Chap1. A Tour of Computer Systems

컴퓨터 시스템은 하드웨어와 시스템 소프트웨어로 구성되고 이 위에 다양한 응용 프로그램들이 수행된다. 모든 컴퓨터들은 유사한 구성요소를 가지고 있기 때문에 컴퓨터를 구성하는 요소들이 어떻게 동작하고 프로그램의 성능과 정확성에 도움을 주는지 공부한다면 앞으로도 좋은 프로그램을 개발할 수 있을 것이다. 따라서 이 책은 컴퓨터 시스템에 대한 이해를 통해 우리가 에러와 위험성에서 벗어나 최적화된 프로그램 개발을 할 수 있도록 실용적인 기술들에 대해 알려준다.

1.1 정보는 비트와 컨텍스트로 구성된다.

프로그래머가 편집기를 사용하여 코드를 작성하면 이를 소스파일(소스 프로그램)이라 한다. 만약 C 언어로 hello라는 프로그램을 작성했다면 그 소스파일 이름은 hello.c 라는 텍스트 파일이 저장되는 것이다. 소스프로그램은 항상 0,1의 연속된 비트들로 이루어지는데 비트 8개가 모여 바이트가 되고 텍스트의 각 문자는 이 바이트로 나타낸다.

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

#	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

Figure 1.2: **The ASCII text representation of `hello.c`.**

그렇다면 텍스트 문자를 어떻게 바이트로 나타낼까? 우리는 아스키 표준이라는 것을 사용하여 문자 하나하나를 바이트로 나타낸다. 아스키 문자로만 이루어진 파일을 텍스 파일, 그 외 다른 모든 파일은 바이너리 파일이라고 한다.

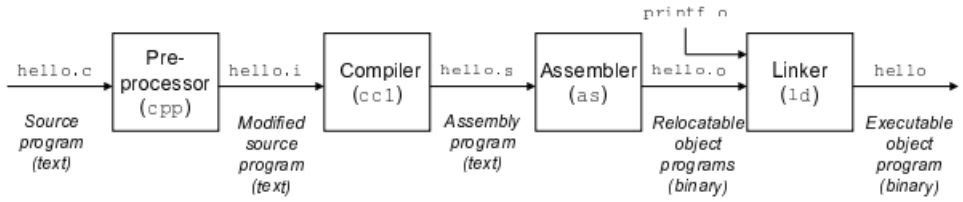
소스프로그램 뿐만 아니라 모든 시스템 내부의 정보들(메모리상 프로그램/데이터 등)은 비트들로 표시된다. 이런 비트와 바이트들을 구분하는 유일한 방법은 컨텍스트에 의해서이다. 동일한 일련의 바이트가 있어도 컨텍스트 즉, 내가 해당 바이트를 구분하기 위해서 혹은 어떤 작업을 수행하기 위해 제공하는 추가적인 정보가 함께 주어진다면 서로 다른 객체도 구분될 수 있다. *(컨텍스트*

의 개념은 따로 찾아보고 이해한 대로 작성해 봤습니다.)

1.2 프로그램은 다른 프로그램에 의해 다른 형태로 번역된다.

우리가 소스 프로그램을 작성할 때는 인간이 바로 읽고 이해할 수 있도록 High level language로 작성된다. 그런데 이 프로그램이 실행되려면 각 문장들이 더 낮은 단계의 언어인 기계 명령어로 변역 되어야 한다. 그리고 이 기계 명령어는 실행가능한 object 프로그램으로 합쳐져 바이너리 디스크 파일로 저장된다. 즉, 사람이 이해할 수 있는 언어로 작성된 프로그램이 컴퓨터가 이해할 수 있게끔 번역되는 4가지 단계(전처리기, 컴파일러, 어셈블러, 링커)가 있고 이 단계를 컴파일 시스템이라고 한다.

각 단계를 살펴보면 이러하다.



전처리 단계: 우리가 프로그램을 작성하면 .c 를 같는 파일이 생긴다고 했는데 이 파일이 전처리기(cpp)를 지나며 필요한 시스템 헤더 파일의 삽입을 지시하도록 수정되어 .i 로 끝나는 C프로그램이 생성된다.

컴파일 단계: 컴파일러(cc1)는 텍스트 파일을 어셈블리 언어라는 중간 수준의 언어로 번역하여 어셈블러 프로그램이 되고 이를 .s 파일에 저장한다.

어셈블러 단계: 어셈블러(as)가 어셈블리 파일을 기계 명령어로 번역하여 재배치 가능한 object 프로그램을 만든다. .o 형태의 파일이 만들어진다.

