시스템프로그래밍 제1차 과제

작성자: 7분반 소프트웨어학과 김채은 (32191197)

작성일: 2020. 09. 14.

Ⅰ. 1장 1.1-1.4 내용 요약

1.1 정보는 비트와 컨텍스트이다.

'hello' 프로그램은 프로그래머가 소스 파일을 만들면서 시작된다. 소스 프로그램은 0과 1로 이루어진 비트들의 연속이다.

대부분의 현대 시스템들은 문자형 텍스트를 아스키코드를 통해 표현한다. 각각의 바이트는 정수 값을 가지고 있으며 몇몇의 문자와 대응된다. 예를 들어, 첫 번째 바이트가 정수 값 35를 가지고 있으면, 그것은 '#'과 대응된다. 아스키 문자들로만 이루어진 파일들은 텍스트 파일이라 부르고, 다른 모든 파일들은 바이너리 파일이라고 한다.

모든 정보는 비트의 묶음이다. 우리가 보는 컨텍스트만이 다른 데이터 객체를 구분할 수 있다. 다른 컨텍스트들에서 동일한 바이트들의 연속은 정수, 부동 소수점 수, 문자열, 기계 명령어를 나타낼 수 있다.

프로그래머는 숫자들의 기계 표현을 이해해야 한다.

1.2 프로그램은 다른 프로그램에 의해 다른 형태로 번역된다.

'hello' 프로그램은 C 프로그램(고급 언어)으로 시작된다. 왜냐하면 인간이 그 형태를 읽고 이해할 수 있기 때문이다. 하지만 hello.c가 실행되기 위해, 개별 C 실행문은 다른 프로그램에 의해 저급 언어 명령으로 번역되어야 한다. 이러한 명령은 실행가능한 오브젝트 프로그램(파일)이라는 형태로 패키징 되어 바이너리 디스크 파일로 저장된다.

유닉스 시스템에서, 소스 파일에서 오브젝트 파일로의 번역은 컴파일러 드라이버에 의해 행해진다. GCC 컴파일러 드라이버는 hello.c 소스파일을 읽고 실행가능한 오브젝트 파일 hello로 번역한다.

전처리 단계

전처리기(cpp)는 원래의 C 프로그램을 '#'로 시작하는 지시문에 따라 수정한다.

컴파일 단계

컴파일러(cc1)는 텍스트 파일 hello.i를 어셈블리 언어 프로그램을 포함하는 텍스트 파일 hello.s

로 번역한다. 어셈블리 언어는 다양한 컴파일러에서 공통된 출력 언어를 제공하기에 유용하다.

어셈블리 단계

어셈블러는 hello.s를 기계어로 번역하고 그들을 재배치 가능한 오브젝트 프로그램으로 패키징한다. 그리고 그 결과를 오브젝트 파일 hello.o로 저장한다.

링크 단계

hello 프로그램이 모든 C 컴파일러에서 제공되는 C 표준 라이브러리의 printf 함수를 호출한다. printf 함수는 printf.o라는 별도의 미리 컴파일된 오브젝트 파일에 있다. 그것은 반드시 hello.o 프로그램과 병합되어야 한다. 링커는 이 병합을 수행하고 그 결과로 파일은 실행가능한 오브젝트 파일로 메모리에 로딩되어 시스템에 의해 실행된다.

1.3 컴파일 시스템이 어떻게 동작하는지 이해해야 한다.

프로그래머가 컴파일 시스템이 어떻게 작동되는지 이해해야 하는 이유가 있다.

프로그램 수행 최적화: 현대의 컴파일러는 대개 좋은 코드를 만드는 매우 복잡한 도구이다. 프로그래머로써 효율적인 코드 작성을 위해 컴파일러의 내부 작동을 알 필요는 없다. 하지만 C 프로그램에서 좋은 코딩 결과를 내기 위해 우리는 기계 레벨 코드에 대해 기본적으로 이해하고 있어야 하며, 어떻게 컴파일러가 다른 C 구문을 기계 코드로 번역하는 지 알아야 한다.

