시스템프로그래밍 3분반 1번째 과제 32203660 이현정

1.1 **정보 = 비트 + 컨텍스트**

프로그램은 **0 또는 1로 표시되는 비트**가 연속이고 **8비트 단위인 바이트**로 된 소스파일로 만들어진 후, 텍스트 파일로 저장된다. 대부분 컴퓨터 시스템은 바이트로 **나타내지는 텍스트 문자를 아스키 표준을 사용해서 표시**하고, 연속된 바이트가 저장된 프로그램에서 각 바이트는 특정 문자에 대응되는 정수 값을 가진다. 텍스트 라인을 종료되게 만드는 newline이라는 '₩n' 문자는 정수 값 10으로 표시한다.

모든 시스템의 내부 정보인 디스크 파일, 메모리상의 프로그램, 데이터, 네트워크를 통해 전송되는 **데이터는 모두 비트들로 표시**된다. 이들의 유일한 구분 방법은 **컨텍스트**에 의해서다. 다른 컨텍스트에서는 동일한 일련의 바이트가 정수, 부동소수, 문자열 또는 기계어 명령을 의미할 수 있다.

1.2 **프로그램**은 다른 프로그램에 의해 **다른 형태로 번역**됨

c 프로그래밍 언어로 작성된 소스 파일인 hello.c는 시스템에서 실행시키려면, 각 문장들이 다른 프로그램들에 의해 **저급 기계어 인스트럭션들로 번역**되어야한다. 인스트럭션들은 **실행가능 목적 프로그램이라는 형태로 합쳐져서 바이너리 디스크 파일로 저장**된다. 실행파일로 소스파일의 번역이 일어나기 위해서는 4단계가 거쳐진다. 전처리기, 컴파일러, 어셈블러, 링커를 합쳐서 컴파일 시스템이라고 부른다.

전처리 단계 : 전처리 Preprocessor(cpp)는 c 프로그램을 #문자로 시작하는 디렉티브(directive)에 따라 수정한다. 전처리기에게 헤더파일을 프로그램 문장에 직접 삽입하라는 지시를 하면, 일반적으로는 결과로 .i로 끝나는 새로운 c 프로그램이 생성된다.

컴파일 단계 : 컴파일러 compiler(ccl)는 텍스트 파일 hello.i를 텍스트 파일인 hello.s로 번역하고, 이 파일에는 어셈블리어 프로그램이 저장된다. 이 프로그램은 저수준 기계어 명령어를 텍스트 형태로 나타내고 있는 main 함수의 정의를 포함한다.

* 어셈블리어는 여러 상위수준 언어의 컴파일러들을 위한 공통의 출력언어를 제공하기에 유용함.

어셈블리 단계 : 어셈블러 assemble(as)가 hello.s를 기계어 인스트럭션으로 번역하고, 이것을 재배치가능 목적프로그램의 형태로 묶어서 목적파일에 그 결과를 저장한다.

링크 단계 : c 컴파일러에서 제공하는 표준 c 라이브러리에 들어있는 함수(printf)는 컴파일된 별도의 목적파일에 들어있으며, 해당 목적파일은 hello.o과 어떤 형태로든 결합되어야 한다. 링커 프로그램(ld)이 통합작업을 수행하고, 결과인 hello 파일은 실행가능 목적파일(= 실행파일)로 메모리에

적재되어 시스템에 의해 실행된다.

1.3 **컴파일 시스템이 어떻게 동작**하는지 이해가 중요

프로그램 성능 최적화하기 : 효율적인 코드 작성을 위해서 컴파일러의 내부 동작을 알 필요는 없지만, 올바른 판단을 위해서 기계어 수준 코드를 기본적으로 이해하고, 컴파일러가 어떻게 c문장을 기계어 코드로 번역하는지 알 필요가 있다.

링크 에러 이해하기 : 가장 당혹스러운 에러는 링커의 동작과 관련되고, 큰 규모의 소프트웨어 시스템을 빌드하는 경우다.

보안 약점 피하기 : 버퍼 오버플로우 취약성이 인터넷과 네트워크상의 보안 약점의 주요 원인으로 설명되었다. 이것은 프로그래머가 신뢰할 수 없는 곳에 획득한 데이터를 주의깊게 제한해야 할 필요를 거의 인식하지 못하기 때문에 발생한다. 안전한 프로그래밍의 첫 단계는 프로그램 스택에 데이터와 제어 정보가 저장되는 방식 때문에 생겨나는 영향을 이해하는 것이다.

