과목명	시스템 프로그래밍
담당교수	최종무 교수님
학과	소프트웨어학과
학번	32200185
이름	곽다은
제출일자	2021.09.17

1.1 Information is Bits + Context

정보는 비트와 컨텍스트로 이루어진다

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

다음 hello 프로그램은 프로그래머가 작성한 소스코드를 hello.c라는 파일로 저장하여 생겨난 프로그램이다. 이 소스 프로그램은 값이 0 또는 1인 비트들로 이루어져 있는데, 각 비트들은 하나의 텍스트 문자를 나타낸다. 거의 모든 컴퓨터 시스템은 텍스트 문자를 바이트 길이의 정수 값으로 표현된 아스키코드를 통해 표시한다.

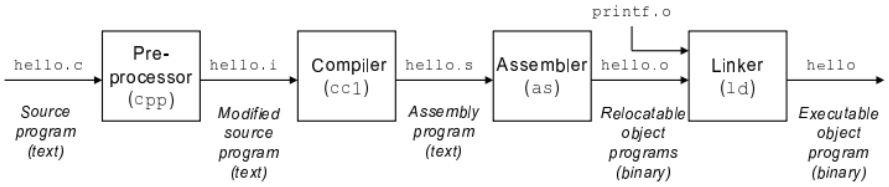
#	i	n	c	l	u	d	e	<sp>	<	s	t	d	i	o	.
35	105	110	99	108	117	100	101	32	60	115	116	100	105	111	46
h	>	\n	\n	i	n	t	<sp>	m	a	i	n	(	)	\n	{
104	62	10	10	105	110	116	32	109	97	105	110	40	41	10	123
\n	<sp>	<sp>	<sp>	<sp>	p	r	i	n	t	f	(	"	h	e	l
10	32	32	32	32	112	114	105	110	116	102	40	34	104	101	108
l	o	,	<sp>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

위의 그림과 같이 오로지 아스키코드만으로 이루어진 파일을 텍스트 파일이라 하고, 나머지 파일들은 모두 바이너리 파일이라 한다. 우리는 hello.c 프로그램을 통해 모든 정보들은 메모리에 저장되고, 비트들로 표현됨을 알 수 있다. 이들이 갖는 컨텍스트만이 이런 객체들을 구분할 수 있는 유일한 방법이다. 예를 들어, 다른 컨텍스트에서 동일한 비트들은 정수, 부동 소숫점, 문자열 또는 컴퓨터 명령으로 다른 기능을 나타낼 수 있다. 따라서 우리는 프로그래머로서, 숫자의 기계적인 표현을 이해할 필요가 있다.

1.2 Programs are Translated by Other Programs into Different Forms

프로그램은 다른 프로그램에 의해 다른 형태로 번역된다

hello 프로그램은 사람들이 이해할 수 있는 high-level의 C언어로 작성된 프로그램이다. 그러나 이 프로그램을 시스템상에서 실행하려면, 각 문장들은 low-level 언어의 인트럭션으로 번역되어야 한다. 이 인트럭션들은 실행기능 목적 프로그램이라 부르고, 바이너리 디스크 파일 형태로 저장된다. 다음 그림은 이 과정을 설명해준다.



- 전처리 단계 : 전처리기(cpp)는 원래의 C 프로그램을 #문자로 시작하는 명령에 따라 수정한다. 예를 들어, #include <stdio.h>는 헤더파일 stdio.h에서 내용들을 읽어들여 프로그램 텍스트에 직접 삽입하라는 명령을 지시하고, 결과적으로 .i로 끝나는 새로운 C 프로그램이 생성된다.
- 컴파일 단계 : 컴파일러(cc1)는 전처리 단계에서 생성된 .i 파일을 어셈블리어가 포함된 텍스트 파일인 .s파일로 번역한다. 어셈블리 언어는 서로 다른 high-level 언어와 컴파일러에 대하여 공통된 출력 언어를 제공하기 때문에 유용하다.
- 어셈블리 단계 : 어셈블러(as)는 컴파일 단계에서 생성된 .s파일을 기계어 인스트럭션으로 번역하고, 재배치 가능한 객체 프로그램 형식으로 패키징하여 .o로 끝나는 파일로 저장한다.
- 링크 단계 : hello 프로그램에서 사용하는 printf 함수는 컴파일러마다 제공되는 표준 C 라이브러리의 일부분이다. printf 함수는 printf.o라는 사전 컴파일된 파일에 들어있는데, 이 파일은 hello.o 프로그램과 병합되어야 한다. 링커(ld)가 이 단계를 수행하고, 결과적으로 수행 가능한 파일 객체를 생성한다.