링크 타임 에러에 대한 이해: 우리의 경험상, 가장 난처한 프로그래밍 에러는 링커의 작동과 관련되어 있다(특히, 큰 소프트웨어 시스템을 만들려 시도할 때).

보안 약점을 피하는 것: 오랫동안, 버퍼 오버 플로우 취약성은 네트워크와 인터넷 서버에서 주요한 보안 약점으로 여겨져 왔다. 이 취약성은 대부분 프로그래머들이 신뢰할 수 없는 출처에서 획득한 데이터의 양과 형태를 주의 깊게 제한해야 할 필요성을 이해하지 못하기 때문에 발생한다. 안전한 프로그래밍을 배우는 첫 단계는 프로그램 스택에 데이터와 제어 정보가 저장되는 방식에 의해 생겨나는 결과를 이해하는 것이다.

1.4 프로세서는 메모리에 저장된 명령을 읽고 해석한다.

유닉스 시스템에서 실행 파일을 실행하기 위해, 우리는 그 이름을 응용 프로그램(셸)에 입력한다.

셸은 프롬프트를 출력하고 명령어 라인을 입력 받아 명령을 수행한다. 셸은 hello 프로그램을 로드하고 실행한 다음 프로그램이 종료되기를 기다린다. hello 프로그램은 스크린에 메시지를 인쇄한 후 종료한다. 그러면 셸은 프롬프트를 출력하고 다음 명령어 라인이 입력되기를 기다린다.

1.4.1 시스템의 하드웨어 조직

hello 프로그램을 실행할 때 어떤 일이 일어나는지 이해하기 위해서, 일반적인 시스템의 하드웨어 조직을 이해해야 한다.

버스(Buses)

버스는 구성 요소 간 바이트 정보를 전달하는 전기관의 집합이며 시스템 전체에 걸쳐 구동된다. 버스는 일반적으로 단어라고 불리는 고정 크기의 바이트 덩어리를 전송하도록 설계되었다. 단어의 바이트 수(단어 크기)는 기본적인 시스템 매개변수이고, 이는 시스템마다 다르다. 대부분은 4바이트 또는 8바이트의 단어 크기를 가진다.

입출력 장치(Input/output devices)

입출력 장치는 외부 세계에 대한 시스템 연결이다. 우리의 예제 시스템에는 네 개의 입출력 장치가 있는데, 입력을 위한 키보드와 마우스, 출력용 디스플레이, 그리고 저장용 디스크 드라이브이다. 처음에 실행 파일은 디스크에 있다.

각각 입출력 장치는 컨트롤러 또는 어댑터에 의해 입출력 버스에 연결된다. 컨트롤러는 디바이스가 칩이거나 시스템의 인쇄 기판(motherboard)에 장착된다. 어댑터는 마더보드의 슬롯에 장착되는 카드이다. 그와 관계 없이 이들 각각의 목적은 입출력 버스와 입출력 장치들 간 정보를 주고 받도록 해주는 일이다.

메인 메모리

메인 메모리는 프로그램과 데이터를 보관하는 임시 저장 장치이다. 메인 메모리는 DRAM의 모음으로 구성된다. 논리적으로, 메모리는 바이트의 선형 배열로 구성되며, 각각 0부터 시작하는 고유한 주소(배열 인덱스)를 가지고 있다. 일반적으로 각각 기계 프로그램을 구성하는 명령어는 다양한 바이트 크기로 구성된다.

프로세서

중앙처리장치(CPU), 즉 단순 프로세서는 메인 메모리에 저장된 명령들을 해석하는(또는 실행하는) 엔진이다. 프로세서의 핵심은 단어 크기의 저장 장치(또는 레지스터)인 PC이다.