1.4 **프로세서는 메모리에 저장된 인스트럭션을 읽고 해석**

소스 프로그램은 컴파일 시스템에 의해 실행가능한 목적파일로 번역되어 디스크에 저장된다.

1.4.1 시스템의 **하드웨어 조직**

프로그램이 실행될 때 일어나는 과정을 설명하기 위해서는 하드웨어 조직을 이해해야 한다.

버스 Buses : 시스템을 관통하는 전기적 배선군으로, 컴포넌트들 간에 바이트 정보들을 전송한다. 일반적으로 워드라고 하는 고정 크기의 바이트 단위로 데이터를 전송하도록 설계되고, 한 개의 워드를 구성하는 바이트 수는 시스템마다 보유하는 기본 시스템 변수로 대부분 4나 8바이트다.

입출력 장치 : 시스템과 외부세계의 연결을 담당하며, 입력용 키보드와 마우스, 디스플레이, 데이터와 프로그램의 장기 저장을 위한 디스크 드라이브, 실행 가능 파일인 프로그램은 디스크에 저장되어 있다. 각 입출력 장치는 입출력 버스와 **컨트롤러나 어댑터(패키징이라는 차이를 가짐)**를 통해 연결된다. 디바이스 자체가 칩셋이거나 시스템의 인쇄기판(머더보드)에 장착되는 컨트롤러와 머더보드의 슬롯에 장착되는 카드인 어댑터의 목적은 입출력 버스와 입출력 장치들 간에 정보를 주고받게 하는 것이다.

메인 메모리 : 프로세서가 프로그램을 실행하는 동안 데이터와 프로그램을 모두 저장하는 임시 저장장치이다. 물리적으로는 DRAM칩들로 구성되어 있고, 논리적으로는 연속적인 바이트들의 배열로 0부터 시작해서 고유의 주소를 가진다. 1개의 프로그램을 구성하는 각 기계어 인스트럭션은

다양한 바이트 크기를 가진다.

프로세서 : 주처리장치 CPU 또는 간단한 프로세서는 메인 메모리에 저장된 인스트럭션들을 실행하는 엔진이고, 프로세서 중심에는 워드 크기의 저장장치 또는 레지스터인 프로그램 카운터 PC가 있다.

시스템에 전원이 공급되는 동안 프로세서는 프로그램 카운터가 가리키는 곳의 인스트럭션을 반복적으로 실행하고 PC값이 다음 인스트럭션의 위치를 가리키도록 업데이트한다. 프로세서는 자신의 인스트럭션 집합 구조로 정의되는 단순한 인스트럭션 실행 모델을 따라 작동하는 것처럼 보이고, 이 모델에서 인스트럭션들은 규칙적인 순서로 실행되고 1개의 인스트럭션을 실행하는 것은 여러 단계를 수행함으로써 이루어진다. 프로세서는 PC가 가리키는 메모리로부터 인스트럭션을 읽어오고, 거기서 비트들을 해석하여 인스트럭션이 지정하는 동작을 실행하고 PC를 다음 인스트럭션 위치로 업데이트한다. 새로운 위치는 방금 수행한 인스트럭션과 메모리 상에서 연속적일수도, 아닐수도 있다.

몇 개의 단순한 동작들만 있을 뿐이고, 이들은 메인모리, 레지스터 파일, 는 새 데이터와 주소 값을 계산하는 수식/논리 처리기 주위를 순환한다. 레지스터 파일은 각각 고유의 이름을 가지는 워드 크기의 레지스터 집합으로 구성되어 있다.

적재 load : 메인 메모리에서 레지스터에 1바이트 or 워드를 이전 값에 덮이게 복사

저장 store : 레지스터에서 메인 메모리에 1바이트 or 워드를 이전 값에 덮이게 복사

작업 operate : 두 레지스터 값을 ALU로 복사하고, 2개의 워드로 수식연산 수행 후 결과를 덮어쓰기 방식으로 레지스터에 저장

점프 jump : 인스트럭션 자신으로부터 1개의 워드를 추출하고, 이것을 PC에 덮이게 복사

프로세서가 자신의 인스트럭션 집합 구조를 단순 구현한 것처럼 보인다고 하지만, 사실 최신 프로세서들은 프로그램 실행 속도를 높이기 위해 훨씬 복잡한 방식을 사용한다. 그래서 각 기계어 인스트럭션의 효과를 설명하고, 프로세서가 실제 어떻게 구현되었는지 설명하면서 프로세서의 마이크로구조와 인스트럭션 집합 구조를 주별할 수 있게 도니다.

