과목명	시스템 프로그래밍
담당교수	최종무 교수님
학과	소프트웨어학과
학번	32153180
이름	이상민
제출일자	2018.9.19

[hello 프로그램]

```
#include <stdio.h>

int main()
{
    printf("hello, world\n");
    return 0;
}
```

1.1 정보는 비트와 컨텍스트로 이루어진다

hello 프로그램은 프로그래머가 작성한 소스 프로그램으로 시작해 hello.c라는 텍스트 파일러 저장된다. 소스 프로그램은 0 또는 1로 표시되는 비트들의 연속이며, 바이트(byte)라는 8비트 단위로 구성된다. 각 바이트는 텍스트 문자를 나타낸다.

컴퓨터 시스템은 텍스트 문자를 아스키(ASCII) 표준을 사용하여 표시하는데, 이것은 각 문자를 바이트 길이의 정수 값으로 나타낸다. 위의 프로그램을 아스키 값으로 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	SP
108	111	44	32	119	111	114	108	100	92	110	34	41	59	10	32
SP	SP	SP	r	e	t	u	r	n	SP	0	;	\n	}	\n	
32	32	32	114	101	116	117	114	110	32	48	59	10	125	10	

위처럼 오로지 아스키 문자들로만 이루어진 파일들은 텍스트 파일, 다른 모든 파일들은 바이너리 파일이라고 한다. 모든 시스템 내부의 정보(디스크 파일, 메모리상의 프로그램, 데이터, 네트워크를 통해 전송되는 데이터)는 비트들로 표시된다. 이들을 바라보는 컨텍스트가 유일하게 서로 다른 객체들을 구분해준다.

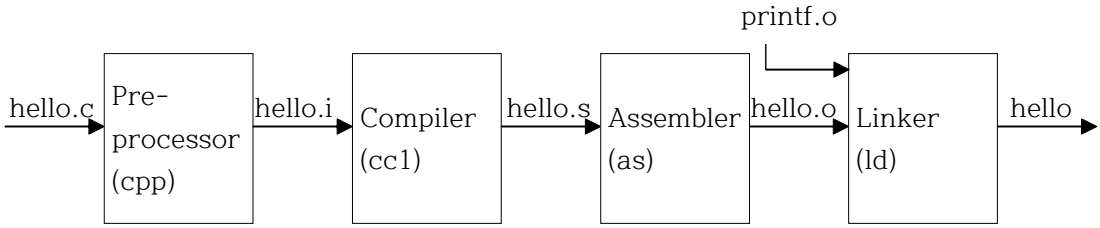
1.2 프로그램은 다른 프로그램에 의해 다른 형태로 번역된다

hello 프로그램은 인간이 바로 이해하고 읽을 수 있어 고급 C 프로그램으로 시작한다. 하지만 hello.c를 실행시키려면 각 문장들은 다른 프로그램들에 의해 저급 기계어 인스트럭션들로 번역되어야 한다. 이 인스트럭션들은 실행가능 목적 프로그램이라고 하는 형태로 합쳐져 바이너리 디스크 파일로 저장된다.

컴파일러 드라이버는 유닉스 시스템에서 다음과 같이 소스파일에서 오브젝트 파일로 번역한다.

```
linux> gcc -o hello hello.c
```

GCC 컴파일러 드라이버는 소스파일 hello.c를 읽어 실행파일인 hello로 번역한다. 번역은 4개의 단계를 거쳐 실행되는데 이 4단계를 실행하는 프로그램(전처리기, 컴파일러, 어셈블러, 링커)을 합쳐서 컴파일 시스템이라고 부른다.



