과목명	시스템프로그래밍
담당교수	최종무 교수님
학과	소프트웨어학과
학번	32191105
이름	김지민
제출일자	2020.09.11

1. A Tour of Computer Systems

code/intro/hello.c

```
1  #include <stdio.h>
2
3  int main()
4  {
5      printf("hello, world\n");
6  }
```

code/intro/hello.c

그림 1 hello 프로그램

1.1 Information Is Bits + Context

hello 프로그램은 프로그래머가 hello.c라는 텍스트 파일에 저장하는 소스 프로그램으로 시작된다. 소스 프로그램은 비트들의 연속으로 각각 0 또는 1의 값을 가지며, 8bit(1Byte)로 구성된다.

현재 대부분의 시스템은 고유한 바이트 크기의 정수 값을 가진 각 문자를 나타내는 아스키(ASCII) 표준을 사용하여 텍스트 문자를 나타낸다. hello 프로그램을 아스키코드로 표현하면 다음과 같다.

#	i	n	c	l	u	d	e	<sp>	<	s	t	d	i	o	.
35	105	110	99	108	117	100	101	32	60	115	116	100	105	111	46
h	>	\n	\n	i	n	t	<sp>	m	a	i	n	(	)	\n	{
104	62	10	10	105	110	116	32	109	97	105	110	40	41	10	123
\n	<sp>	<sp>	<sp>	<sp>	p	r	i	n	t	f	(	"	h	e	l
10	32	32	32	32	112	114	105	110	116	102	40	34	104	101	108
l	o	,	<sp>	w	o	r	l	d	\	n	"	)	;	\n	}
108	111	44	32	119	111	114	108	100	92	110	34	41	59	10	125

그림 2 아스키코드로 표현한 hello.c

아스키 문자만으로 구성된 위와 같은 파일을 텍스트 파일이라고 하며, 다른 모든 파일은 바이너리파일이다. 디스크 파일, 메모리에 저장된 프로그램, 메모리에 저장된 사용자 데이터, 네트워크를 통해 전송되는 데이터 등 시스템의 모든 정보는 비트들로 표시된다. 서로 다른 객체를 구별하는 유일한 것은 이들을 바라보는 맥락이다.

1.2 Programs Are Translated by Other Programs into Different Forms

hello 프로그램은 그 형태를 인간이 읽고 이해할 수 있어 고급 C프로그램으로 시작한다. 그러나 시스템에서 hello.c를 실행시키려면 개별 C문을 다른 프로그램에 의해 저급 기계어로 번역되어야한다. 이러한 인스트럭션들은 실행 가능한 개체 프로그램이라고 하는 형태로 합쳐져 바이너리 디스크 파일로 저장된다. 유닉스 시스템에서는 컴파일러 드라이버에 의해 소스파일이 오브젝트 파일로 번역된다. (unix> gcc -o hello hello.c)

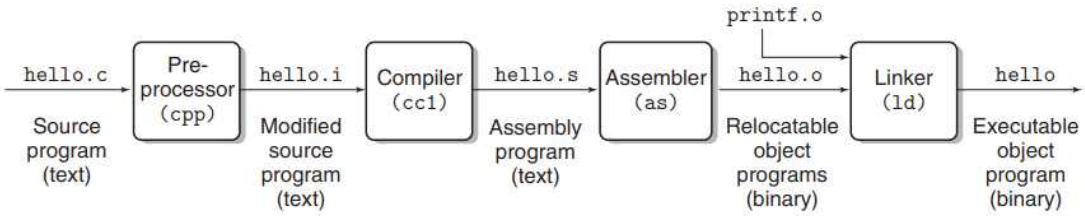

그림 3 컴파일 시스템

gcc 컴파일러 드라이버는 소스파일 hello.c를 실행파일인 hello로 번역한다. 번역은 그림 3과같이 4단계(전처리기, 컴파일러, 어셈블러, 링커)로 수행되며, 4단계를 실행하는 프로그램을 컴파일 시스템이라고 부른다.

① 전처리 단계 : 전처리기(cpp)는 # 문자로 시작하는 디렉티브에 따라 원래의 C프로그램을 수정한다. 예를 들어 #include <stdio.h>는 전처리기에게 stdio.h를 프로그램 문장에 직접 삽입하도록 지시한다. 그 결과, .i로 끝나는 새로운 C프로그램이 생성된다.

② 컴파일 단계 : 컴파일러(cc1)는 hello.i 텍스트 파일을 hello.s 텍스트 파일로 번역하며, 이 텍스트 파일에는 어셈블리언어 프로그램이 저장된다.

