주간 업무 사항 정리

작성자	제품팀 이민성 인턴
업무 일시	2023.12.26 – 2023.12.29

세부 사항

1. 업무 내역 요약 정리

목표 내역	Done & Plan	To-do
- 주제 선정 - 프로젝트 계획서 작성 - 프로젝트 진행	- 주제 선정: 운영체제 개발 1. CLI 기반 운영체제 개발 인공지능: 2. sns 데이터를 통해 특정 키워드나 주제에 개한 대중의 감정추이를 분석하고 시각화하여 트렌드 파악하는 시스템 개발 3. 딥러닝 모델을 사용하여 주어진 이미지에 대해 설명하는 캡션을 자동으로 생성하는 시스템 개발 4. 개, 고양이 이미지 구분하는 시스템 개발 데이터베이스: 5. 개인이나 소규모 사업체를 위한 재고 관리 시스템 개발 6. 회사 내부의 이슈 트래킹 시스템 개발 7. 실시간 데이터를 복제하는 시스템 개발 인공지능 + 데이터베이스: 8. 머신러닝을 이용한 주가 예측 시스템 개발	- 주제 선정 - 프로젝트를 위한 기본 개념 공부 1. 어셈블리어에 대해서 2. X86 아키텍처 3. 넷와이드 어셈블러(NASM) 4. 부트로더 5. 하드디스크 읽기 6. 리얼모드에서 보호모드 전환 7. 인터럽트 8. 키보드 입력 9. 기초적인 쉘 구현 10. 하드디스크 읽기/쓰기 기능 11. ext2 파일 시스템 12. 자주 사용되는 레지스터와 어셈블리 명령어 - 프로젝트 계획서 작성

2. 내용 세부 (업무 세부 내역 정리 및 기타 사항 정리)

1. 주제 선정

- 주제: CLI 기반 운영체제 개발

- 주제 선정 이유:
1. 컴퓨터 과학의 근본적 이해: 운영체제는 컴퓨터 시스템의 핵심 구성 요소로, 하드웨어 자원을 관리하고, 사용자와 하드웨어 사이의 인터페이스 역할을 합니다. CLI 기반 운영체제를 개발하면서 프로세스 관리, 메모리 관리, 파일 시스템 등의 기본적인 운영체제 개념을 실질적으로 이해하고 구현해 볼 수 있습니다. 이는 컴퓨터 과학의 근본적인 원리를 체계적으로 학습하는 데 큰 도움이 됩니다.

2. 시스템 레벨 프로그래밍 경험: 운영체제 개발은 하드웨어와 가까운 시스템 프로그래밍을 필요로 합니다. 이는 C, 어셈블리 언어와 같은 저수준 언어를 사용하여 하드웨어를 직접 제어하게 되므로, 컴퓨터 시스템에 대한 깊이 있는 이해를 기르는 데 도움이 됩니다. 이러한 경험은 프로그래밍 역량을 보다 견고하게 만들어줍니다.

3. 독특하고 도전적인 프로젝트: 현재 많은 개발자들이 딥러닝이나 데이터베이스를 활용한 프로젝트를 진행하고 있습니다. 반면, 운영체제를 직접 개발하는 것은 상대적으로 드물어, 독특하고 도전적인 프로젝트로서 개발자의 능력을 입증하는 데 좋은 기회가 될 수 있습니다.

4. 프로젝트 관리 경험: 운영체제 개발은 여러 컴포넌트를 개발하고 통합하는 복잡한 과정을 포함합니다. 이를 통해 프로젝트 관리와 협업에 대한 경험을 쌓을 수 있습니다.

따라서, CLI 기반 운영체제 개발은 컴퓨터 과학의 근본적 이해, 시스템 레벨 프로그래밍 능력 향상, 독특한 프로젝트 경험, 그리고 프로젝트 관리와 협업 역량 등을 갖추는 데 매우 유익한 프로젝트로 볼 수 있습니다. 딥러닝이나 데이터베이스를 활용한 프로젝트가 제공하는 경험과는 다른, 독특하고 근본적인 학습 기회를 제공합니다.

2. 프로젝트를 위한 기본 개념 공부

1. 어셈블리어에 대해서

정의: 어셈블리어는 저수준 프로그래밍 언어로, 인간이 이해할 수 있는 기호를 사용하여 기계어를 대체합니다. 어셈블리어는 기계어와 거의 1:1 로 대응되며, 각 명령어는 특정 기계어를 나타냅니다.

구성: 어셈블리어는 기본적으로 명령어, 레지스터, 메모리 주소, 상수 등으로 구성됩니다. 명령어는 CPU가 수행할 연산을 나타내며, 레지스터는 CPU 내부의 작은 저장 공간을 나타내며, 고속으로 데이터를 처리할 수 있습니다. 메모리 주소는 특정 메모리 위치를 가리키며, 상수는 변하지 않는 값을 나타냅니다.