■ 전처리 단계 : 전처리기(cpp)는 본래의 C 프로그램을 # 문자로 시작하는 디렉티브에 따라 수정한다. 예를 들어 #include <stdio.h>는 전처리기에게 stdio.h를 프로그램 문장에 직접 삽입하라고 지시한다. 그 결과 .i로 끝나는 새로운 C 프로그램이 생성된다.
■ 컴파일 단계 : 컴파일러(cc1)는 텍스트파일 hello.i를 텍스트파일인 hello.s로 번역하며, 이 파일에는 어셈블리어 프로그램이 저장된다.
■ 어셈블리 단계 : 어셈블러(as)가 hello.s를 기계어 인스트럭션으로 번역하고, 이들을 재배치가능 목적프로그램 형태로 묶어 hello.o라는 목적파일에 그 결과를 저장한다. 이 파일은 17바이트를 포함하는 바이너리 파일이다.
■ 링크 단계 : hello 프로그램이 C 컴파일러에서 제공하는 printf 함수를 호출한다. printf 함수는 이미 컴파일된 별도의 목적파일인 printf.o에 들어 있으며, 이 파일은 hello.o 파일과 결합되어야 한다. 링커 프로그램(ld)이 이 통합작업을 수행한다.

1.3 컴파일 시스템이 어떻게 동작하는지 이해하는 것은 중요하다

컴파일 시스템은 어떻게 동작할까?
■ 프로그램 성능 최적화하기 : C 프로그램 작성 시 기계어 수준 코드에 대한 기본적인 이해를 할 필요가 있으며, 컴파일러가 어떻게 C 문장들을 기계어 코드로 번역하는지 알 필요가 있다.
■ 링크 에러 이해하기 : 가장 당혹스러운 프로그래밍 에러는 링커의 동작과 관련되어 있으며, 큰 규모의 소프트웨어 시스템을 빌드할 경우 더욱 그렇다.
■ 보안 약점 피하기 : 오랫동안 버퍼 오버플로우 취약성이 인터넷과 네트워크상의 보안 약점의 주요 원인으로 설명되었다. 이것은 프로그래머들이 신뢰할 수 없는 곳에

서 획득한 데이터의 양과 형태를 주의 깊게 제한해야 할 필요를 거의 인식하지 못하기 때문에 생겨난다.

1.4 프로세서는 메모리에 저장된 인스트럭션을 읽고 해석한다

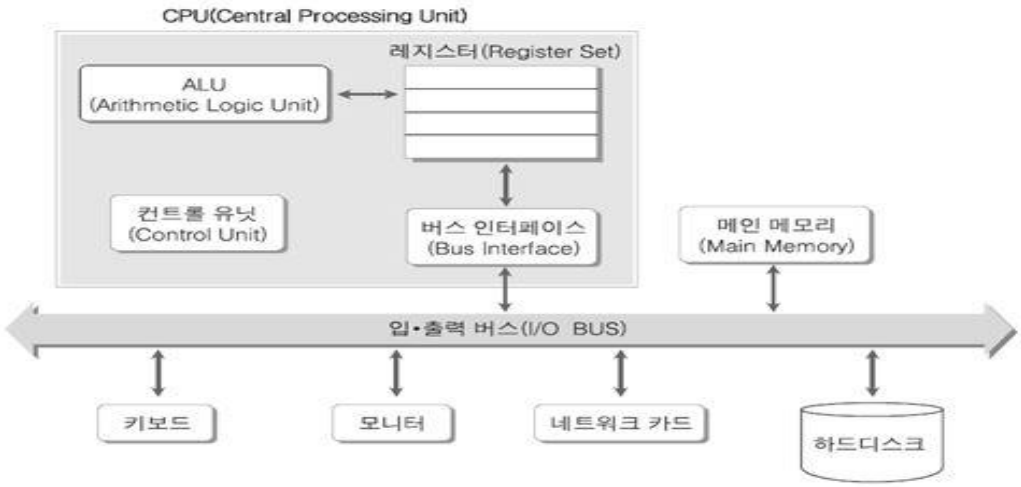
hello.c 소스 프로그램은 컴파일 시스템에 의해 hello라는 실행 가능한 목적파일로 번역되어 디스크에 저장되었다. 이 실행파일을 유닉스 시스템에서 실행하기 위해서 쉘이라는 응용프로그램에 그 이름을 입력한다.

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

쉘은 커맨드라인 인터프리터로 프롬프트를 출력하고 명령어 라인을 입력 받아 실행한다. hello 프로그램은 메시지를 화면에 출력하고 종료한다. 쉘은 프롬프트를 출력해주고 다음 입력 명령어 라인을 기다린다.