③ 어셈블리 단계 : 어셈블러(as)는 hello.s를 기계어 인스트럭션으로 번역하고, 이를 재배치 가능 목적프로그램 형태로 묶어, 그 결과를 목적파일 hello.o에 저장한다. 이 파일은 바이트가 문자가 아닌 기계어 명령을 인코딩하는 바이너리파일이다.

④ 링크 단계 : hello프로그램은 C컴파일러가 제공하는 printf 함수를 호출한다. printf함수는 사전 컴파일된 별도의 목적파일인 printf.o에 들어있으며 이 파일은 링커(id)에 의해 hello.o파일과 결합된다. 그 결과 hello파일이 생성되고 이것은 메모리에 로드 되어 시스템에 의해 실행될 준비가 된 실행 가능한 목적파일이다.

1.3 It Pays to Understand How Compilation Systems Work

프로그래머는 컴파일 시스템이 어떻게 동작하는지 이해하는 것이 중요하다.

① 프로그램 성능 최적화하기 : C프로그램에서 좋은 코드를 작성하기 위해서는 기계어 수

준 코드에 대한 것과, 어떻게 컴파일러가 다른 C문들을 기계어 코드로 번역하는지 이해할 필요가 있다.

ex)while문과 for문중 무엇이 더 효율적인가?

② 링크 에러 이해하기 : 가장 당혹스러운 프로그래밍 에러중 하나는 링커의 동작과 관련된 것이며, 특히 큰 규모의 소프트웨어 시스템을 빌드 할 경우 더욱 그렇다.

ex)전역변수와 지역변수의 차이점은?

③ 보안 약점 피하기 : 오랫동안 버퍼 오버플로우의 취약성이 인터넷, 네트워크상의 보안 약점의 주요 원인으로 설명되었다. 이것은 프로그래머들이 신뢰할 수 없는 곳에서 얻은 데이터의 양과 형태를 주의 깊게 제한해야 할 필요을 인식하지 못해 발생한다.

1.4 Processors Read and Interpret Instructions Stored in Memory

hello.c 소스프로그램은 컴파일시스템에 의해 hello라고 불리는 실행 가능한 목적파일로 번역되어 디스크에 저장된다. 이것을 유닉스 시스템에서 실행하기 위해서는 쉘이라는 응용프로그램에 그 이름을 입력해야한다.(unix> ./hello
hello, world
unix>)

쉘은 명령라인 인터프리터로 프롬프트 출력하고, 명령어 라인을 입력받아 실행한다. hello 프로그램은 메시지를 화면에 출력하고 종료한다. 그러면 쉘은 프롬프트를 출력해주고 다음 명령어 라인 입력을 기다린다.

1.4.1 Hardware Organization of a System

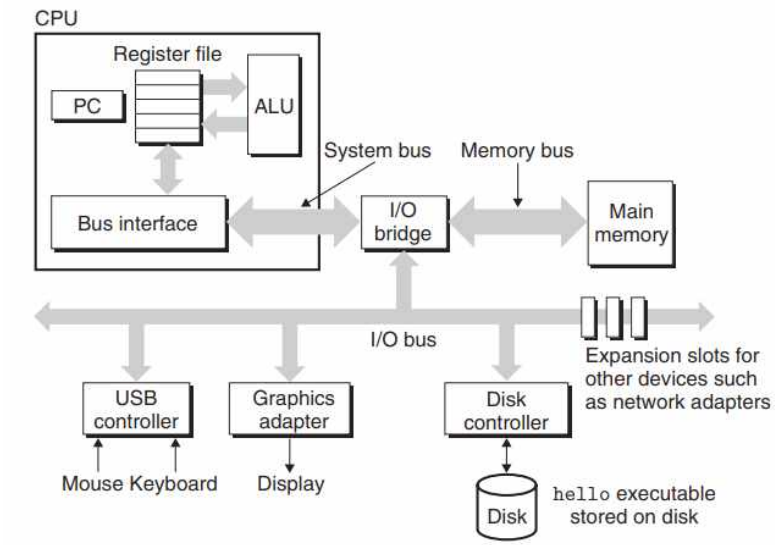

그림 4 하드웨어 조직

하드웨어 장치

① 버스 : 시스템 전체에 걸쳐 구동되는 전기 도관의 집합으로 컴포넌트들 사이의 바이트 정보들을 전송한다. 일반적으로 word라고 하는 고정 크기의 바이트 단위로 데이터를 전송하도록 설계된다. 대부분의 기기는 4바이트 또는 8바이트의 크기이다.