1.3 It Pays to Understand How Compilation Systems Work
컴파일 시스템이 어떻게 동작하는지 이해하는 것은 중요하다

프로그래머들은 왜 컴파일 시스템이 어떻게 동작하는지 이해해야 할까?

- 프로그램 성능 최적화하기 : 프로그래머들은 효율적인 코드를 작성하기 위해서라면 컴파일러의 내부 작업을 알 필요가 없지만, switch문과 if-else문 중 어떤 것을 사용할 것인가와 같은 C 프로그램 코딩에서의 좋은 결정을 내리기 위해서는 기계어 코드와 번역에 대한 기본적인 이해가 필요하다
- 링크 타임 에러 이해하기 : 링커의 작동과 관련된 오류는 우리가 프로그래밍을 하면서 발생하는 오류 중 가장 복잡한 오류 중 하나이다. 특히 규모가 큰 프로그램을 작성할 때 많이 발생한다.
- 보안 약점 피하기 : 수년 동안 네트워크 및 인터넷 서버의 많은 보안 결함은 버

퍼 오버플로 취약성으로부터 나타났다. 이러한 취약성은 신뢰할 수 없는 곳에서 받는 데이터의 양과 형식을 제한해야 함을 이해하는 프로그래머가 적기 때문이다.

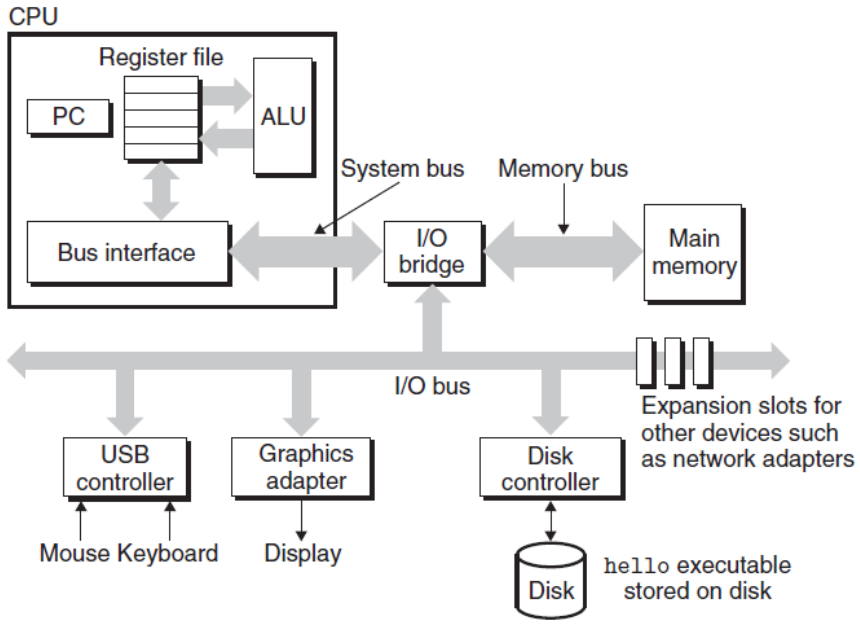
1.4 Processors Read and Interpret Instructions Stored in Memory
프로세서는 메모리에 저장된 인스트럭션을 읽고 해석한다

지금까지 살펴본 내용으로는, hello.c 프로그램은 컴파일 시스템에 의해 hello라는 실행가능한 파일로 번역되어 디스크에 저장되었다. 이 실행파일을 유닉스 시스템에서 실행하려면 우리는 쉘이라는 응용 프로그램을 사용한다.

```
linux> ./hello
hello, world
linux>
```

쉘은 위와같이 프롬프트를 출력하고, 다음 명령을 기다렸다 수행하는 커맨드라인 인터프리터이다. 입력 내용이 명령어와 일치하지 않으면 파일 이름으로 간주하여 실행시킨다.