시스템에 전원이 공급되는 시점부터 꺼질 때까지, 프로세서는 반복적으로 PC가 가리키는 명령을 실행하고 PC값이 다음 명령 위치를 업데이트 한다. 프로세서는 자신의 명령어 집합 구조로 정의되는 매우 간단한 명령 실행 모델에 따라 작동하는 것으로 보인다. 이 모델에서, 명령은 규칙적인 순서로 실행되고, 한 개의 명령을 실행하는 것은 여러 단계의 연속적 수행들을 포함한다. 프로세서는 PC가 가리키는 메모리로부터 명령어를 읽고, 명령어 속 비트를 해석하고, 간단한 작동을 수행하고, 다음 명령 위치로 PC를 업데이트 한다. 이 위치는 방금 수행된 명령의 메모리에서 연속적일 수도, 그렇지 않을 수도 있다.

이 같은 아주 간단한 작동들이 있고, 이들은 메인 메모리, 레지스터 파일, 수식/논리 처리기(ALU)를 순환한다. 레지스터 파일은 고유한 이름을 갖는 단어 크기의 레지스터 집합으로 이루어져 있다. ALU는 새로운 데이터와 주소 값들을 계산한다.

로드

메인 메모리의 바이트 또는 단어를 레지스터로 복사하고 레지스터의 이전 내용을 덮어쓴다.

저장

레지스터에서 바이트 또는 단어를 메인 메모리의 위치로 복사하고, 위치의 이전 내용을 덮어쓴다.

작동

두 레지스터의 내용을 ALU에 복사하고 두 단어로 산술 연산을 한 뒤 결과를 레지스터에 저장한다. 그리고 레지스터의 이전 내용을 덮어쓴다.

점프

명령에서 단어를 추출하고 해당 단어를 PC로 복사한다. 그리고 PC의 이전 값을 덮어쓴다.

프로세서는 간단한 명령 집합을 구현하는 걸로 보이지만, 현대의 프로세서는 훨씬 더 복잡한 실행 속도를 높이는 매커니즘을 사용한다.

1.4.2 hello 프로그램 실행

시스템의 하드웨어 조직과 운용에 대한 이러한 단순한 시각으로 볼 때, 우리는 우리가 예제 프로그램을 실행할 때 어떤 일이 일어나는지 이해하기 시작한다.

처음에 셸 프로그램은 우리가 명령을 입력하기 기다리며 지시를 실행하고 있다. 키보드에 "./hello"라고 문자를 입력하면, 셸 프로그램은 각각의 것을 레지스터로 읽고, 메모리에 저장한다.

키보드의 엔터 키를 치면, 셸은 명령 입력이 끝났다고 인식하고 디스크로부터 메인 메모리로 hello 오브젝트 파일의 코드와 데이터를 카피하는 명령을 실행하여 hello 파일을 로드한다. 이 데이터는 인쇄될 "hello, world₩n"라는 문자열을 포함한다.

직접 메모리 접근(DMA)이라고 알려진 기법을 사용하여, 데이터는 프로세서를 통과하지 않고 디스크에서 메인 메모리로 직접 이동한다.

hello 오브젝트 파일의 코드와 데이터는 메모리로 로드 되고, 프로세서는 hello 프로그램의 메인 루틴의 기계어 명령을 수행한다. 이 명령들은 메모리로부터 "hello, world₩n" 문자열의 바이트를 레지스터 파일로 복사하고, 다시 디스플레이 장치로 전송하여 화면에 글자들이 보이게 한다.

표. 1장 1.7 내용 요약

1.7 운영체제(OS)는 하드웨어를 관리한다.

hello 예제로 돌아가서, 셸이 hello 프로그램을 로드하고 실행했을 때, 그리고 hello 프로그램이 그것의 메시지를 출력했을 때, 프로그램은 키보드, 디스플레이, 디스크 또는 메인 메모리에 직접적으로 접근하지 않았다. 오히려 운영체제에서 제공되는 서비스에 의존하였다. 운영체제는 응용프로그램과 하드웨어 사이의 소프트웨어 계층으로 생각될 수 있다. 응용프로그램이 하드웨어를 조종하려는 시도는 반드시 운영체제를 통해서 해야 한다.