용도: 어셈블리어는 컴퓨터 하드웨어를 정밀하게 제어해야 하는 경우에 사용됩니다. 예를 들어, 운영체제, 컴파일러, 링커, 로더, 디바이스 드라이버, 임베디드 시스템, 게임 등의 개발에 사용될 수 있습니다.

장단점: 어셈블리어의 장점은 컴퓨터 하드웨어를 정밀하게 제어할 수 있다는 점과, 프로그램의 성능을 최적화할 수 있다는 점입니다. 그러나 어셈블리어는 코드를 이해하고 작성하기 어렵다는 단점이 있습니다. 또한, 특정

컴퓨터 아키텍처에 종속적이므로, 다른 아키텍처에서는 동작하지 않을 수 있습니다.

어셈블러: 어셈블리어 코드는 어셈블러라는 프로그램을 통해 기계어 코드로 변환됩니다. 어셈블러는 각 어셈블리어 명령어를 해당하는 기계어로 번역하며, 링크 과정을 통해 실행 가능한 프로그램을 생성합니다.

이렇게 어셈블리어는 컴퓨터 하드웨어와 가장 깊은 부분에서 작동하는 언어입니다. 그러나 그 복잡성과 특정성 때문에 일반적인 프로그래밍에는 더 고수준의 언어가 주로 사용됩니다. 어셈블리어는 필요한 경우, 특히 하드웨어를 직접 제어하거나 최적화가 필요한 경우에 사용됩니다.

2. x86 아키텍처

정의: x86 아키텍처는 인텔이 개발한 마이크로로세서 아키텍처로, 8086 을 시작으로 80186, 80286, 80386, 80486 등과 같이, 이 아키텍처를 기반으로 한 프로세서들의 이름 끝에 '86'이 붙었습니다.

특징: x86 아키텍처는 CISC(Complex Instruction Set Computing) 구조를 가지고 있어, 복잡하고 다양한 기능을 가진 명령어 집합을 가집니다. 하나의 명령어가 여러 개의 연산을 수행할 수 있도록 설계되어 있으며, 이는 프로그래밍을 상대적으로 용이하게 만듭니다. 또한, 이진 호환성을 가지고 있어, 예전에 작성된 소프트웨어를 새로운 x86 프로세서에서도 실행할 수 있습니다.

운영체제 개발에 사용하는 이유: x86 아키텍처는 개인용 컴퓨터(PC) 시장에서 매우 널리 퍼져 있으므로, 이를 기반으로 운영체제를 개발하면 많은 기기에서 운영체제를 실행할 수 있습니다. 또한, 이진 호환성으로 인해 오래된 소프트웨어도 새로운 x86 기반 컴퓨터에서 실행할 수 있습니다. 또한, 오랜 역사와 널리 퍼진 사용자 기반 덕분에 다양한 개발 도구와 문서, 커뮤니티 등의 자원이 풍부하게 존재합니다.

단점: x86 아키텍처는 명령어 집합이 복잡하고 전력 소모가 높다는 단점도 가지고 있습니다. 이에 따라, 저전력이 필요한 모바일 및 임베디드 시스템에서는 ARM과 같은 RISC(Reduced Instruction Set Computing) 아키텍처가 주로 사용되고 있습니다.

3. 넷와이드 어셈블러(NASM)

정의: NASM은 인텔 x86 아키텍처를 대상으로 하는 어셈블러입니다. 이는 기계어 코드를 생성하기 위해 어셈블리어 코드를 해석하고 변환하는 도구를 의미합니다.

호환성: NASM은 MS-DOS, Microsoft Windows, Linux 등 다양한 운영체제에서 사용될 수 있습니다.

출력 형식: NASM은 다양한 출력 형식을 지원합니다. 이에는 RAW 바이너리 파일, COFF(Common Object File Format), ELF(Executable and Linkable Format), Mach-O(Mac OS X), Microsoft 16-bit OBJ, Win32, Win64 등이 포함됩니다.

문법: NASM은 Intel 문법을 사용하여 어셈블리어 코드를 작성합니다. 이는 AT&T 문법을 사용하는 GNU Assembler(GAS)와는 다르다는 점을 주의해야 합니다. Intel 문법은 코딩에 있어 명확성을 제공하며, 많은 개발자들이 선호하는 편입니다.

매크로 처리: NASM은 매크로 처리 기능을 제공합니다. 이는 반복되는 코드를 매크로로 정의하고 이를

재사용하게 해주어 코드의 가독성과 재사용성을 향상시킵니다.

사용 분야: NASM은 주로 운영체제 개발, 부트로더 작성, 저수준 프로그래밍, 시스템 소프트웨어 개발 등에 사용됩니다. 또한, 성능 최적화가 필요한 고성능 소프트웨어나 게임 개발에서도 사용됩니다.

오픈 소스: NASM은 퍼블릭 도메인 소프트웨어로서, 소스코드가 공개되어 있습니다. 이는 사용자가 NASM을 자유롭게 사용하고 수정하며, 그 결과를 공유할 수 있음을 의미합니다.