1.4.1 시스템의 하드웨어 조직



우리는 이 그림을 통해 hello 프로그램을 실행할 때 무슨 일이 일어나는지 이해할 수 있다.

- 버스(Buses) : 버스는 시스템 전체에서 실행되는 전기적 배선군이다. 시스템의 구성 요소 간에 바이트로 정보를 전달하며, 전형적으로는 워드(word)라고 하는 고정된 사이즈의 바이트 단위로 데이터를 운송하도록 되어있다.
- 입출력 장치 : 입출력 장치는 키보드, 마우스 등의 외부 세계와의 연결을 담당하는 장치이다. 각각의 입출력 장치들은 컨트롤러(Controller)나 어댑터(Adapter) 그리고 입출력 버스로 연결되어 있다. 컨트롤러는 장치 자체 또는 시스템의 인쇄회로에 있는 칩 세트이고, 어댑터는 마더보드의 슬롯에 연결하는 카드이며, 둘 다 입출력 버스와 입출력 장치 사이의 정보를 주고받는 데에 사용된다.
- 메인 메모리 : 메인 메모리는 프로세서가 프로그램을 실행하는 동안 프로그램과 데이터를 모두 저장하는 임시 저장장치이다. 물리적으로 메인메모리는 DRAM 칩들로 구성되어 있고, 논리적으로는 각각 고유한 주소를 가진 바이트의 배열로 구성된다.
- 프로세서 : 중앙 처리 장치인 CPU와 프로세서는 메인 메모리에 저장된 명령을 해석하는 엔진이다. 프로세서의 중심에는 워드 크기의 저장장치인 PC가 있고, 항상 메인메모리의 기계어 명령에 대기중이다. 시스템에 전원이 공급되는 시점부터 전원이 꺼질 때까지 프로세서는 PC가 가리키는 위치의 명령을 반복적으로 실행하고, 다음 위치를 가리키도록 프로그램 카운터를 업데이트한다. CPU가 수행할 수 있는 간단한 작업으로는 적재(Load), 저장(Store), 작업(Operate), 점프(Jump)가 있다.

1.4.2 hello 프로그램의 실행

hello 프로그램을 실행하기 위해 처음으로는 쉘 프로그램이 자신의 인스트럭션을 수행하기 위해 명령을 기다린다. 우리가 ./hello 를 키보드를 통해 입력하면, 쉘은 각 문자들을 레지스터에 읽고 메모리에 저장한다. 우리가 enter를 입력하면 쉘은 명령이 끝났다는 것을 인지하고, 코드와 데이터를 메인메모리의 디스크에 목적파일로 복사하는 인스트럭션들을 실행하여 수행가능한 hello 파일로 로딩한다. 이 데이터 부분에 최종적으로 프린트되는 hello, world₩n 문자들이 포함된다.

1.5 Cahes Matter

캐시 메모리

위의 내용에서 알아보았듯, 시스템은 정보를 한 장소에서 다른 장소로 이동시키는 데 많은 과정을 거쳐야하고, 그에따른 시간을 소비한다. 프로그램이 메인메모리로 복사되어 로딩되는 데, 프로그래머의 관점에서 이 복사물의 대부분은 프로그램의 실제 작업 수행을