1.4.1 시스템의 하드웨어 조직

hello 프로그램을 실행할 때 무슨 일이 일어나는지 설명하기 위해서는 아래와 같은 전형적인 시스템에서의 하드웨어 조직을 이해해야 한다.



■ 버스(Buses) : 시스템 내를 관통하는 전기적 배선군이며, 컴포넌트들 간에 바이트 정보들을 전송한다. 버스는 일반적으로 워드(word)라고 하는 고정 크기의 바이트 단위로 데이터를 전송하도록 설계된다. 대부분의 컴퓨터들은 4바이트 또는 8바이트 워드 크기를 갖는다.

■ 입출력 장치 ; 시스템과 외부세계와의 연결을 담당한다. 입력용 키보드와 마우스, 출력용 디스플레이, 저장을 위한 디스크 드라이브 등이 있다. 각 입출력 장치는 입출력 버스와 컨트롤러나 어댑터를 통해 연결된다. 이 두 장치의 차이는 패키징에 있다.
■ 메인 메모리 ; 프로세서가 프로그램을 실행하는 동안 데이터와 프로그램을 모두 저장하는 임시 저장장치다. 메인 메모리는 물리적으로 DRAM 칩들로 구성되어 있다.
■ 프로세서 : 메인 메모리에 저장된 인스트럭션들을 실행하는 엔진이다. 프로세서의 중심에는 워드 크기의 저장장치인 PC가 있다. 시스템에 전원이 끊어질 때까지 프로세서는 PC가 가리키는 곳의 인스트럭션을 반복적으로 실행하고 PC값이 다음 인스트럭션의 위치를 가리키도록 업데이트 한다. CPU가 실행하는 단순한 작업의 예로는 적재(Load), 저장(Store), 작업(Operate), 점프(Jump)가 있다.

1.4.2 hello 프로그램의 실행

처음에 쉘 프로그램은 자신의 인스트럭션을 실행하면서 사용자가 명령을 입력하기를 기다린다. 사용자가 “:\hello"를 입력하면 쉘은 각각의 문자를 레지스터에 읽어 들인 후, 메모리에 저장한다. 엔터 키를 누르면 쉘은 명령 입력을 끝마쳤다는 것을 알게 된다. 그러면 쉘은 실행파일 hello를 디스크에서 메인 메모리로 로딩한다. 데이터 부분은 최종적으로 출력되는 문자 스트링인 "hello, world\n”을 포함한다. 데이터는 DMA(직접 메모리 접근)를 이용하여 프로세서를 거치지 않고 디스크에서 메인 메모리로 직접 이동한다.

hello 목적파일의 코드와 데이터가 메모리에 적재된 후, 프로세서는 hello 프로그램의 main 루틴의 기계어 인스트럭션을 실행하기 시작한다. 이 인스트럭션들은 “hello, world\n" 스트링을 메모리로부터 레지스터 파일로 복사한다. 거기로부터 디스플레이 장치로 전송하여 화면에 글자들이 표시된다.

시스템 프로그래밍을 왜 공부하는가?

시스템 프로그래밍을 공부하기에 앞서 컴퓨터 시스템이 무엇인지에 대해 먼저 알아야 할 것 같다. 책 첫 페이지에 서술되어 있듯이 컴퓨터 시스템은 하드웨어와 소프트웨어로 구성되며, 이들이 함께 작동하여 응용프로그램을 실행한다. 모든 컴퓨터 시스템들은 유사한 기능을 수행하는 유사한 하드웨어와 소프트웨어 컴포넌트를 가지고 있다. 처음 보는 단어들이 나와 생소하게 정의되어 있는 것 같기도 하지만 간단히 말하면 시스템을 제어하는 프로그램, 그것이 시스템 프로그램이다.