링크 단계: 재배치 가능한 object 프로그램과 표준 C 라이브러리에 있는 함수 등이 링커 프로그램(ld)에 의해서 합쳐져 실행가능한(수행 가능한) 목적 파일이 되어 메모리에 올라가고 비로서 실행 될 수 있게 된다. 표준 C 라이브러리에 있는 함수들은 컴파일 시, 이미 별도의 목적 파일에 들어가게 되고 링크 단계에서는 이런 목적 파일들을 합치는 작업이다.

1.3 컴파일 시스템의 동작을 이해하는 것은 중요하다.

컴파일러의 내부 동작을 세세하게 알 필요는 없다. 하지만 컴파일 시스템이 어떻게 동작하는지, C 문장들이 어떻게 기계어로 번역되는지에 대한 전반적인 이해는 프로그램 작성을 올바르고 효율적

으로 할 수 있게 해준다. 같은 반복문이라는 기능을 하더라도 while문을 쓰는 것과 for문을 쓰는 것이 공간 활용이나 속도 등과 같은 성능에 영향을 줄 수도 있다. 또한 링크 에러와 같이 목적 파일들을 합치는 과정에서 일어나는 에러는 단순 오타로 인한 에러와는 다르게 컴파일 시스템의 동작에 대한 개념을 알고 있어야 헤쳐 나갈 수 있다. 뿐만 아니라 우리가 신뢰할 수 없는 데이터가 어떤 형태로 얼마나 우리 컴퓨터 메모리에 쌓이는지에 대한 이해가 있어야 한다. 이에 대한 고려가 없으면 버퍼 오버플로우(Buffer overflow) 같은 일이 발생하여 인터넷이나 네트워크 상의 보안 문제 원인을 제공하게 될지도 모른다.

1.4 프로세서는 메모리에 저장된 기계 명령어를 읽고 해석한다.

컴파일을 통해 실행가능한 파일이 디스크에 저장되면 그 프로그램을 유닉스 환경에서 실행하기 위해서는 쉘이라는 응용프로그램에 파일의 이름을 입력해야 한다. 명령어를 기다리는 쉘에 파일 이름을 입력하면 쉘은 프로그램을 로딩하고 실행한 뒤 종료를 기다린다. 그런데 쉘에 입력한 프로그램이 실행 될 때 하드웨어에서는 무슨일이 일어나고 있는걸까? 우리는 이것을 이해하기 위해 하드웨어 조직을 이해할 필요가 있다.

먼저 하드웨어는 크게 버스(Buses), 입출력장치, 메인메모리, CPU로 나눌 수 있다.

버스 : 하드웨어 내부의 구성요소들 간에 바이트 정보들 마치 버스처럼 실어 날라주고 전송한다. 보통 워드(word)라고 하는 고정된 크기의 바이트 단위로 데이터를 전송하는데 대부분 4Byte 혹은 8Byte의 워드 크기를 갖는다.

입출력장치 : 시스템 내부와 외부세계를 연결한다. 키보드나 마우스 등은 외부에서 시스템 내부로 정보를 입력 받는거고, 모니터나 프린터 등은 시스템 내부에서 외부로 정보를 출력한다. 각 입출력 장치는 입출력 버스와 컨트롤러/어댑터를 통해 연결된다. 컨트롤러는 디바이스 자체가 칩셋이거나 마더보드에 바로 부착되지만 어댑터는 마더보드의 슬롯 카드 형태로 장착된 차이가 있다.

메인 메모리 : 프로세서가 프로그램을 실행하는 동안 데이터와 프로그램을 모두 저장하는 임시 저장장소이다. 메인 메모리는 DRAM 칩셋으로 구성되어 있고, 연속적인 바이트들의 배열로 이루어져 있는데 각각의 배열은 0부터 시작해서 인덱스 주소를 가지고 있다. 여기에 프로그램을 구성하는 기계 명령어들이 저장된다.

CPU : 주처리장치인 CPU 혹은 프로세스*(현재 수행중인 프로그램을 가르킨다)*는 메인 메모리에 저장된 기계 명령어를 해석하고 처리하는 뇌와 같은 부분이다. CPU의 중심에는 레지스터라는 빠르게 데이터에 접근할 수 있는 다양한 저장장치들이 있다. 이 책에서 소개하고 있는 주요 레지스터는 PC인데 PC는 프로세서가 수행할 기계 명령어의 메모리 주소를 미리 저장하고 있는 저장장치이다. 마치 사장님이 더 빠르고 효과적으로 일을 처리할 수 있도록 다음 업무를 체크하고 알려주는 유능하고 충실한 비서 같다. PC뿐 아니라 각각의 고유 이름을 갖는 워드 크기의 레지스터 집합(레지스터 파일)도 있다. PC와 레지스터 파일등을 이용해 저장하고 있던 기계 명령어들은 ALU

같은 수식/논리 처리기에서 연산이나 새 데이터/주소 값 계산등과 같은 처리 과정을 거치고 결과가 출력된다.