② 입출력장치 : 시스템을 외부와 연결해주는 장치로 키보드, 마우스, 출력용 디스플레이, 디스크 드라이브 등이 있다. 입출력장치는와 입출력 버스는 컨트롤러나 어댑터를 통해 연결된다.

③ 메인메모리 : 프로그램을 실행하는 동안 프로그램과 그것이 조작하는 데이터를 모두 저장하는 임시 저장 장치이다. 물리적으로 DRAM칩들로 구성되어있고 논리적으로는 메모리가 바이트의 선형 배열로 구성되어있다.

④ 프로세서(CPU) : 메인메모리에 저장된 인스트럭션을 실행하는 엔진으로 중심에 워드 크기의 저장장치인 PC가 있다. 시스템 전원이 꺼질때까지 프로세서는 PC가 가리키는 곳의 인스트럭션을 반복적으로 실행하고 PC값이 다음 인스트럭션의 위치를 가리키도록 업데이트한다. 단순 작업의 예로 Load, Store, Operate, Jump 가 있다.

1.4.2 Running the hello Program

처음에 쉘 프로그램은 명령 입력을 기다리며 자신의 인스트럭션을 실행한다. 사용자가 "./hello"를 입력하면 쉘 프로그램이 각 문자를 레지스터에 읽어들인 후 메모리에 저장한다. 엔터를 입력하면 쉘은 사용자가 입력을 끝낸 것을 알게된다. 그다음 쉘은 hello 목적 파일의 코드와 데이터를 디스크에서 메인메모리로 복사하는 명령을 실행하여 실행가능한 hello파일을 로드한다. 데이터 부분에는 "hello, world₩n"라는 최종 출력 문자열이 포함된다. 데이터는 DMA(직접 메모리에 접근)를 이용해 프로세서를 거치지않고 디스크에서 메인메모리로 직접 이동한다. 일단 hello 목적 파일의 코드, 데이터가 메모리에 로드되면 프로세서는 hello 프로그램의 main루틴에서 기계어 인스트럭션을 실행한다. 이 인스트럭션들은 "hello, world₩n" 문자열을 메모리로부터 레지스터 파일로 복사하고, 거기서 디스플레이 장치로 복사하여 화면에 표시한다.

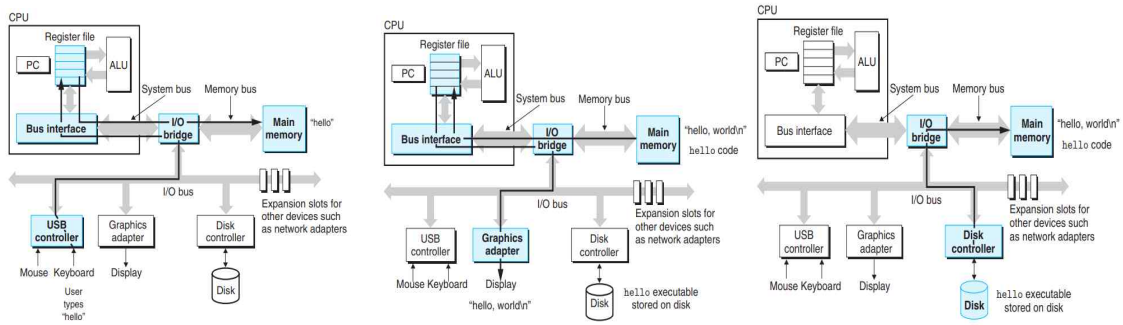

그림 5 hello프로그램 실행과정

1.7 The Operating System Manages the Hardware

쉘이 hello프로그램을 로드하고 실행할 때, hello프로그램이 메시지를 프린트할 때, 두 프로그램은 I/O장치가 아니라 운영체제가 제공하는 서비스에 의존한다. 운영체제는 어플리케이션 프로그램과 하드웨어 사이에 끼워진 소프트웨어 레이어라고 생각할 수 있다. 하드웨어를 조작하기 위한 어플리케이션 프로그램의 모든 시도는 운영체제를 거쳐야한다.