솔직히 첫 수업 시간 때에는 그동안 배우지 않은 것들이어서, 조금 낯설어서 겁을 먹기도 했다. 그 전에는 코딩 위주의 수업을 들어 수업시간에 visual studio를 사용하는 시간이 많았다. 그래서 C언어나 C++언어처럼 새로운 언어들을 접함에도 불구하고 나름 재미있었다. 그에 반해 시스템 프로그램은 그것들의 이론 같다. 어쩌면 이러한 이유로 더 겁을 먹지 않았나 생각된다.

반대로 high-level language를 기계어로 바꾸는 과정처럼 그동안 당연한 것이라고 생각했던 것들을 좀 더 자세히 파고드니까 그런 부분은 신기하다. 그리고 에러를 다루는 능력도 배우는데, visual studio를 쓰다 보면 에러가 난 경우 수정이 쉽게 가능할 때가 있는 반면 그렇지 않은 경우도 있다. 따라서 시스템 프로그래밍을 열심히 공부하면 내가 몰랐던 에러들도 많이 알 수 있을 것 같다.

시스템 프로그래밍을 통해 하드웨어와 소프트웨어가 어떻게 통합되어 동작하는지도 배운다. 그러기 위해서는 컴파일러, 어셈블러, 링커, 로더 등 대표적 시스템 소프트웨어에 대해 정확히 이해를 하고 있어야 한다. 또한 이 과정에서 컴퓨터 시스템을 다각도에서 추상화할 수 있는 능력을 키우고 추상화 간에 인터페이스를 이해하는 것이 중요하다. 이러한 효과적 기법을 사용해 우리가 프로그램을 짜는데 코드들을 최적화하는 방법을 배운다.

앞으로 한학기동안 이러한 배움을 통해 더 효과적인 프로그램을 개발할 수 있을 것 같다. 추가적으로 시스템이 응용프로그램에 미치는 영향도 잘 이해해 이전까지와는 다른 지식으로 프로그램을 개발하도록 해야겠다.

1.7.1 프로세스

프로세스는 실행 중인 프로그램에 대한 운영체제의 추상화다. 대부분 프로세스들은 동일한 시스템에서 동시에 실행될 수 있으며, 각 프로세스는 하드웨어를 배타적으로 사용하는 것처럼 느낀다. 여기서 동시에라는 말은 한 프로세스의 인스트럭션들이 다른 프로세스의 인스트럭션들과 섞인다는 것을 뜻한다. 대부분 시스템에서 프로세스를 실행할 CPU의 숫자보다 더 많은 프로세스들이 존재한다. 과거에는 한 번에 한 개의 프로그램만 실행할 수 있었지만, 요즘 멀티코어 프로세서들은 여러 개의 프로그램을 동시에 실행할 수 있다.

운영체제는 문맥 전환이라는 방법을 사용해 교차실행을 수행한다. 또한 운영체제는 프로세스가 실행하는 데 필요한 모든 상태정보의 변화를 추적한다. 이 컨텍스트라고 부르는 상태정보는 PC, 레지스터 파일, 메인 메모리의 현재 값을 포함한다.

hello 프로그램을 실행하라는 명령을 받으면, 쉘은 시스템 콜이라는 특수 함수를 호출하여 운영체제로 제어권을 넘겨준다. 운영체제는 쉘의 컨텍스트를 저장하고 새로운 hello 프로세스와 컨텍스트를 생성한다. 그 뒤 제어권을 새 hello 프로세스로 넘겨준다.

프로세스 추상화를 구현하기 위해서는 저수준의 하드웨어와 운영체제 소프트웨어가 함께 긴밀하게 협력해야 한다.

