Week 3 과제 답안

1. **리눅스 운영체제는 여러 배포판이 있습니다. 이 중에서 가장 널리 사용되는 Debian GNU, Ubuntu, CentOS, RedHat 4가지의 차이점을 정리하여 작성해주세요.**
   - 각 배포판별 장점과, 주요 사용처를 포함하여 작성해주세요.

(답안)

Debian GNU - 데비안 프로젝트가 개발한 오픈소스 리눅스 운영체제이다. 데비안은 패키지 설치 및 업그레이드를 단순하게 관리하기 위한 용도로 개발되었다. 패키지 매니저인 apt 를 사용하면 다른 소프트웨어 설치 및 다른 소프트웨어와의 의존성 확인, 보안 업데이트등을 자동으로 설정해준다. 데비안은 안정성과 보안을 중점에 두어 개발되고 있다.

Ubuntu - 영국 기업 캐노니컬이 개발, 배포하는 Debian 리눅스 기반 운영체제이다. 사용자의 편의성에 초점을 둔 리눅스 배포판이다. 설치 및 사용이 용이하며 현재 개인 리눅스 사용자들이 가장 많이 사용하는 배포판이다. 일반 버젼과 장기 지원 버젼이 따로 존재하며 주로 LTS(Long Term Support) 버젼을 사용한다.

CentOS - 센트 OS 프로젝트에서 레드햇 제휴로 개발한 리눅스 운영체제 배포판이다. RedHat Enterprise Linux(이하 RHEL)과 완벽하게 호환되는 무료 기업용 컴퓨팅 플랫폼을 제공할 목적으로 개발되었다. 개인보다는 기업에서 더 많이 사용한다.

RedHat - 레드햇 재단에서 개발한 리눅스 운영체제이다. 코드 자체는 레드햇 사이트에 무료 공개가 되어있지만, 배포판 자체는 유료로 사용해야 한다. 요금은 subscription 형태로 운영되며 기술 지원은 구매 이후 7년동안 제공된다. 유료로 서비스 되는 만큼, 기업용 메인 프레임 서버, 슈퍼 컴퓨터 등에 많이 사용되며 버젼 5 이후로는 데스크톱 / 베이스 서버 / 고급 플랫폼 용도로 버전이 나뉘어 있다.

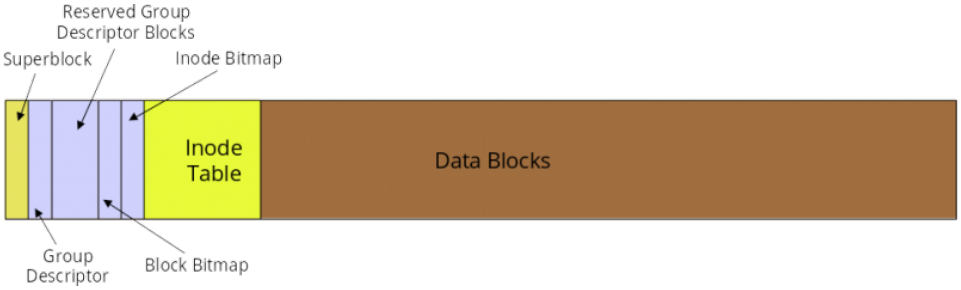
2. **리눅스 파일 시스템(ext file system)에 대해서 정리한 내용을 작성해주세요.**
   - inode에 대한 설명은 반드시 포함하여 주세요.

(답안)

Extended File System(or ext)는 Linux kernel에 구현되어 있는 파일 시스템입니다. ext는 Remy Card에 의해 1992년에 만들어졌다. 기본적인 ext는 초창기에 등장한 MINIX file system(이하 MINIX)의 한계를 극복하기 위해 만들어졌다. MINIX는 교육용으로 만들어진 Unix-like한 운영체제이며 오픈소스이다. MINIX에는 다음과 같은 구조들을 포함하고 있다.