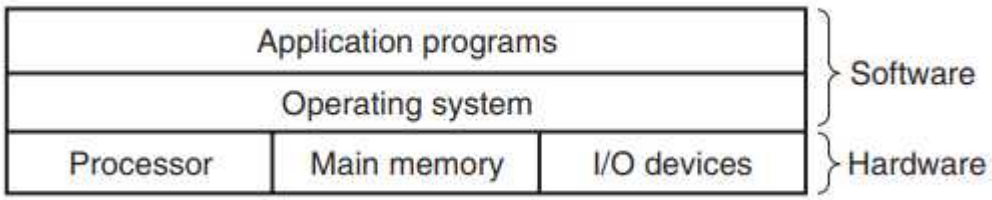

그림 6 컴퓨터 시스템의 층 구조

운영체제의 주요 목적 두 가지
① 응용프로그램의 오용으로부터 하드웨어 보호
② 복잡하고, 완전히 다른 저수준 하드웨어 장치를 조작하기 위한 간단하고 통일된 메커니즘을 응용프로그램에 제공

1.7.1 Processes

프로세스는 실행 중인 프로그램에 대한 운영체제의 추상화이다. 여러 프로세스가 동일한 시스템에서 동시에 실행될 수 있으며, 각 프로세스는 하드웨어를 독점적으로 사용하는 것처럼 보인다. 대부분의 시스템에서는 CPU보다 실행할 프로세스가 더 많다. 그래서 하나의 CPU가 프로세서 스위치를 사용하여 여러 프로세스를 동시에 실행하는 것처럼 보일 수 있다. 운영체제는 컨텍스트 스위칭이라고 알려진 메커니즘으로 이 인터리빙을 수행한다.
운영체제는 프로세스가 실행되기 위해 필요한 모든 상태정보를 추적한다. 컨텍스트라고 불리는 이 상태정보는 PC,레지스터 파일, 메인 메모리의 내용 등의 정보를 포함한다.
예제로 쉘 프로세스와 hello프로세스 두가지 동시 프로세스가있다. 처음에는 쉘 프로세스가 단독으로 실행되어 명령줄에서 입력을 기다린다. 그러다 hello프로그램을 실행하라는 명령을 받으면 쉘은 시스템콜이라는 특수 함수를 호출하여 운영체제로 제어권을 넘겨준다. 운영체제는 쉘의 컨텍스트를 저장하고, 새로운 hello프로세스와 컨텍스트를 만든 후, 새로운 hello프로세스에 제어권을 넘겨준다. hello가 종료된 후 운영체제는 쉘 프로세스의 컨텍스트를 복원하고 제어권을 다시 여기에 전달하여 다음 명령줄 입력을 기다린다.

1.7.2 Threads

프로세스는 쓰레드라고 불리는 여러 실행 단위로 구성될 수 있으며, 각각은 프로세스의 컨텍스트에서 실행되고 동일한 코드와 전역 데이터를 공유할 수 있다.

1.7.3 Virtual Memory

가상메모리란 메인메모리를 독점적으로 사용하는 듯한 착각을 각 프로세스마다 제공하는 추상화이다. 각 프로세스에는 가상주소공간이라고 알려진 균일한 메모리 모습이 있다. 리눅스 프로세스의 가상 주소공간은 다음과 같다.