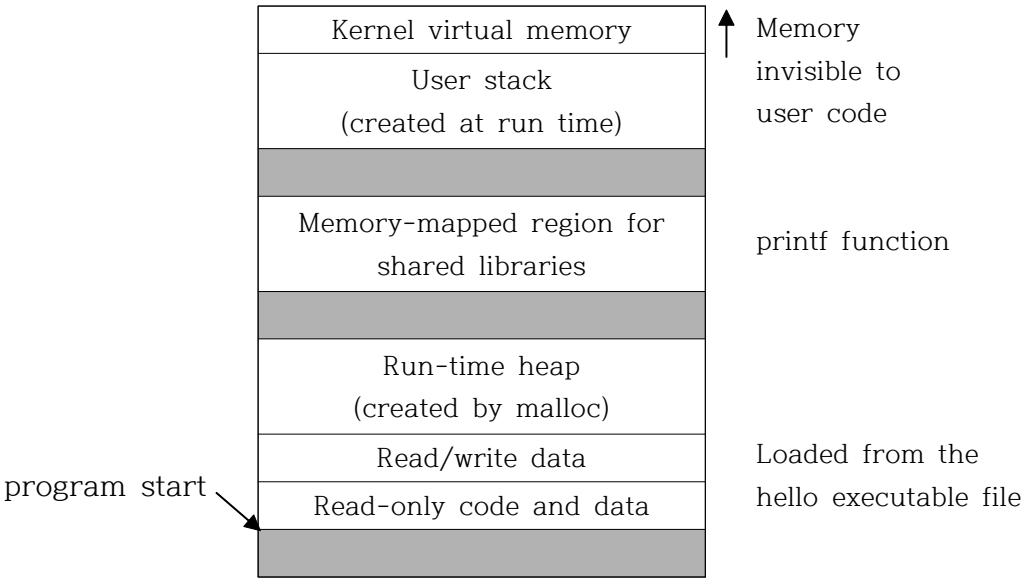
1.7.2 쓰레드(Thread)

프로세스가 한 개의 제어흐름을 갖는 것처럼 보이지만, 최근의 시스템에서는 프로세스가 실제로 쓰레드라고 하는 다수의 실행 유닛으로 구성되어 있다. 각각의 쓰레드는 해당 프로세스의 컨텍스트에서 실행되며 같은 코드와 전역 데이터를 공유한다.

1.7.3 가상메모리

가상메모리는 각 프로세스들이 메인 메모리 전체를 독점적으로 사용하고 있는 것 같은 환상을 주는 추상화이다. 각각의 프로세스는 가상주소 공간이라고 하는 균일한 메모리의 모습을 갖는다. 이 주소공간의 하위 영역에는 사용자 프로세스의 코드와 데이터를 저장한다.

아래의 그림은 리눅스 프로세스들의 가상주소 공간을 나타낸 것이다. 그림에서 위쪽으로 갈수록 주소가 증가한다.



■ 프로그램 코드와 데이터 : 코드는 모든 프로세스들이 같은 고정 주소에서 시작한다. 코드와 데이터 영역은 실행 가능 목적파일인 hello로부터 직접 초기화된다.
■ 힙(Heap) : 코드와 데이터 영역 다음으로 런타임 힙이 따라온다. 힙은 프로세스가 실행되면서 C 표준함수인 malloc이나 free를 호출하면서 런타임에 동적으로 그 크기가 늘었다 줄었다 한다.
■ 공유 라이브러리 : 주소공간의 중간 부근에 공유 라이브러리의 코드와 데이터를 저장하는 영역이 있다.
■ 스택(Stack) : 사용자 가상메모리 공간의 맨 위에 컴파일러가 함수 호출을 구현하기 위해 사용하는 사용자 스택이 위치한다. 힙처럼 사용자 스택은 프로그램이 실행되는 동안에 늘어났다 줄어들었다 한다. 스택은 함수 호출 시 커지고, 함수 리턴 시 줄어든다.
■ 커널 가상메모리 : 주소공간의 맨 윗부분은 커널을 위해 예약되어 있다. 응용프로그램들은 이 영역의 내용을 읽거나 쓰는 것이 금지되어 있고, 커널 코드 내에 정의된 함수를 직접 호출하는 것도 금지되어 있다. 그러나 이러한 작업을 수행하기 위해서는 커널을 호출해야 한다.

1.7.4 파일

파일은 단지 연속된 바이트들이다. 디스크, 키보드, 디스플레이, 네트워크까지 포함하는 모든 입출력 장치는 파일로 모델링한다. 시스템의 모든 입출력은 유닉스 I/O라는 시스템 콜들을 이용하여 파일을 읽고 쓰는 형태로 이루어진다.