적재(Load): 메인 메모리 -> 레지스터 (한 바이트/워드를 이전 값 위에 덮어씀)

저장(Store): 레지스터 -> 메인 메모리 (한 바이트/워드를 이전 값 위에 덮어씀)

작업(Operate): 두 레지스터 값을 ALU에 복사-> 수식연산 -> 레지스터에 저장(덮어쓰기)

점프(Jump): 기계 명령어 자신으로부터 한 개의 워드를 추출하고 이를 PC에 복사(덮어쓰기)

따라서 프로그램 실행은 다음과 같이 진행된다. (hello 프로그램 실행 기준)

1. 쉘 프로그램이 명령어 입력 기다림(엔터 누르면 명령 입력 끝)
2. 요청 받은 디스크의 파일 코드와 데이터를 복사하여 메인 메모리로 로딩

   ("직접 메모리 접근 방식" 사용해 프로세서 거치지 않음)

   실행가능한 목적 파일의 코드와 데이터가 메모리에 적재된 후, 명령어 수행
3. 수행 결과는 메인 메모리로 이동 후, 레지스터에 복사되고, 출력장치를 통해 출력

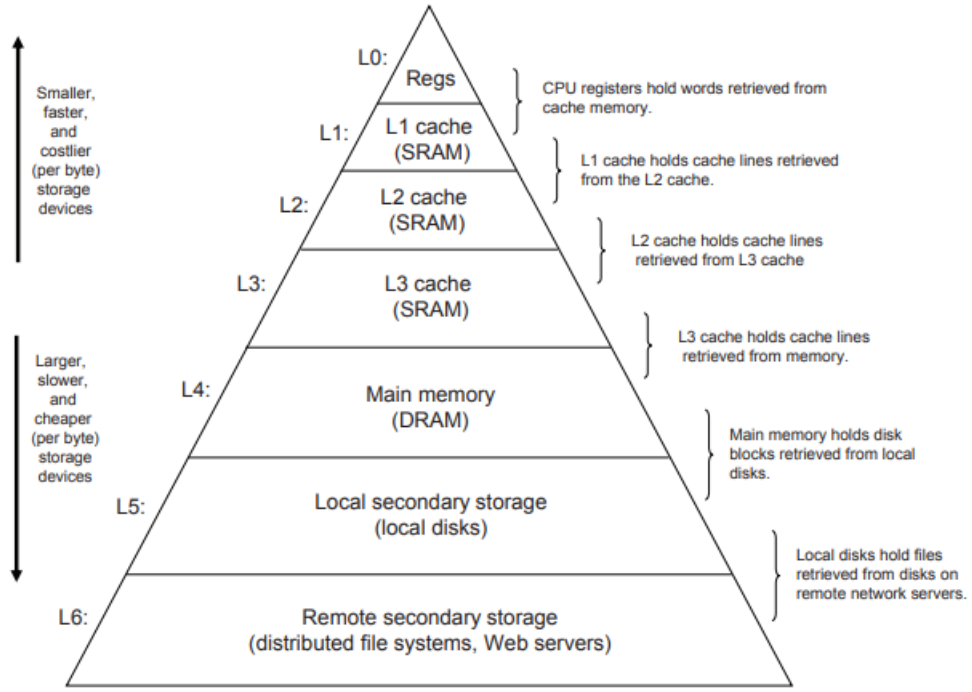
1.5 캐시가 중요하다

위의 프로그램 실행되기까지 메모리와 CPU 등에서 어떤 일이 일어나는지 살펴보면서 데이터를 이동시키는 일에는 많은 과정이 있고 그만큼 시간도 많이 걸린다는 것을 깨닫게 된다. 위의 예제에서 hello프로그램은 하드디스크에 존재했고 이를 메인 메모리로 옮기기 위해서 복사하는 과정들이 필요한데 이를 가능한 빠르게 하는 것이 시스템 설계자들의 주요 목적이자 컴퓨터 시스템의 성능을 위한 일이다.

물리법칙에 의해서 큰 저장장치일수록 데이터를 읽어 들이는 속도는 느리고 시간도 오래 걸린다. 디스크-메모리-레지스터 순서에 따라 크기는 작고 데이터는 더 빠르게 읽어 들인다. 특히 중앙처리장치와 메모리 간의 속도 격차는 지속적으로 벌어지고 있기 때문에 이 둘의 중간에 작고 빠른 캐시 메모리라는 저장장치를 두기로 했다. **캐시에는 단기간에 필요한 가능성이 높은 정보들을 임시로 저장해 놓고 CPU나 프로세스에서 이를 요구하면 빠르게 바로 데이터를 전달**한다. 1.6에서 배우겠지만 캐시는 계층구조로 이루어져 있는 L1~3처럼 숫자가 커질수록 용량이 커지고 속도는 느려진다. L1과 L2는 SRAM이라는 하드웨어 기술을 이용해 구현하기도 했다. 캐시메모리는 지엽적인 영역의 코드와 데이터에 접근하여 빠른 수행 효과를 얻는 효과적인 아이디어이다.