운영체제는 두가지 주요 목적을 가지고 있다: (1) 폭주하는 응용프로그램으로부터 하드웨어가 잘못 사용되는 걸 막기 위해, 그리고 (2) 응용프로그램들이 간단하고 균등한 매커니즘으로 복잡하고 아주 다른 저수준 하드웨어 디바이스를 조작할 수 있도록 하기 위해. 운영체제는 이 두 가지 목표를 프로세스, 가상 메모리, 파일이라는 근본적인 추상화를 통해 이뤄낸다. 파일은 입출력 장치의 추상화이고, 가상 메모리는 메인 메모리와 디스크 입출력 장치의 추상화이며, 프로세스는 프로세서, 메인 메모리, 입출력 장치 모두의 추상화이다.

1.7.1 프로세스

hello와 같은 프로그램이 최신 시스템에서 실행될 때, 해당 프로그램이 프로세서, 메인 메모리, 입출력 장치 모두를 독점적으로 사용하는 것처럼 보인다. 프로세서는 방해받지 않고 차례로 명령을 수행하는 것처럼 보이며, 프로그램의 코드와 데이터는 시스템 메모리의 유일한 객체처럼 보인다. 이러한 착각은 프로세스라는 개념에 의해 일어난다.

프로세스는 실행 중인 프로그램에 대한 운영체제의 추상화다. 다중 프로세스는 동일한 시스템에서 동시에 실행될 수 있고, 각각 프로세스는 하드웨어를 독점적으로 사용하는 것처럼 나타난다. 동시에라는 건 한 프로세스의 명령들이 다른 프로세스의 명령들과 섞인다는 것을 의미한다. 대부분의 시스템에서 그들을 실행시킬 CPU 보다 더 많은 프로세스들이 존재한다. 프로세서가 프로세스들을 바꿔주며 한 개의 CPU가 여러 개의 프로세스를 동시에 실행하는 것처럼 보이게 한다. 운영체제는 문맥 전환(context switching)을 이용하여 이 교차 실행을 수행한다.

운영체제는 프로세스를 실행하는 데에 필요한 모든 상태 정보와 변화를 추적한다. 컨텍스트라고 부르는 상태 정보는, PC, 레지스터 파일, 메인 메모리의 현재 값들과 같은 정보들을 포함한다. 어느 순간 단일 프로세서 시스템은 하나의 프로세스 코드만을 실행할 수 있다. 운영체제가 현재의 프로세스로부터 새로운 프로세스로 제어를 변경하려 할 때, 컨텍스트 스위치가 수행되어 현재 프로세스의 컨텍스트를 저장하고 새 프로세스의 컨텍스트를 복원시켜, 제어권을 새 프로세서로 넘겨준다. 새 프로세스는 이전에 중단되었던 위치부터 다시 실행된다.

1.7.2 스레드

프로세스가 한 개의 제어 플로우를 갖는 것처럼 생각되지만 최근 시스템에서 프로세스는 실제로 스레드라는 다중 실행 유닛으로 구성되어 있다. 스레드는 각각 프로세스의 컨텍스트에서 실행되고, 동일한 코드와 전역 데이터를 공유한다. 다수의 스레드는 다수의 프로세스들보다 데이터의 공유가 쉽고, 효율적이다. 멀티 프로세서를 활용할 수 있다면 멀티 스레드도 프로그램을 빠르게 만드는 방법 중 하나다.

1.7.3 가상 메모리

가상 메모리는 각 프로세스들이 메모리 전체를 독점적으로 사용하는 것 같은 착각을 일으키는 추상화이다. 각 프로세스는 가상 주소 공간이라고 하는 균일한 메모리의 형태를 띤다. 리눅스에서 주소 공간의 최상위 영역은 모든 프로세스들이 공통으로 사용하는 운영체제의 코드와 데이터를 위한 공간이다. 주소 공간의 하위 영역은 사용자 프로세스의 코드와 데이터를 보관한다.