1.4.2 hello 프로그램 실행

쉘 프로그램은 자신의 인스트럭션을 실행하면서 사용자가 명령을 입력하기를 기다리고, 문자가 입력되면 각각의 문자를 레지스터에 읽어 들인다. 엔터 키로 명령 입력을 마쳤다는 것을 알게 되면 파일 내의 코드와 데이터를 복사하라는 일련의 인스트럭션을 실행하여 실행파일을 디스크에서

메인 메모리로 로딩한다. 데이터 부분은 최종적인 출력 문자까지 포함되고, 직접 메모리 접근 DMA 기법을 이용해서 데이터는 프로세서를 거치지 않고 디스크에서 메인 메모리로 직접 이동한다.

목적파일의 코드와 데이터가 메모리에 적재된 후, 프로그램은 main 루틴의 긱계어 인스트럭션을 실해하기 시작하고, 출력 문자를 메모리로부터 레지스터 파일로 복사하고, 거기로부터 디스플레이 장치로 전송하여 화면에 글자들이 표시된다.

1.5 **캐시**가 중요함

시스템은 정보를 한 곳에서 다른 곳으로 이동시키는 일에 많은 시간을 보낸다.

ex) 프로그램의 기계어 인스트럭션들은 하드디스크에 저장되어 있고, 로딩될 때 메인 메모리로 복사되고, 프로세서가 프로그램을 실행할 때 인스트럭션들은 메인 메모리에서 프로세서로 복사된다. 데이터 스트링(출력 문자)도 디스크에 저장되었지만, 메인 메모리로 복사되었다가 디스플레이 장치로 복사된다.

물리학의 법칙 때문에 큰 저장장치는 작은 저장장치들보다 느린 속도를 갖고, 빠른 장치들은 느린 장치들보다 만드는데 많은 비용이 든다. 마찬가지로 **일반 레지스터 파일은 수백 바이트의 정보를 저장하는 반면, 메인 메모리는 십억 개의 바이트를 저장**한다. 그러나 프로세서는 레지스터 파일의 데이터를 읽는데 메모리보다 100배 빨리 읽을 수 있다. 점점 프로세서-메모리 간 격차가 증가하고 있고, 메인 메모리를 빠르게 동작하게 하는 것보다 프로세서를 더 빨리 동작하게 하는 것이 더 쉽고 비용도 적게 든다. **프로세서-메모리 간 격차에 대한 대응**으로 시스템 설계자는 작고 빠른 **캐시 메모리라고 부르는 저장장치를 고안**하여 프로세서가 단기간에 필요로 할 가능성이 높은 정보를 임시 저장할 목적으로 사용한다.

1.6 **저장장치들은 계층구조**를 이룬다

컴퓨터 저장장치들은 메모리 계층구조로 구성되어 있다. **계층의 꼭대기에서부터 아래로 내려갈수록 느리고, 크고, 바이트 당 가격이 싸진다.** 메모리 계층구조의 주요 아이디어는 한 레벨의 저장장치가 다음 하위레벨 저장장치의 캐시 역할을 한다는 것으로 L1과 L2의 캐시는 L2와 L3의 캐시이다. 이런 식으로 L3의 캐시는 메인 메모리의 캐시, 이 캐시는 디스크의 캐시 역할을 한다.

1.7 **운영체제는 하드웨어를 관리**한다

쉘 프로그램이 hello 프로그램을 로드와 실행과 메시지 출력할 때 프로그램이 키보드, 디스플레이. 디스크. 메인 메모리에 직접 액세스하지 않고, 운영체제가 제공하는 서비스를 활용한다. 운영체제

는 hw와 sw 사이에 위치한 sw계층으로 생각할 수 있으며, 응용프로그램이 hw를 제어하려면 언제나 운영체제를 통해서 한다. 운영체제는 제멋대로 동작하는 응용프로그램들이 하드웨어를 잘못 사용하는 것을 막기 위해, 응용프로그램들이 단순하고 균일한 매커니즘을 사용하여 복잡하고 매우 다른 저수준 hw 장치들을 조작할 수 있도록 하기 위해서 라는 2가지 목적을 가진다.

1.7.1 **프로세스**

프로세스라는 개념에 의해서 운영체제는 시스템에서 **1개의 프로그램만 실행**되는 것 같은 착각에 빠지게 하고 프로세서와 메인 메모리와 입출력장치를 **모두 독차지**하고 있는 것처럼 보이고, 프로세서는 인스트럭션들을 다른 방해없이 **순차적으로 실행**하고 프로그램의 코드와 데이터가 시스템 메모리의 **유일한 객체**인 것처럼 보인다.