4. 부트로더

정의: 부트로더는 컴퓨터가 부팅될 때 실행되는 프로그램으로, 운영체제를 메모리로 로드하고 실행하는 작업을 담당합니다. 이는 컴퓨터를 켜면 처음으로 실행되는 소프트웨어입니다.

역할: 주요 역할은 컴퓨터의 하드웨어를 초기화하고, 운영체제 커널을 메모리로 로드하는 것입니다. 부트로더는 또한 다중 부팅 환경에서 사용자가 여러 운영체제 중 하나를 선택할 수 있게 하며, 선택된 운영체제를 메모리로 로드하고 시작합니다.

위치: 부트로더는 하드 드라이브의 부트 섹터에 위치합니다. 이는 컴퓨터가 부팅될 때 BIOS 또는 UEFI에 의해 먼저 읽히는 디스크의 특정 영역입니다.

작동 방식: 컴퓨터가 켜지면, BIOS 또는 UEFI는 부트로더를 메모리로 로드하고 실행합니다. 부트로더는 그 후 운영체제 커널을 메모리로 로드하고, 커널이 제어를 받을 수 있도록 준비합니다.

예: GRUB(GRand Unified Bootloader), LILO(LInux LOader), NTLDR(New Technology Loader, Windows XP), BOOTMGR(Windows Vista 이상) 등이 있습니다.

구현: 부트로더는 일반적으로 어셈블리어와 C 언어를 사용하여 작성됩니다. 어셈블리어는 하드웨어를 직접 제어하고, 초기화 과정을 수행하기 위해 필요하며, C 언어는 복잡한 로직을 구현하는데 사용됩니다.

5, 하드디스크 읽기

하드디스크는 컴퓨터의 주요 저장 장치 중 하나로, 우리가 사용하는 모든 데이터와 정보, 예를 들어 문서, 사진, 비디오, 음악 파일 등이 저장되는 곳입니다. 이런 데이터는 하드디스크의 표면, 즉 디스크 플래터에 매우 작은 단위인 섹터라는 곳에 저장됩니다.

하드디스크의 읽기 기능은 이러한 섹터에 저장된 데이터를 컴퓨터가 사용할 수 있도록 가져오는 역할을 합니다. 이 과정은 아래와 같습니다:

주소 지정: 운영체제는 읽어야 할 데이터가 저장된 섹터의 위치를 먼저 찾아야 합니다. 이 위치는 실린더, 헤드, 섹터(Cylinder, Head, Sector - CHS) 또는 논리 블록 주소(Logical Block Addressing - LBA)와 같은 방식으로 표현됩니다.

헤드 이동: 주소가 지정되면, 디스크 컨트롤러는 읽기/쓰기 헤드를 해당 섹터로 이동시킵니다. 이 과정을 "시크(seek)"라고 합니다. 시크 시간은 헤드가 움직이는 데 걸리는 시간을 의미하며, 이 시간이 짧을수록

하드디스크의 성능이 좋아집니다.

회전 지연: 헤드가 해당 섹터 위로 이동한 후, 디스크 플래터는 해당 섹터가 헤드 바로 아래로 오를 때까지 회전합니다. 이 과정을 "회전 지연(rotational latency)"라고 합니다. 회전 지연 시간은 디스크의 회전 속도에 따라 달라집니다.

데이터 읽기: 섹터가 헤드 바로 아래로 오면, 헤드는 섹터의 자기장을 감지하고 이를 전기 신호로 변환합니다. 이 신호는 디스크 컨트롤러에 의해 디지털 데이터로 변환되어, 이후 컴퓨터의 메모리로 전송됩니다. 이렇게 해서 우리는 저장된 데이터에 접근할 수 있게 됩니다.

이렇게 보면, 하드디스크의 읽기 기능은 상당히 복잡한 과정을 거치지만, 이 모든 것이 매우 빠르게 이루어져 사용자는 거의 즉시 데이터에 접근할 수 있습니다.

6. 리얼모드에서 보호모드 전환

리얼모드: 리얼모드는 16 비트 마이크로프로세서가 처음 시작할 때의 동작 모드입니다. 이 모드는 인텔 8086 및 8088 마이크로프로세서에 처음 도입되었으며, 이후 인텔 x86 계열의 프로세서에서도 지원됩니다. 리얼모드에서는 프로세서가 16 비트 연산만 수행할 수 있고, 주 메모리 접근이 1MB로 제한됩니다. 또한, 메모리 보호 기능이 없어 모든 소프트웨어가 전체 메모리에 직접 접근할 수 있습니다.

보호모드: 보호모드는 리얼모드에 비해 향상된 기능을 제공하는 동작 모드입니다. 이 모드는 32 비트 또는 64 비트 연산을 지원하며, 메모리 접근 제한이 없습니다. 따라서 주 메모리의 크기는 이론적으로는 4GB(32 비트) 또는 18.4 엑사바이트(64 비트)까지 가능합니다. 또한, 보호모드에서는 메모리 보호, 가상 메모리, 멀티태스킹 등의 고급 기능도 지원됩니다.