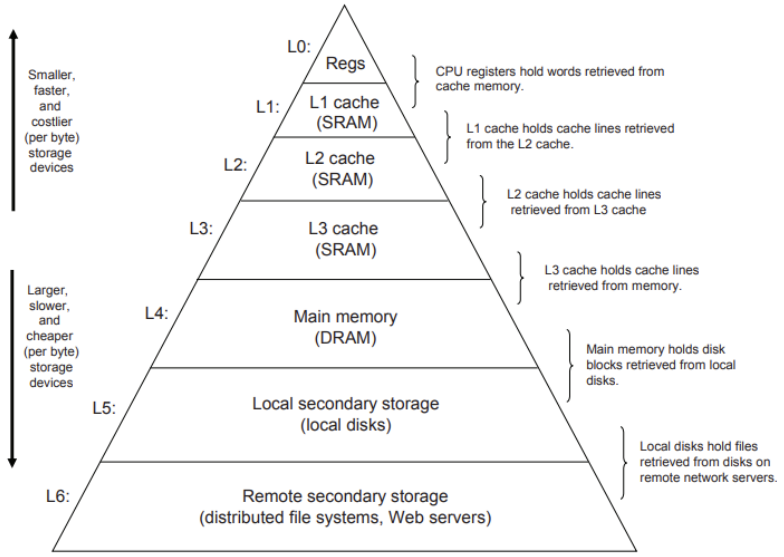
느리게 하는 오버헤드이다. 따라서 시스템 설계자의 주요 목표는 이러한 복사 작업을 가능한 빨리 실행하게 하는 것이다. 그러나 물리적으로, 저장장치는 클수록 속도가 느리고, 속도가 빠를수록 제작 비용이 많이 든다. 일반적인 레지스터 파일은 메인 메모리에 수백 바이트의 정보만 저장한다. 반면 프로세서는 메모리보다 거의 100배 빠른 속도로 레지스터 파일에서 데이터를 읽을 수 있다. 이러한 프로세서-메모리 격차를 해결하기 위해 시스템 설계자는 캐시 메모리라고 하는 작고 빠른 저장장치를 고안하였다.

캐시 메모리는 크기와 속도에 따라 구분되는데, 예를 들면 L2 캐시 메모리는 수십만에서 수백만 바이트의 대용량 캐시가 특수 버스를 통해 프로세서에 연결된다. 캐시는 정적 램(static random access memory)으로 알려진 하드웨어 기술로 구현된다. 캐싱의 핵심적인 부분은 프로그램이 로컬 지역의 데이터와 코드에 접근하려는 경향을 이용하여 시스템이 매우 크고 빠른 메모리의 효과를 모두 얻을 수 있게 되었다는 것이다. 자주 엑세스하는 데이터를 보관하도록 캐시를 설정하면, 경향성과 로컬데이터를 이용하여 빠르게 메모리 작업을 수행할 수 있다.

1.6 Storage Devices Form a Hierarchy

저장 장치는 계층 구조를 형성한다.

실제로 모든 컴퓨터 시스템의 저장 장치는 다음 그림과 유사한 메모리 계층 구조로 구성된다.



위층에서 아래층으로 이동함에 따라 수행속도가 느리고, 크기가 커지고, 비용이 적게 들게 된다. 계층구조의 꼭대기를 차지하는 L0층은 레지스터 파일이 차지한다. L1층에서 L3층은 1.5에서 보았던 캐시 메모리이고, L4층은 메인 메모리가 차지한다. 이러한 메모리 계층 구조의 주요 개념은, 한 층의 스토리지는 차상위 층의 스토리지를 위한 캐시 역할을 한다

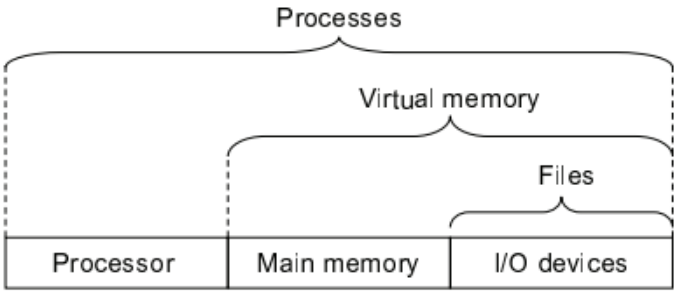
는 것이다.

1.7 The Operating System Manages the Hardware
운영체제는 하드웨어를 관리한다.

우리는 운영체제를 응용 프로그램과 하드웨어 사이에서 작동하는 소프트웨어 계층이라고 생각할 수 있겠지만, 응용 프로그램에서 하드웨어를 조작하려는 모든 시도는 운영 체제를 거쳐야 한다. 운영 체제 시스템에는 두가지 핵심적인 목적이 있다.

1. 어플리케이션에 의한 하드웨어의 오용으로부터 보호한다.
2. 매우 복잡하거나 low-level의 하드웨어 장치들을 조작하기 위한 단순하고 균일한 메커니즘을 제공한다.