프로세스는 실행 중인 프로그램에 대한 운영체제의 추상화로, 다수의 프로세스들은 동일한 시스템에서 동시에(프로세스의 인스트럭션들이 다른 프로세스의 것들과 섞이는 것) 실행될 수 있으며 각각은 hw를 배타적으로 사용하는 것처럼 느낀다. 대부분 시스템에서 프로세스를 실행할 cpu 숫자보다 더 많은 프로세스들이 존재하고, 이전 시스템들은 1번에 1개 프로그램만 실행에서 요즘은 멀티코어 프로세서로, 프로세서가 프로세서를 바꿔주는 방식으로, cpu가 다수 프로세스를 동시에 실행하여 여러 개를 동시에 실행하는 것처럼 보인다.

운영체제는 문맥 전환을 사용해서 교차실행을 수행하고, 프로세스가 실행하는데 필요한 모든 상태 정보의 변화를 추척한다. 어느 순간 단일 프로세서 시스템은 1개의 프로세스 코드만을 실행할 수 있는데, 운영체제는 현재 프로세스에서 새로운 것으로 제어를 옮길 때 현재 프로세스의 컨텍스트를 저장하고, 새것에 컨텍스트를 복원시키는 문맥전환을 실행해서 제어권을 새 프로세스로 넘겨주면, 이전 중단한 위치로부터 다시 실행된다.

쉘 프로세스와 hello 프로세스가 있을 때, 쉘 프로세스가 hello 프로세스 실행 명령에 의해 시스템 콜이라는 특수 함수 호출로 운영체제로 제어권을 넘겨주고, 운영체제는 쉘의 컨텍스트를 저장하고 새로운 프로세스와 컨텍스트를 생성한 뒤 제어권을 새 hello 프로세스로 넘겨준다. hello가 종료후에 운영체제는 쉘 프로세스의 컨텍스트를 복구시키고 제어권을 넘겨주면, 다음 명령 입력을 기다리게 된다.

프로세스의 전환은 운영체제 커널에 의해 관리되며, 커널은 운영체제 코드 일부분으로 메모리에 상주하고 응용프로그램이 운영체제에 작업 요청하면 특정 시스템 콜을 실행해서 커널에 제어를 넘겨주고, 커널은 요청된 작업을 수행하고 응용프로그램을 리턴한다. **커널은 별도의 프로세스가 아니지만, 모든 프로세스를 관리하기 위해 시스템이 이용하는 코드와 자료 구조의 집합**이다.

1.7.2 **쓰레드**

프로세스는 1개의 제어 흐름을 갖는 것이 아니라 쓰레드라고 하는 다수의 실해 유닛으로 최근에는 구성되어 있다. 각 쓰레드는 해당 프로세스의 컨텍스트에서 실행되며 동일한 코드와 전역 데이터를 공유하고, 프로세스들 사이에서 보다 데이터 공유가 쉽다는 것과 프로세스보다 효율적이기에 프로그래밍 모델로서 쓰레드의 중요성이 커지고 있다.

1.7.3 **가상메모리**

각 프로세스들이 메인 메모리 전체를 독점적으로 사용하는 것 같은 환상을 제공하는 추상화로, 각 프로세스는 가상주소 공간이라는 균일한 메모리의 모습을 가지게 된다. 리눅스에서는 주소공간의 최상위 영역은 모든 프로세스들이 공통으로 사용하는 운영체제 코드와 데이터를 위한 공간이고 주소공간의 하위 영역은 사용자 프로세스의 코드와 데이터를 저장한다.

프로그램 코드, 데이터: 코드는 모든 프로세스들이 같은 고정 주소에서 시작하고 c 전역변수에 대응되는 데이터 위치들이 따라오고, 코드와 데이터 영역은 실행가능 목적파일로부터 직접 초기화된다.

힙 : 런타임 힙은 크기가 고정 되어있는 코드로, 프로세스가 실행되면서 c 표준함수 malloc이나 free 호출을 하면서 런타임에 동적으로 크기를 조절한다.

공유 라이브러리 : 주소 공간의 중간 부근에 공유 라이브러리의 코드와 데이터를 저장하는 영역이 있다.

스택 : 사용자 가상메모리 공간의 맨 위에 컴파일러가 함수 호출 구현을 위해 사용하는 사용자 스택이 위치하며, 힙 처럼 동적으로 크기가 변하고, 함수 호출마다 스택이 커지고 리턴될 때는 줄어든다.

커널 가상메모리 : 주소공간의 맨 윗부분은 커널을 위해 예약되어 있어서 응용프로그램은 해당 영역을 읽거나 쓰지 못하고 커널 코드 내에 정의된 함수 직접 호출도 금지되어 있어서 작업 수행을 위해서 커널을 호출 해야한다.