리얼모드에서 보호모드로의 전환: 컴퓨터가 처음 부팅될 때, 프로세서는 리얼모드에서 시작합니다. 이후 운영체제가 로드되면서 프로세서는 보호모드로 전환됩니다. 이 전환 과정은 다음과 같습니다:
첫째, 운영체제는 메모리를 초기화하고, 필요한 시스템 자원을 설정합니다.
둘째, 운영체제는 프로세서의 제어 레지스터 중 하나인 CR0 의 PE(Protection Enable) 비트를 1 로 설정하여 보호모드를 활성화합니다.
셋째, 운영체제는 코드 세그먼트 레지스터(CS)를 업데이트하여 보호모드에서 실행될 코드 세그먼트를 가리키도록 합니다.

이렇게 하면 프로세서는 보호모드로 전환되고, 이후에는 보호모드에서 실행되는 운영체제와 응용 프로그램이 컴퓨터의 제어를 받게 됩니다.

7. 인터럽트

인터럽트는 프로세서가 현재 실행 중인 작업을 일시적으로 중단하고 다른 작업을 수행하도록 하는 메커니즘입니다. 이는 보통 특정 이벤트가 발생했을 때 시스템이 즉시 반응해야 하는 경우에 사용됩니다.

인터럽트는 크게 두 가지 종류가 있습니다:

하드웨어 인터럽트: 이는 키보드, 마우스, 프린터 등의 하드웨어 장치에서 발생합니다. 예를 들어, 사용자가

키보드의 키를 누르면 키보드는 인터럽트를 발생시켜 프로세서에게 사용자 입력이 있음을 알립니다.

소프트웨어 인터럽트: 이는 운영체제나 응용 프로그램에서 명령을 통해 발생시킵니다. 예를 들어, 프로그램이 파일을 읽어야 할 때 운영체제는 소프트웨어 인터럽트를 사용하여 디스크 드라이버에게 파일 읽기 작업을 요청합니다.

인터럽트가 발생하면 시스템은 다음과 같은 기본적인 인터럽트 처리 과정을 수행합니다:

인터럽트 발생: 하드웨어 장치 또는 소프트웨어가 인터럽트를 발생시킵니다.

인터럽트 인식: 프로세서는 현재 실행 중인 명령을 완료한 후 인터럽트를 인식합니다.

인터럽트 처리: 프로세서는 현재 실행 상태를 저장하고, 인터럽트를 처리하기 위한 특별한 루틴인 인터럽트 서비스 루틴(ISR)을 실행합니다. ISR은 인터럽트의 원인을 파악하고 적절한 작업을 수행합니다.

인터럽트 종료: ISR이 완료되면, 프로세서는 저장했던 실행 상태를 복원하고 원래의 작업을 계속 수행합니다.

인터럽트 메커니즘 덕분에 컴퓨터 시스템은 여러 작업을 동시에 처리하고, 다양한 이벤트에 신속하게 반응할 수 있습니다.

8. 키보드 입력

키보드 입력 처리는 사용자의 키 입력을 컴퓨터 시스템이 받아들이고 처리하는 과정입니다. 이 과정은 다음과 같이 진행됩니다:

키 입력: 사용자가 키보드의 특정 키를 누릅니다. 이 때, 키보드는 해당 키에 연관된 특정 코드인 스캔 코드를 생성합니다.

인터럽트 생성: 키보드는 스캔 코드를 컴퓨터 시스템에 전송하고, 이는 하드웨어 인터럽트를 발생시킵니다. 인터럽트는 CPU가 현재 수행하고 있는 작업을 일시 중단하고 새로운 작업을 처리하도록 요청하는 신호입니다.

인터럽트 처리: CPU는 인터럽트를 감지하고 현재 수행 중인 작업을 중지합니다. 그 다음, 인터럽트 서비스 루틴(ISR)을 호출하여 인터럽트를 처리합니다. 이 경우, ISR은 키보드 인터럽트를 처리하는 코드입니다.

키 코드 변환 및 처리: ISR은 키보드로부터 받은 스캔 코드를 키 코드로 변환합니다. 키 코드는 운영 체제나 응용 프로그램이 이해할 수 있는 형태의 코드입니다. 변환된 키 코드는 운영 체제의 입력 버퍼에 저장되고, 이후 운영 체제나 응용 프로그램에 의해 처리됩니다.

작업 재개: 인터럽트 처리가 완료되면, CPU는 원래 수행하던 작업을 계속합니다.

이렇게 키보드 입력 처리는 사용자의 키 입력을 컴퓨터 시스템이 신속하게 인식하고 반응할 수 있도록 하는 중요한 과정입니다.

9. 기초적인 쉘 구현

쉘(shell)은 사용자와 운영체제 사이의 인터페이스를 제공하는 소프트웨어입니다. 사용자는 쉘을 통해 명령어를 입력하고, 쉘은 이러한 명령어를 해석하고 실행합니다.

