.

그러나 코드의 수정 부분에서는 메모리 주소 계산이 누락되어 수정되었습니다. 따라서 아래와 같이 선형 주소를 계산하고 문자열을 해당 위치에 출력하는 코드로 변경되었습니다.

```
// 화면의 가로 길이를 80으로 가정합니다.
int line = 5;
int col = 40;

// 좌표를 선형 주소로 변환합니다.
char *video = (char*)(0xB8000 + 2 * (line * 80 + col));

// 문자열을 출력합니다.
for (int i = 0; "idt_ignore"[i] != 0; i++) {
    *video++ = "idt_ignore"[i];
    *video++ = 0x03; // 문자 색상으로 0x03을 사용한다고 가정합니다.
}
```

코드를 변경한 이유:

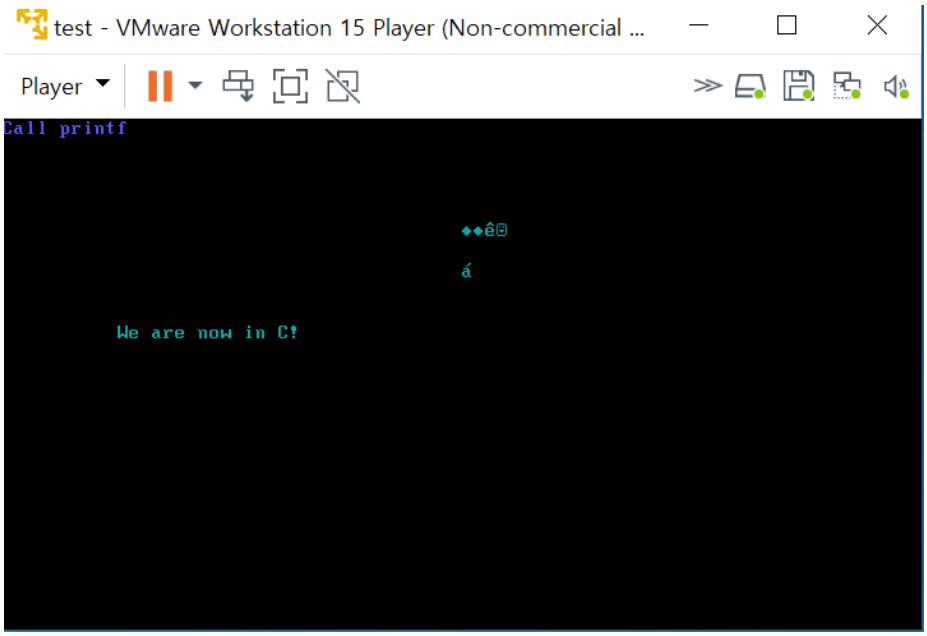
1. 문제 해결: 기존에 주어진 코드에서 kprintf 함수를 사용했을 때 올바르게 작동하지 않았습니다. 따라서 이를 해결하기 위해 직접 화면 좌표를 계산하고 문자열을 출력하는 방식으로 코드를 변경하였습니다.

2. 간소화: kprintf 함수를 사용하지 않고 코드를 직접 작성함으로써 코드를 더 간단하게 만들 수 있었습니다. 직접 메모리 주소를 계산하여 문자열을 출력하는 방식은 더 직관적이고 간단합니다.

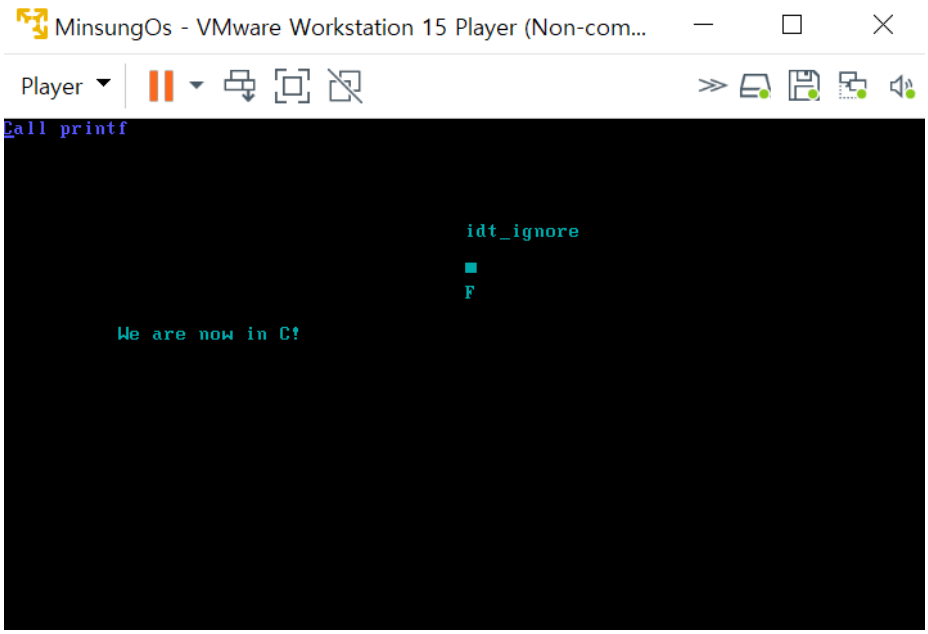
3. 코드 의도 명확화: kprintf 함수를 사용하는 것보다 직접 코드를 작성함으로써 출력하는 문자열의 위치와 색상을 더 명확하게 제어할 수 있습니다.

이 변경된 코드에서는 화면의 가로 길이를 80 으로 가정하고, 주어진 좌표 (5, 40)를 선형 주소로 변환하여 해당 위치에 문자열을 출력합니다. 또한, 문자의 색상을 0x03 으로 설정하여 문자가 출력될 때 색상을 지정하였습니다.

이렇게 수정 후 컴파일을 하였더니 아래와 같이 이상한 문자가 뜨던 코드가



아래와 같이 올바르게 수정되었습니다.



즉 kprintf 함수가 컴파일 오류 등과 같은 문제로 인하여 작동하지 않는 것 같습니다. 그래서 kprintf 함수를 수정해야겠다고 판단하였습니다. 이후 키보드 인터럽트 오류를 수정할 때 kprintf 함수를 수정해야한다고 판단했습니다.