1.7.4 **파일**

연속된 바이트들로, 모든 입출력장치는 파일로 모델링하고 모든 입출력은 유닉스 I/O라는 시스템 콜들을 이용해서 파일을 읽고 쓰는 형태로 이루어진다. 파일 개념은 강력해서 응용프로그램 시스템에 들어 있는 다양한 입출력장치들의 통일된 관점을 제공한다. 예로, 디스크 파일의 내용을 조작하려는 응용 프로그래머는 사용하는 특정 디스크 기술에 대해서 몰라도 되며, 동일한 프로그램이 다른 디스크 기술을 사용하는 시스템에서도 실행될 수 있다.

시스템 프로그래밍에서 배우고 싶은 목표

저는 프로그래밍에 대해서 컴퓨터에서 사용하는 어플리케이션이 동작할 수 있게 만들어진 코드라고 인식을 해왔습니다. 하지만 조금씩 수업을 들으면서 컴퓨터의 내부 구조와 컴퓨터의 동작을 위해 이해가 필요한 개념과 과정을 배워가며 제가 아는 것은 극히 일부이고, 많은 숙지가 필요하겠다는 생각이 들었습니다. 그래서 더 자세한 구조와 동작 방식을 배워가며, 이러한 과정을 이해해서 더욱 효율적인 프로그램을 짜는 사람이 되고 싶다는 생각이 들었습니다.

저의 첫 번째 목표는 컴퓨터의 시스템을 이해하는 것입니다.

현재 저는 어떻게 컴퓨터가 작동이 되는지 가시적으로 보여지는 부분에 대한 내용 밖에 알지 못합니다. 그래서 이 과목에서 배우는 내용은 기초 지식도 없는 저에게는 필요한 지식을 쌓을 수 있겠다는 생각이 들었습니다. 교수님의 설명을 들으면서 모든 내용을 제 것으로 만들 수 있다는 생각은 하지 않지만, 그래도 컴퓨터가 기본적으로 어떤 방식을 이용하고, 어떤 구조를 통해서 작동하게 되는지 해당 내용을 습득하고 싶습니다.

두 번째 목표는 더 효율적인 코딩을 하는 것입니다.

저희 분야에서는 프로그램을 읽는 능력이 기본으로 필요하고, 자신이 원하는 방향으로 만들어낼 수 있는 능력은 아니더라도 오류가 발생한다면 그 부분을 수정할 수 있는 능력은 가지고 있어야 한다는 생각이 듭니다. 그리고 프로그래머의 기본은 코딩이고, 비전공자와 달리 전공자들이 할 수 있는 것은 컴퓨터에 대한 높은 이해력을 바탕으로 한 프로그램을 만드는 능력인 것 같습니다. 그래서 어떤 과정에서 처리 시간이 더 걸리지는 이해를 하고, 코딩을 하는 과정에서도 그 부분을 고려해서 단시간에 해당하는 결과를 얻을 수 있는 프로그램을 만드는 능력을 가지고 싶습니다. 그래서 해당 과목은 저의 코딩 실력을 높이기 위한 발판의 시간으로 여기고, 이용되는 장치와 기술을 하나씩 배워보고 싶습니다.

세 번째 목표는 제가 해당 분야에 잘 맞는지 여부를 확인하는 것입니다.

컴퓨터라고 표현은 되지만, 그 속에는 데이터베이스, 어플리케이션, 시스템 등 여러 분야로 나눠지기에 시스템프로그램이라는 분야를 제가 앞으로의 진로로 고려할 수 있을지에 대해서 알아보려고 합니다. 아직 제대로 배워보지 못해서 어떨지 예상을 할 수는 없지만, 저와 맞지 않는다면 어떤 부분을 힘들어하고 이해를 못했는지를 통해서 부족한 부분이므로 해당 부분은 추가적인 공부를 하면서 기본적인 지식을 쌓겠다는 생각할 수 있을 것입니다. 반대로 시스템 프로그램이 저와 잘 맞는다면, 특히 어떤 부분에서 흥미를 느꼈는지 확인하면서 그 부분을 관심 분야로 설정할 수 있게 될 것이라고 생각합니다.

해당 내용을 잘 모르기에 더욱 이러한 말을 쉽게 할 수도 있다고 생각합니다. 하지만 해당 목표를 세움으로서 제가 이 과목에서 필요한 부분을 얻어가고, 제 진로 설정에도 도움이 되는 좋은 기회가 될 수 있기 바랍니다. 앞으로 수업을 잘 부탁드리겠습니다.