기초적인 쉘을 구현하는 데는 다음과 같은 과정이 포함됩니다:

입력 받기: 쉘은 사용자로부터 명령어를 입력 받아야 합니다. 이를 위해, 쉘은 먼저 프롬프트를 출력한 후, 사용자의 입력을 기다립니다. 사용자가 명령어를 입력하고 엔터 키를 누르면, 쉘은 이 입력을 문자열로 받아들입니다.

명령어 파싱: 쉘은 받아들인 명령어를 파싱해야 합니다. 즉, 명령어를 개별 구성 요소로 분해해야 합니다. 이 구성 요소는 일반적으로 명령어 이름과 그에 따른 인자들입니다.

명령어 실행: 쉘은 파싱된 명령어를 실행해야 합니다. 이를 위해, 쉘은 우선 내장 명령어인지 아닌지를 확인합니다. 내장 명령어라면, 쉘 자체가 처리하고, 그렇지 않다면 쉘은 새로운 프로세스를 생성하여 해당 명령어를 실행합니다.

결과 출력: 쉘은 명령어의 실행 결과를 사용자에게 출력해야 합니다. 이는 일반적으로 표준 출력(STDOUT) 또는 표준 에러(STDERR) 스트림을 통해 이루어집니다.

이러한 과정은 일반적으로 루프 내에서 반복적으로 수행됩니다. 즉, 쉘은 사용자가 종료 명령어를 입력할 때까지 계속해서 명령어를 입력 받고, 파싱하고, 실행하고, 결과를 출력합니다.

10. 하드디스크 읽기/쓰기 기능

하드디스크의 읽기/쓰기 작업은 데이터를 저장하고 검색하는 기본적인 방법입니다. 이 과정은 다음과 같이 이루어집니다:

1. 데이터 쓰기:
명령 발행: 운영체제나 응용 프로그램이 하드디스크에 데이터를 쓰기 위한 명령을 발행합니다. 이 명령은 쓰려는 데이터와 그 데이터를 저장할 위치에 대한 정보를 포함합니다.

장치 제어기 작업: 하드디스크의 제어기는 이 명령을 받아들이고, 해당 위치로 디스크 헤드를 이동시킵니다. 이때 디스크 헤드는 디스크 표면 위를 빠르게 회전하며 이동합니다.

데이터 쓰기: 디스크 헤드가 목표 위치에 도달하면, 제어기는 헤드를 통해 디스크 표면에 전기 신호를 보냅니다. 이 신호는 디스크 표면의 자성 입자를 정렬하여 데이터를 표현합니다.

2. 데이터 읽기:
명령 발행: 운영체제나 응용 프로그램이 하드디스크에서 데이터를 읽기 위한 명령을 발행합니다. 이 명령은 읽으려는 데이터의 위치에 대한 정보를 포함합니다.

장치 제어기 작업: 하드디스크의 제어기는 이 명령을 받아들이고, 해당 위치로 디스크 헤드를 이동시킵니다.

데이터 읽기: 디스크 헤드가 목표 위치에 도달하면, 제어기는 헤드를 통해 디스크 표면의 자성 상태를 검출합니다. 이 상태는 전기 신호로 변환되어 컴퓨터 시스템이 이해할 수 있는 데이터로 해석됩니다.

이렇게 하드디스크의 읽기/쓰기 작업은 물리적인 움직임과 전자적인 신호 변환을 통해 이루어집니다. 이 기능은 하드디스크의 기본적인 작동 원리로, 컴퓨터 시스템의 데이터 저장 및 검색 기능을 가능하게 합니다.

11. ext2 파일 시스템

파일 시스템은 컴퓨터에서 데이터를 파일 형태로 저장하고 구조화하는 방법을 말합니다. 파일 시스템은 운영 체제의 중요한 부분으로, 데이터를 안전하게 보존하고, 필요할 때 쉽게 접근할 수 있도록 돕습니다.

파일 시스템은 주로 다음과 같은 역할을 합니다:
데이터 저장: 파일 시스템은 디스크나 SSD 등의 저장 매체에 데이터를 기록합니다. 이 때 데이터는 파일이라는 단위로 저장되며, 각 파일은 고유의 이름을 가집니다.

데이터 구조화: 파일 시스템은 파일들을 효율적으로 관리하기 위해 디렉토리(또는 폴더)라는 구조를 사용하여 파일들을 분류합니다. 디렉토리는 파일 또는 다른 디렉토리를 포함할 수 있으므로, 사용자는 복잡한 계층 구조를 만들어 파일들을 체계적으로 관리할 수 있습니다.

데이터 접근 제어: 파일 시스템은 파일과 디렉토리에 대한 접근 권한을 관리합니다. 이를 통해 특정 사용자가 파일을 읽거나 수정하는 것을 제한하거나 허용할 수 있습니다.