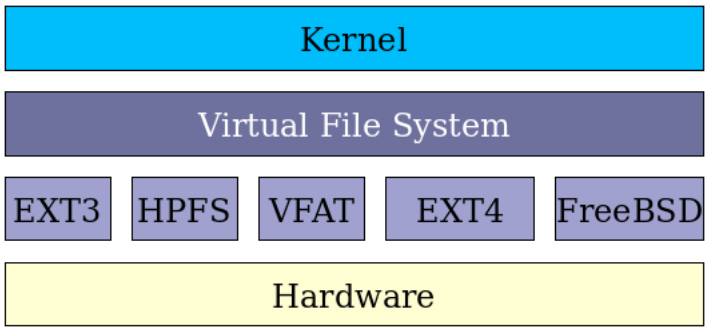
- Boot sector : 하드디스크의 첫번째 섹터이며, 컴퓨터의 구동(booting)에 필요한 기본적인 정보가 담겨있다. (섹터란, 하드디스크를 구역별로 나눠서 부르는 단위입니다.)
- superblock : 각 파티션의 첫번째 블록이다. 파일 시스템 구조에 대한 metadata를 포함하고 있으며, 이를 실제 파티션으로 구분된 영역과 물리적인 디스크를 연결한다. (파티션이란, 리눅스 운영체제에서 논리적으로 하드디스크를 구분하는 단위이다. 블록은 파티션내에서 파일을 관리하는 논리적 단위이다.)
- inode : index-node의 줄임말이며, 256B 크기로 실제 파일의 저장 정보를 담고 있는자료구조다.
- inode bitmap block : 현재 어떤 inode가 사용되고 있는지 알 수 있는 블록이다.
- data zone : 실제 데이터가 저장되어 있는 영역
- zone bitmap : 현재 어떤 data가 저장되어있는지 알 수 있는 bitmap.

아래는 ext filesystem의 구조이다.



ext는 이러한 MINIX의 구조는 계승하고, 한계를 극복하기 위해 개발되었다. MINIX는 파일 이름은 14글자만 가능하였고, 64MB의 저장공간만 지원했기 때문에 기본적인 성능 향상이 필요했다.

ext는 이를 극복하기 위해서 Virtual File System(VFS)를 사용했다.



(source: https://opensource.com/life/16/10/introduction-linux-filesystems)

VFS는 실제로 사용하는 파일 시스템에 관계없이 Linux Kernel이 하드웨어를 컨트롤할 수 있게끔 지원하는 구조를 말한다. VFS는 system call을 통해서 실제 파일 시스템들에게 명령을 내리며, 이때 device driver가 특정 파일 시스템과 VFS의 system call을 소통할 수 있게 도와준다. 이 부분이 우리가 컴퓨터에서 특정 기기를 사용할 때, device driver를 설치해야 하는 이유이다.

VFS를 통해 Linux는 파일이름을 255자까지 사용가능 지원하고, 최대 2GB의 저장 공간을 다룰 수 있게 되었다.

3. **다음은 실제로 리눅스 운영체제에서 시스템 콜을 구현한 어셈블리 코드 예제이다. 아래 코드를 해석하여, 어떤 프로그램 명령을 수행한 코드인지 서술해주세요.**
   - eax, ebx, ecx, edx 레지스터의 역할을 포함합니다.
   - 0x04, 0x01이 의미를 포함합니다.
   - int 0x80의 의미를 포함합니다.

```
mov     eax,    0x04
mov     ebx,    0x01
mov     ecx,    $buf
mov     edx,    14

int     0x80
```

(답안)

mov eax, 1
mov ebx, 0
int 0x80 // 소프트웨어 인터럽트 명령
<강의자료 발췌>

위의 예제는 eax 레지스터에 시스템 콜 번호(1번)을 넣고, ebx 레지스터에는 해당 시스템 콜의 인자를 넣고 int 0x80인 인터럽트를 수행하면서 시스템 콜이 수행되는 어셈블리 코드이다.

%eax	kernel function (system call)	%ebx
1	sys_exit (exit)	int
2	sys_fork (fork)	struct pt_regs
3	sys_read (read)	unsigned int
4	sys_write (write)	unsigned int
5	sys_open (open)	const char*

<강의자료 발췌>
(강의자료 발췌) 위의 테이블을 Interrupt Descriptor Table(IDT)라고 하는데, IDT는 해당 시스템 콜에 대한 정보를 모두 가지고 있다. 현재 1번 시스템콜을 호출하였고, 그 때 해당하는 integer value는 0이다.
이를 응용해서 위의 문제를 분석하면 아래와 같다.

1. 소프트웨어 인터럽트가 발생하면서 (시스템 콜 호출을 했기 때문에) 0x80값을 넘겨준다.
2. CPU는 사용자 모드에서 커널 모드로 바꿔준다.
3. IDT에서 0x80에 해당하는 주소를 찾아서 실행한다. 이 때 eax, ebx 값을 함께 넘겨준다.
4. 불러온 시스템 콜 함수의 eax 레지스터를 확인하여 시스템 콜 번호를 IDT에서 확인한다. 현재 0x04이므로 4번 시스템 콜인 sys_write를 호출한다.
5. sys_write 함수의 인자를 ebx 레지스터에서 확인한다. 0x01을 sys_write 함수의 인자로 넘겨준다.
6. sys_write 함수가 디스크에 write할 내용을 ecx 레지스터를 통해 확인한다. ecx 레지스터는 buffer를 저장하고 있다. buffer는 시스템 콜을 호출한 C언어로 구현된 프로그램 내에서 저장할 내용을 함께 부여받는다. (e.g. “Hello World\n”)
7. buffer의 내용중에 앞에서 부터 edx 레지스터에 할당된 크기만큼 write를 수행한다. edx에 저장된 값은 0x14이므로, 14bytes 만큼 buffer의 내용을 디스크에 저장한다.