운영 체제는 다음 그림과 같이 추상화 된 프로세스, 가상 메모리, 파일들을 통해 두 가지 목표를 모두 달성한다.



1.7.1 프로세스

운영체제는 하나의 시스템에 하나의 프로그램만이 실행 중인 것 같다는 착각을 일으켜, 프로그램의 명령을 중단 없이 차례로 실행하는 것처럼 보인다. 하지만 프로세스는 운영체제의 추상화로서, 대부분의 프로세스들은 동일한 시스템에서 동시에 실행될 수 있고, CPU의 개수보다 실행되는 프로세스의 개수가 더 많다.

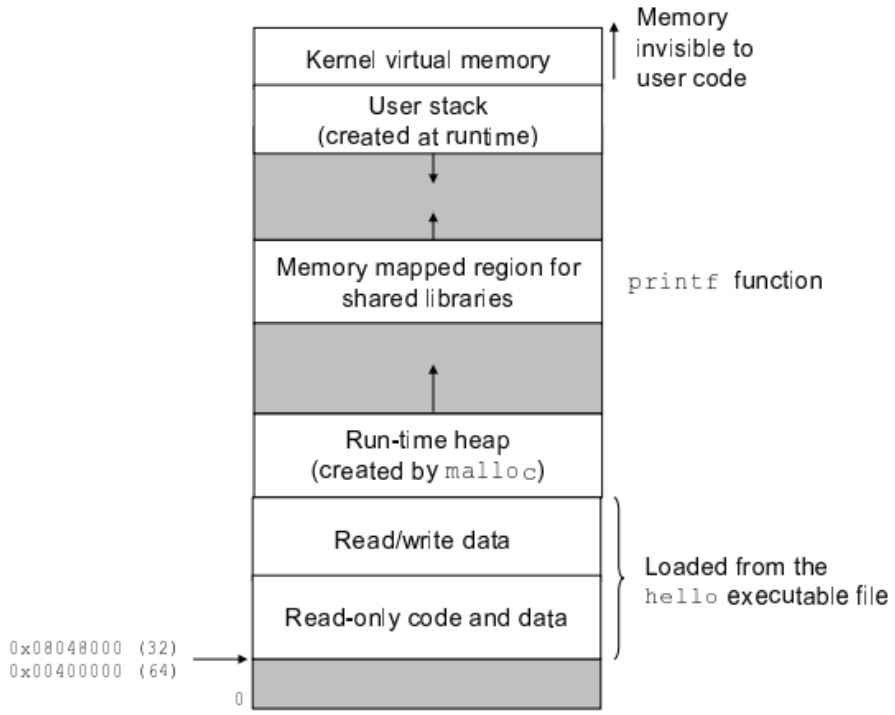
과거의 시스템들은 동시에 한개의 프로그램만 실행할 수 있었지만 요즘의 운영체제는 컨텍스트 전환이라는 메커니즘으로 교차실행이 가능하다. 운영체제는 현재 프로세스의 컨텍스트를 저장하고, 새로운 프로세스의 컨텍스트를 복원한 다음, 새로운 프로세스에 제어를 전달함으로써 컨텍스트 간의 전환을 수행한다.

1.7.2 쓰레드

프로세스는 단일 제어 흐름으로 보이지만, 실제로 현대 시스템에서 프로세스는 컨텍스트에서 각각 실행되며, 동일한 코드와 글로벌 데이터를 공유하는 쓰레드라고 하는 실행 단위로 구성된다. 다중 스레딩은 여러 프로세서를 사용할 수 있을 때 프로그램을 더 빠르게 실행할 수 있는 방법 중 하나이다.

1.7.3 가상 메모리

가상 메모리는 각 프로세스들이 메인 메모리를 독점적으로 사용하고 있는 것 같은 착각을 일으키는 추상화이다. 각각의 프로세스는 가상 주소 공간이라고 하는 같은 크기의 메모리를 갖고, 이 공간에 프로세스의 코드와 데이터를 저장한다. 다음 그림은 리눅스 프로세스들의 가상 주소 공간을 나타낸다.