리눅스, 윈도우, macOS 등 다양한 운영 체제는 각각 다른 파일 시스템을 사용합니다. 예를 들어, 리눅스는 ext4, 윈도우는 NTFS, macOS는 APFS와 HFS+ 등의 파일 시스템을 사용합니다. 각 파일 시스템은 특정 환경에서 최적의 성능을 발휘하도록 설계되었으며, 서로 다른 특성과 기능을 가집니다.

ext2(두 번째 확장 파일 시스템)는 리눅스 운영 체제에서 사용하는 파일 시스템 중 하나입니다. ext2 는 ext 파일 시스템의 개선 버전으로, ext3, ext4 와 같은 후속 버전들의 기반을 제공했습니다.

ext2 파일 시스템은 다음과 같은 특징을 가집니다:

블록 기반 저장: ext2 는 데이터를 블록 단위로 저장합니다. 각 파일은 하나 이상의 데이터 블록으로 구성되며, 파일 시스템은 이 블록들의 위치 정보를 inode라는 구조체에 저장합니다.

inode 구조: inode는 파일의 메타데이터를 저장하는 구조로, 파일의 권한, 소유자, 생성 시간, 마지막 수정 시간, 파일 크기 등의 정보를 포함합니다. 또한, inode는 파일의 데이터 블록들을 참조하는 포인터를 가집니다.

디렉토리 구조: ext2 의 디렉토리는 파일 이름과 해당 파일의 inode 번호를 연결하는 엔트리들의 목록입니다. 이를 통해 파일 시스템은 파일 이름을 inode와 연결된 데이터 블록으로 변환할 수 있습니다.

저널링 미지원: ext2 는 저널링 기능을 지원하지 않습니다. 이는 파일 시스템의 일관성을 유지하기 위해 후속 버전인 ext3 와 ext4 에서 추가된 기능입니다. 따라서, 시스템이 비정상적으로 종료되면 ext2 파일 시스템은 fsck와 같은 도구를 사용해 수동으로 복구해야 합니다.

마지막으로, ext2 는 그 단순성과 효율성 때문에 여전히 임베디드 시스템이나 플래시 메모리 같은 곳에서 널리 사용되고 있습니다.

12. 자주 사용되는 레지스터와 어셈블리 명령어

- 레지스터와 명령어의 차이

레지스터는 컴퓨터의 중앙 처리 장치(CPU) 내부에 존재하는 작은 저장 공간을 말합니다. 이들은 CPU가 매우 빠르게 접근할 수 있으며, CPU의 연산이나 명령어의 실행에 필요한 데이터를 임시적으로 보관하거나 중간 결과를 저장하는 데 사용됩니다. 레지스터는 그 크기가 작고, 비용이 비싸기 때문에, 한정적인 개수만이 존재합니다.

각 레지스터는 각기 다른 목적으로 사용될 수 있습니다. 예를 들어, 어떤 레지스터는 데이터 연산에 사용되며, 다른 레지스터는 메모리 주소를 저장하거나, 프로그램 카운터(다음에 실행될 명령어의 주소를 가리키는 레지스터) 등의 역할을 수행합니다.

반면에, 명령어는 CPU가 수행할 작업을 나타내는 코드를 말합니다. 명령어는 여러 종류가 있으며, 각각은 CPU에게 특정한 작업을 지시합니다. 예를 들어, 데이터를 레지스터에 로드하거나 저장하는 명령어, 레지스터에 저장된 값들을 가지고 산술 연산을 수행하는 명령어, 조건에 따라 프로그램의 흐름을 제어하는 명령어 등이 있습니다.

명령어는 기본적으로 CPU가 이해할 수 있는 언어, 즉 기계어로 작성되며, 각 명령어는 바이너리 코드로 표현됩니다. 이러한 명령어들이 순서대로 실행되면서, 프로그램이 동작하게 됩니다.

즉, 레지스터는 CPU의 작업 공간이라고 볼 수 있고, 명령어는 CPU에게 어떤 작업을 해야 하는지 지시하는 역할을 합니다. 이 두 개념은 컴퓨터의 기본적인 동작 원리를 이해하는 데 필요한 중요한 요소들입니다.

- 자주 사용되는 레지스터

1. EAX: "Accumulator Register"로, 가장 일반적으로 사용되는 레지스터입니다. 이 레지스터는 주로 산술 연산, 특히 곱셈과 나눗셈에 사용됩니다. 또한, 시스템 호출의 결과 값(return 값)을 저장하는 데도 사용됩니다.