4. **리눅스 운영체제에서 사용자가 CLI를 통해서 “gcc -o test test.c” 명령어를 실행했을 때 실제 내부에서 동작하는 방식을 상세하게 서술해주세요.**
   - 사용자가 커멘드창을 오픈하면, 자동으로 쉘프로그램이 실행됨\
   - gcc 명령은 주어진 C 파일을 컴파일 하는 명령어임
   - 해당 쉘에서는 키보드를 입력하면, 해당 키보드 문자가 화면에 표시하며, 엔터를 누르면 그동안 입력받은 문자열을 명령으로 인식하고, 해당 명령을 실행함

- 스케쥴링 방식은 선점형을 지원하며, 기본적으로 멀티 태스킹을 지원함
- 인터럽트가 발생하며 내부에서는 gcc 컴파일러가 수행됨
- gcc 컴파일러는 주어진 옵션으로 test.c에 해당하는 test.o를 생성함
- test.o 파일을 기준으로 실행파일인 test 파일을 생성함

(답안)

gcc는 GNU Compiler Collection의 약자로, 실제 gcc가 컴파일러는 아니고 엄밀하게 얘기하면 컴파일러 드라이버입니다. 컴파일러 드라이버는 실제 컴파일을 해주지 않고, 컴파일에 필요한 여러 프로그램들을 순차적으로 호출해주는 역할을 합니다. gcc를 이용한 컴파일 과정을 요약하면 다음과 같습니다.

전처리기 호출 -> C컴파일러 호출 -> 어셈블러호출 -> 링커 호출

1. 전처리기 호출
   - gcc는 C언어 코드 전처리기인 cpp0를 호출합니다. 전처리 단계에서 #include에 포함된 라이브러리(stdio.h)를 불러와 현재 컴파일 대상인 코드인 test.c에 붙여넣습니다. 그 결과 test.i가 생성됩니다.
2. C컴파일러 호출
   - gcc는 C컴파일러인 cc1을 호출합니다. cc1은 GNU C Compile 문법을 따르는 symbol table을 통해서 해당 C언어를 tokenizing -> lexical analysis -> syntax analysis -> semantic analysis를 거쳐 중간 표현 코드인 GIMPLE Tree를 생성합니다. GIMPLE Tree는 code optimizer와 code generator를 거쳐 최종 어셈블리 파일인 test.s로 생성됩니다.
3. 어셈블러 호출
   - gcc는 어셈블러인 as를 호출합니다. as는 test.s를 ELF(Executable and Linking) 바이너리 파일 구조 규약을 따르는 형태의 바이너리 코드인 오브젝트 파일 test.o로 변환합니다.
4. 링커호출
   - gcc는 링커인 collect2를 호출합니다. 링커는 해당 프로그램이 사용하고 있는 종속성을 가지는 라이브러리 정보들과 해당 코드를 연결해줍니다. 현재 사용되는 stdio.h의 내용이 이미 메모리에 있다면 따로 추가 연결을 해줄 필요가 없으며, 만약 메모리에 해당 내용이 없다면 stdio.h을 미리 컴파일 해둔 오브젝트 코드를 가져옵니다. stdio.h는 정적 연결 라이브러리(Static Linking Library)이기 때문에 미리 컴파일을 해두고 .lib 파일 형태로 미리 가지고 있습니다. 링킹을 하는 이유는 어셈블러가 생성한 test.o에는 printf와 같은 stdio.h에서 불러와서 사용하는 코드에 대한 내용이 없기 때문에, 실제 실행 파일을 생성할 때는 printf 함수에 대한 오브젝트 코드를 연결하여 실행 파일에서 문제없이 코드 순서대로 실행할 수 있게 해줍니다. 링커는 최종 실행 파일인 a.out을 생성합니다.

이렇게 gcc가 test.c를 a.out (기본 실행파일 이름)으로 생성하게 됩니다. 이 때 -o 옵션과 함께 문자열을 부여하면, 해당 문자열의 이름을 가지는 실행파일을 생성합니다. 즉, gcc -o test test.c 명령어는 test.c를 위의 과정을 모두 거쳐 최종 test라는 이름의 실행파일을 생성해달라는 의미가 됩니다.