각 프로세스들에게 보여지는 가상 주소 공간은 몇 개의 영역으로 이루어져 있다.

프로그램 코드와 데이터 코드는 모든 프로세스들이 같은 고정 주소에서 시작하고, C 전역 변수에 대응되는 데이터 위치들이 따라온다. 코드와 데이터 영역은 실행 오브젝트 파일의 내용으로부터 초기화된다.

힙 코드와 데이터 영역을 런타임 힙이 즉시 따라온다. 프로세스가 시작될 때 크기가 고정되는 코드와 데이터 영역과 달리, 힙은 런타임동안 C 표준함수인 malloc이나 free를 호출하며 크기를 확장하거나 줄인다.

공유 라이브러리 주소 공간 중앙에 가깝게 C 표준 라이브러리나 수학 라이브러리와 같은 공유 라이브러리의 코드와 데이터를 저장하는 영역이 있다.

스택 유저 가상 메모리 공간의 맨 위에 컴파일러가 함수를 호출하기 위해 사용하는 스택이 있다. 사용자 스택은 함수를 호출할 때마다 스택이 커지고, 함수에서 리턴될 때는 줄어든다.

커널 가상 메모리 주소 공간의 맨 위 영역은 커널의 자리이다. 응용 프로그램들은 이 영역을 사용할 수 없고, 커널 코드 안에 정의된 함수를 직접 호출할 수도 없다. 커널을 호출해야 이러한 작업을 할 수 있다.

1.7.4 파일

파일은 연속된 바이트들 그 이상, 그 이하도 아니다. 모든 입출력 장치는 파일로 모델링을 한다. 모든 입출력은 유닉스 I/O라는 시스템 콜을 이용하여 파일을 읽고 씀으로써 실행된다.

파일 개념은 간단하지만 매우 강력하다. 왜냐하면 시스템의 다양한 입출력 장치들에 대한 통일된 관점으로 응용 프로그램을 제공하기 때문이다.

Ⅲ. 시스템 프로그래밍을 수강하며 세운 목표

그동안 C, C++, 파이썬 등의 언어를 이용하여 프로그래밍을 할 때, 문법에 맞추어 코딩하고 원하는 결과를 출력하는 데에 집중했다면 시스템 프로그래밍을 수강하면서는 프로그램이 CPU 상에서 어떻게 동작하는지 이해하고, 시간·공간 효율적인 프로그래밍을 할 수 있도록 노력할 것입니다.

또한 컴파일러와 어셈블러라는 것을 이름으로는 배웠지만 어떻게 동작하는지 몰랐으므로, 이 원리를 정확히 파악하고 로더, 링커, 디버거의 기능과 함께 시스템이 돌아가는 원리를 파악하고 싶습니다.

1주차 수업에서 소프트웨어와 하드웨어는 동떨어진 게 아니라, 두 가지가 동작하는 데에는 서로 밀접하게 연관되어 있다는 것을 배웠습니다. 앞으로의 수업을 통해 소프트웨어와 하드웨어가 어떤 식으로 연관이 되어 시스템을 움직이고, 프로그램이 수행되게 하는지 배우고 싶습니다.

2주차 수업에서는 리눅스의 동작과 명령어 등에 대해 배웠는데, 아직 실습은 하지 못했으므로 앞으로 리눅스 서버에 자주 접속해서 스스로 연습을 해보며 C언어만큼 익숙하게 사용할 수 있도록 노력하겠습니다.

이번 과제로 1장 내용을 요약하면서, 시스템이 운용되는 전반적인 흐름은 이해했지만, 아직 메인 메모리와 가상 메모리, 힙, 스택의 개념, 프로세스가 무엇인지 등 구체적인 내용은 이해하기 힘들었습니다. 앞으로 수업을 들어가면서 이러한 부분을 완전히 이해할 수 있게 되면 좋겠습니다.