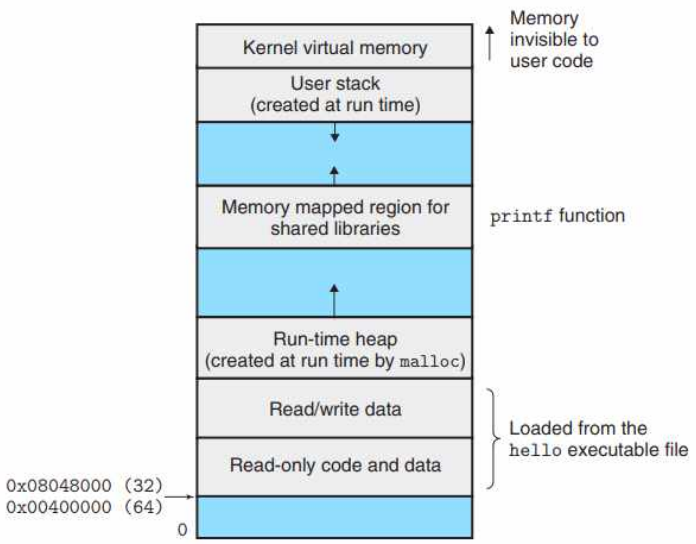

그림 7 프로세스 가상주소환경

① 프로그램 코드 및 데이터 : 코드는 모든 프로세스에 대해 동일한 고정 주소에서 시작하며 코드, 데이터 영역을 실행 가능한 목적파일을 직접 초기화한다.
② 힙(Heap) : 코드 및 데이터 영영 뒤에 런타임 힙이 바로 이어진다. 프로세스가 실행되면 C 표준함수에 대한 호출결과로 힙의 크기가 동적으로 확장, 수축된다.
③ 공유 라이브러리 : 주소 공간 중간 부근에 공유라이브러리의 코드와 데이터를 보관하는 영역이 있다.
④ 스택(Stack) : 사용자의 가상 주소 공간 맨 위에 컴파일러가 함수 호출을 구현하기 위해 사용하는 사용자 스택이 있다. 힙과 마찬가지로 프로그램 실행 중에 확장, 수축한다. 특히 함수를 부를 때 커지고, 리턴 시 줄어든다.
⑤ 커널(Kernel) 가상메모리 : 커널은 항상 메모리에 상주하는 운영체제의 부분이다. 주소 공간의 상단 영역은 커널을 위해 예약되어있고, 응용프로그램은 이 영역의 내용을 읽거나 쓰거나 커널 코드에 정의된 함수를 직접 호출할 수 없다.

1.7.4 Files

파일은 바이트의 배열이다. 디스크, 키보드 디스플레이, 네트워크를 포함한 모든 I/O장치는 파일로 모델링된다. 시스템의 모든 입출력은 유닉스 I/O라는 시스템 콜들을 이용해 파일을 읽고 쓰는 형태로 이루어진다.

<이번 학기 시스템 프로그래밍을 수강하며 배우고 싶은 목표>

이전에 컴퓨터 프로그램이 돌아가는 원리에 대해 공부한 적도, 생각해본 적도 없어서 1,2주차 강의가 상당히 어려웠다. 강의를 통해 대충 어떤 원리인지 알 것 같으면서도 생소한 단어가 많아 어렵게 느껴지는 부분이 많았다. 이전 학기에 수강했던 프로그래밍 과목들은 처음부터 직접 코딩을 하며 배웠기 때문에 나름 흥미롭게 공부할 수 있었는데 처음부터 모르는 단어들과 이론이 나오니까 솔직히 조금 당황했다. 하지만 더욱 열심히 공부해야겠다는 오기가 들었고, 이번 학기 시스템 프로그래밍 강의를 통해 컴퓨터 시스템이 무엇인지, 어떤 구조를 가지고 있는지, 어떤 원리로 프로그램이 돌아가는지 등에 대한 기초적인 부분을 확실히 다질 것이다.

2주차 강의를 들으며 느낀 점은 컴퓨터 시스템에서 추상화가 굉장히 중요하다는 것이었다. 추상화는 미술과 관련된 용어로만 들어본 적이 있어서 처음에는 잘 이해가 가지 않았는데 강의를 듣고, 직접 검색하다보니 무슨 의미인지 언뜻 알 것 같았다. 앞으로 강의를 통해 추상화가 무엇인지와 추상화 간에 인터페이스에 대해 확실히 이해하고 이해에서 끝나는 것이 아니라 컴퓨터 시스템을 다양한 각도에서 추상화할 수 있는 능력을 키울 것이다.

시스템 프로그래밍에 대해 정확히 알지 못해서 인터넷에 검색해봤는데, 시스템 프로그래밍의 사전적 의미는 컴퓨터 시스템을 동작시키는 프로그램 제작이라고 한다. 또한 윈도우, 유닉스 같은 운영체제 뿐 아니라 해당 운영체제 용 비디오 드라이버, 파일 시스템 드라이버, 네트워크 드라이버 등을 제작하는 것도 시스템 프로그래밍의 범주에 속한다고 한다. 이렇게 시스템에 대해 공부하는 이유는 프로그래밍에 있어 더 나은 퍼포먼스를 이끌어 내기 위해서, 즉 하드웨어와 코드 사이의 긴밀한 관계를 알고 그 사이 간에 여러 최적화를 위해서 라고 한다. 나는 이 말을 보며 어떤 분야의 프로그래머가 되더라도 시스템을 공부하는 것은 필수구나 라는 생각이 들었다. 또한 원래 하드웨어에 대한 공부는 선택이라고 생각했는데 시스템 프로그래밍 강의를 들으며 하드웨어와 시스템프로그램이 밀접한 관계가 있다는 것을 알았다. 더불어 프로그래밍을 하면서 쉽게 찾을 수 없는 버그중 하나가 하드웨어적인 문제들이 많다고 한다. 그래서 이번 시스템프로그래밍 강의를 통해 하드웨어적인 부분들에 대해 공부하고 보다 넓은 통찰력으로 프로그래밍을 할 수 있는 사람이 돼야겠다고 생각했다.