2. EBX: "Base Register"로, 데이터 세그먼트를 가리키는 역할을 합니다. 이 레지스터는 다양한 용도로 사용될 수 있지만, 주로 메모리 주소를 저장하는 데 사용됩니다.

3. ECX: "Counter Register"로, 루프 카운터로 많이 사용됩니다. 예를 들어, 어떤 작업을 특정 횟수만큼 반복해야 할 때 이 레지스터를 사용합니다.

4. EDX: "Data Register"로, EAX와 함께 사용하여 더 큰 데이터를 처리할 때 사용됩니다. 예를 들어, 64 비트의 곱셈이나 나눗셈을 수행할 때 EAX와 EDX를 함께 사용합니다.

5. ESI: "Source Index"로, 문자열이나 배열 연산에 사용되는 소스 주소를 가리킵니다. ESI는 데이터의 원본 위치를 가리키는 데 사용됩니다.

6. EDI: "Destination Index"로, 문자열이나 배열 연산에 사용되는 대상 주소를 가리킵니다. EDI는 데이터의 최종 목적지 위치를 가리키는 데 사용됩니다.

7. ESP: "Stack Pointer"로, 현재 스택의 최상위 주소를 가리킵니다. 함수 호출이 이루어질 때마다 ESP는 변경되며, 함수의 리턴 주소와 지역 변수가 저장됩니다.

8. EBP: "Base Pointer"로, 스택의 기준점을 가리키는 데 사용됩니다. EBP는 현재 함수의 스택 프레임을 가리키며, 이를 통해 함수의 매개변수와 지역 변수에 접근할 수 있습니다.

9. EIP: "Instruction Pointer"로, 다음에 실행될 명령어의 메모리 주소를 가리킵니다. CPU는 EIP 레지스터가 가리키는 주소의 명령어를 가져와 실행하고, EIP를 다음 명령어로 업데이트합니다.

10. EFLAGS: "Flags Register"로, 이전 연산의 결과에 대한 정보를 가집니다. 예를 들어, 산술 연산 결과가 0 이면 Zero 플래그가 설정되고, 오버플로우가 발생하면 Overflow 플래그가 설정됩니다. 이 플래그들은 조건 분기 명령(Jump)에서 주로 사용됩니다.

- 자주 사용되는 x86 어셈블리 명렁어

구조: [명령어] [A] [B]

1. ADD: A에 B를 더하여 A에 저장
ex: add eax 20 (eax에 20 을 더하여 eax에 저장)

2. SUB: A에 B를 빼서 A에 저장
ex: sub eax 20 (eax에 20 을 빼서 eax에 저장)

3. MOV: A에 B의 값을 넣음
ex: mov ebp 20 (ebp에 20 을 복사)

4. PUSH: 스택에 값을 넣음 -> ESP(스택의 최상위 주소)의 값이 4 만큼 줄어들고 이 위치에 새로운 값이 채워짐
ex: push reg16, push ax

5. POP: ESP가 가리키고 있는 위치의 스택 공간에서 4byte만큼을 피연산자에 복사하고 ESP + 4 를 함
ex: pop eax

6. JMP: 점프할 때 사용하는 명령어로 지정된 레이블 혹은 주소로 점프를 함
ex: jmp 0x0000 (0x0000 주소로 점프함)

7. CALL: 이 명령어는 함수를 호출하는 역할을 함
예를 들어, CALL print라고 하면 'print'라는 이름의 함수를 호출하게 됨. 그런데 여기서 중요한 점은, 함수를 호출한 후에는 원래의 위치로 돌아와야 한다는 것임. 그래서 CALL 명령어는 함수를 호출하기 전에, 다음에 실행될 명령어의 주소를 스택에 저장함. 이렇게 해서 함수가 끝나면 원래 위치로 돌아올 수 있게 됨. 그리고 함수의 결과값은 EAX 레지스터에 저장됨.

8. RET: 이 명령어는 함수가 끝나고 원래의 위치로 돌아갈 때 사용됨. RET 명령어를 실행하면, 스택에 저장되어 있던 원래의 실행 위치(EIP)를 꺼내와서 다시 돌아감.
즉, CALL 명령어로 함수를 호출하고 나서, 함수가 끝난 후에는 RET 명령어로 원래 위치로 돌아오게 됨.

이렇게 보면, CALL과 RET 명령어는 함께 동작하는 쌍으로 이해할 수 있음. CALL으로 함수를 호출하고, RET으로 원래 위치로 돌아오는 것임. 이와 달리 JMP 명령어는 단순히 다른 위치로 이동만 하고, 원래 위치로 돌아오지는 않음.

9. LEA: A에 B의 주소를 넣는 일을 함
ex: lea eax, dword ptr ds:[esi] (esi를 eax 저장 공간에 주소를 넣는 것. 따라서 eax엔 esi의 주소가 저장됨.)

10, CMP (compare): 인자 A, B의 값을 비교한다. 주로 조건 점프 명령어와 세트로 사용

11. TEST: 인자 A와 인자 B를 AND 연산
이 연산의 결과는 ZF(zero flag)에만 영향을 미치고 Operand(연산자) 자체에는 영향을 미치지 않음.
보통 rax의 값이 0 인지 확인할 때 rax 0, 0 이런 식으로 사용됨.
만약 TEST의 연산 결과가 0 이라면 ZF는 1 로, 연산 결과가 1 이라면 ZF는 0 으로 세트됨.