주소 공간의 맨 위 영역은 모든 프로세스에 공통으로 해당되는 운영 체제의 코드 및 데이터를 위해 사용된다. 주소 공간의 하단 영역은 사용자의 프로세스에 의해 정의된 코드와 데이터가 있다.

각 프로세스에서 볼 수 있는 가상 주소 공간은 각각 특정 목적을 가진 영역으로 구성되어있다.

- 프로그램 코드와 데이터 : 코드는 모든 프로세스들의 같은 고정된 주소로부터 시작한다. 이러한 코드와 데이터 영역은 수행가능한 목적 파일의 내용들로 직접 초

기화되어 수행된다.

- 힙(Heap) : 코드와 데이터 영역은 런타임 힙 바로 다음에 위치한다. 프로세스가 실행되면, 힙은 malloc과 free와 같은 C 표준 라이브러리의 호출에 대한 결과로 런타임에 동적으로 할당되고 줄어든다.
- 공유 라이브러리 : 주소 공간의 중간 부근에는 C 표준 라이브러리 및 수학 라이브러리와 같은 공유 라이브러리의 코드와 데이터를 보관하는 영역이 있다.
- 스택(Stack) : 주소 공간의 맨 위에는 컴파일러가 함수 호출을 구현하기 위해 사용하는 사용자 스택이 있다. 힙과 마찬가지로 프로그램 실행 중에 동적으로 확장 및 축소되는데, 함수 호출 시 커지고, 함수 리턴 시 줄어든다.
- 커널 가상 메모리 : 주소 공간의 맨 윗부분은 커널을 위한 공간이다. 응용 프로그램들은 이 영역의 내용을 읽고 쓰거나, 커널 코드에 정의된 함수를 직접 호출하는 것이 불허된다. 따라서 이러한 작업을 수행하기 위해 커널을 호출해야 한다.

1.7.4 파일

파일은 바이트의 연속 그 이상도 이하도 아니다. 모든 입출력 장치는 파일로 모델링되고, 모든 입출력은 Unix I/O라는 작은 시스템 호출 세트를 사용하여 파일을 읽고 쓰는 방식으로 수행된다.

시스템 프로그래밍을 통해 얻어갈 점

소프트웨어학과에서 3학기동안 코딩 위주의 수업을 들으면서 visual studio같은 개발 툴 내에서 코드를 짜는 위주의 수업들을 주로 들어왔기 때문에, 하드웨어라면 CPU, 컴파일러, cmd창, 디스크, 램과 같은 일상생활에서도 많이 쓰이는 단어들의 개념정도만 아는 수준이었다. 그러다보니 코딩을 하며 소프트웨어를 개발하는 데에 하드웨어적 지식이 필요할거라는 생각을 한 적이 없었는데, 시스템 프로그래밍 수업을 3주동안 수강해보니 생각이 많이 달라졌다.

high-level 언어들을 기계어로 바꾸는 과정같이 내가 코딩하며 당연하게 생각하고, 원리는 궁금해하지 않았던 부분들을 자세히 배우니까 흥미가 생겼고, 그동안 발생했던 에러들도 되짚어보게 되었다. 시스템 프로그래밍을 배우면서, 시스템상에서 코드가 실행되는 과정을 정확히 이해함으로써 내가 짠 코드상의 에러가 아닌 시스템상에서 발생하는 에러들을 해결하는 능력을 기르고 싶다.

가상환경에서 리눅스를 한 번 다뤄본 경험이 있어서, ls, gcc, vi등의 명령어들에는 익숙해져있었지만, 각각 어떤 명령을 의미하고, 내부에서는 어떤 프로세스들이 수행되는지 정확하게 알지 못했다. 이 수업을 통해 이런 명령어들에 의한 프로세스들의 이해를 기반으로 컴파일러, 어셈블러, 링커, 로더와 같은 대표적 시스템 소프트웨어에 대한 이해도를 높일것이다. 따라서 앞으로 프로그램을 설계할 때 하드웨어적으로도 최적화 된 코드를 짤 수 있도록 다양한 방법들을 배워서 이전보다 나은 프로그램을 개발할 것이다.