12. INC / DEC
INC: 인자의 값을 1 증가
DEC: 인자의 값을 1 감소

13. NOP: 아무것도 하지 않음

- x86 어셈블리어로 작성된 "Hello, World!" 출력 프로그램

```
section .data
    hello db 'Hello, World!',0

section .text
    global _start

_start:
    ; write syscall
    mov eax, 4
    ; file descriptor (stdout)
    mov ebx, 1
    ; pointer to message
    mov ecx, hello
    ; length of message
    mov edx, 13
    ; make syscall
    int 0x80
```

```
    ; exit syscall
    mov eax, 1
    ; exit status code
    xor ebx, ebx
    ; make syscall
    int 0x80
```

1. section .data로 데이터 섹션을 시작합니다. 이 섹션에는 상수나 초기화된 변수 등이 들어갑니다. hello db 'Hello, World!',0 라는 라인은 'Hello, World!'라는 문자열을 저장하는 'hello'라는 변수를 선언합니다. 'db'는 'define byte'의 약자로, 바이트 단위로 데이터를 정의합니다.

2. section .text로 코드 섹션을 시작합니다. 이 섹션에는 실제 프로그램의 코드가 들어갑니다. _start:는 프로그램의 시작점을 나타냅니다.

3. 첫 번째 시스템 호출은 문자열을 출력합니다. 시스템 호출 번호 4 는 write를 의미하며, 파일 디스크립터 1 은 표준 출력(stdout)을 의미합니다. mov ecx, hello는 hello 문자열의 시작 주소를 ecx 레지스터에 저장하고, mov edx, 13 은 문자열의 길이를 edx 레지스터에 저장합니다. 이후 int 0x80 을 통해 시스템 호출을 실행합니다.

4. 두 번째 시스템 호출은 프로그램을 종료합니다. 시스템 호출 번호 1 은 exit를 의미하며, xor ebx, ebx는 ebx 레지스터의 값을 0 으로 만들어 프로그램의 종료 코드를 설정합니다. 이후 int 0x80 을 통해 시스템 호출을 실행하여 프로그램을 종료합니다.

3. 프로젝트 계획서

1. 프로젝트 개요
프로젝트 명칭: 운영체제 개발

개발 목표: 부트로더, 하드디스크 읽기, 리얼모드, 보호모드, 인터럽트 처리, 키보드 드라이버, 기초적인 쉘, 하드디스크 드라이버, ext2 파일 시스템 구현 등의 기능을 포함하는 운영체제를 개발

타깃 플랫폼: x86 아키텍처
개발 기간: 2024 년 1 월 1 일부터 2024 년 2 월 28 일까지 총 2 개월

2. 목표 및 범위
최소 기능 목표: 부트로더 구현, 하드디스크 읽기 기능, 리얼모드에서 보호모드 전환, 기본적인 인터럽트 처리, 키보드 입력 처리, 기초적인 쉘 구현, 하드디스크 읽기/쓰기 기능, ext2 파일 시스템 구현

범위 제한: 네트워킹, GUI, 멀티 태스킹, 보안 기능 등은 구현하지 않음

.

3. 개발 환경 및 도구
프로그래밍 언어: C, 어셈블리 언어

개발 도구: GCC, NASM, VMware, HxD(프리웨어 16 진수 편집기, 디스크 편집기 및 메모리 편집기)
개발 플랫폼: Linux
소스 코드 관리: GitHub

4. 개발 단계별 세부 계획

1. 부트로더 개발: 컴퓨터 부팅 시, 커널을 메모리로 로드하는 부트로더를 개발
2. 하드디스크 읽기 모듈 개발: 하드디스크에서 데이터를 읽는 기능을 구현
3. 모드 전환 모듈 개발: 리얼모드에서 보호모드로 전환하는 기능을 구현
4. 인터럽트 핸들러 개발: 하드웨어 인터럽트를 처리하는 인터럽트 핸들러를 개발
5. 키보드 드라이버 개발: 키보드 입력을 처리하는 키보드 드라이버를 개발
6. 입출력 관리자 개발: 화면 출력 및 키보드 입력을 처리하는 입출력 관리자를 개발
7. 쉘 개발: 사용자의 명령어를 받아 커널에 요청하고, 그 결과를 사용자에게 보여주는 기능을 가진 쉘을 개발
8. 하드디스크 드라이버 개발: 하드디스크에서 데이터를 읽고 쓰는 기능을 제공하는 하드디스크 드라이버를 개발
9. 파일 시스템 (ext2) 개발: ext2 파일 시스템을 구현하여 파일 및 디렉토리를 조작하는 기능을 개발

5. 리스크 관리

문제 발생 시, 해당 문제를 정확히 파악하고 문제의 원인을 찾아내는 것이 중요함.
각 단계별로 테스트를 실시하여 문제가 발생한 지점을 찾아내고, 이를 수정하는 방식으로 진행함.
필요한 경우, 외부 자료나 커뮤니티를 활용하여 문제를 해결함.

6. 결론 및 기대 효과

본 프로젝트를 통해 운영체제의 기본 원칙과 구조에 대한 이해를 높일 수 있을 것으로 기대함.
실제로 운영체제를 구현하는 경험을 통해 이론과 실제 사이의 간극을 줄이는 것이 목표임.
이를 통해 기본적인 운영체제 개발 능력을 쌓는 데 큰 도움이 될 것으로 기대함.