1.6 계층구조를 이루는 저장장치

위에서 배운 캐시라는 아이디어처럼 보다 작고 빠른 장치를 크고 느린 장치 사이에 끼워 넣는 개념이 보편적인 아이디어라고 한다. 따라서 컴퓨터 시스템의 저장장치들도 모두 계층구조로 표현할 수 있다. CPU-캐시1-캐시2-캐시3-메인 메모리-로컬 디스크-원격외부저장장치까지. 뒤로 갈 수록 저장장치의 용량은 커지고 속도는 느리 지며 값은 싸진다.



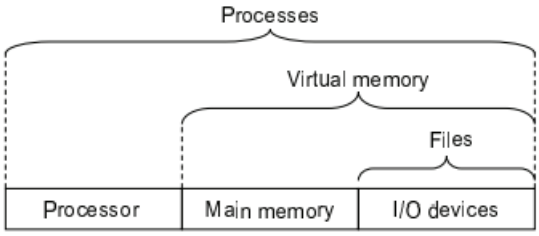
이 계층구조의 특징은 한 레벨의 저장장치가 다음 하위 레벨 저장장치의 캐시 역할을 한다는 것이다. L1은 L2의 캐시 역할을 하고, L3는 메인 메모리의 캐시이고, 메인 메모리는 로컬 디스크의 캐시 역할을 한다. 원격 네트워크 시스템을 사용할 경우, 로컬디스크가 캐시 역할을 하여 원격으로 가져온 파일들을 보관한다.

1.7 운영체제는 하드웨어를 관리한다.

컴퓨터 시스템은 하드웨어가 있고, 그 위에 운영체제 라는 서비스 관리자가 있고, 그 위에 여러가지 프로그램들이 깔린다. 그리고 그 프로그램은 절대 바로 하드웨어에 들어가 기계 명령어를 수행하는 것이 아니라 운영체제의 관리하에 수행된다. 왜 꼭 운영체제가 중간에 끼어들어야 하는 것일까? 여기에는 중요한 두 가지 목적이 있다.

1) 프로그램들이 제멋대로 동작해 하드웨어를 잘못 사용하는 것을 막기 위해서
2) 프로그램들이 단순하고 균일한 매카니즘을 사용해 저수준의 복잡한 하드웨어 장치들을 조작할 수 있게 하기 위해서

만약 운영체제가 중간에서 이런 것들을 제어해주고 관리해주지 않는다면 프로그램은 멋대로 하드웨어를 사용해 메모리 공간이 금방 부족해져 버리거나 프로그램의 수행 결과가 복잡한 하드웨어 시스템에서 길을 잃어 출력되지 않을 수도 있다. 따라서 이 두가지 목표 달성을 위해 하드웨어를 추상화한 개념을 사용한 운영체제의 도움으로 응용프로그램과 하드웨어가 적절한 조화를 이뤄 원하는 결과가 잘 수행될 수 있게 해준다.



<하드웨어의 추상화>

프로세스

프로세스는 실행중인 프로그램이다. 우리는 컴퓨터를 사용하며 여러 프로그램을 실행하는데 이는 마치 여러 개의 프로세스가 하드웨어를 쪼개어 각각 차지하고 동시에 프로그램을 실행하는 것처럼 보인다. 하지만 실제로는 동시에 진행되는 것이 아니라 시분할로 CPU 공간을 할당 받아 사용하는 것이고, 위에서 말했듯이 하드웨어를 여러 개로 나눠서 사용하는 것처럼 보이게 하는 거이 운영체제가 제공하는 추상화인 것이다.

운영체제는 프로세스가 실행하는데 필요한 모든 상태정보들의 변화를 추적한다. 한 프로세스에서 쉘이 실행되고 있다고 가정해보자. 그런데 이 쉘에 프로그램을 실행하고자 하는 명령을 입력하면 쉘은 "시스템 콜"이라는 특수한 함수를 호출하여 운영체제로 제어권을 넘겨준다. 그리고 쉘의 상태정보 등을 저장하고, 실행하고자 하는 프로그램의 프로세스와 상태정보를 생성한 후, 제어권을 이 프로세스에게 넘겨준다. 그리고 프로그램이 수행되고, 종료되면 운영체제는 아까 저장해 두었던 쉘의 상태정보를 복구시키고 다시 쉘에게 제어권을 주어 다음 명령어를 받을 수 있게 한다.

이렇게 프로세스의 전환은 커널이라는 운영체제의 핵심 부분에서 관리된다. 커널은 메모리에 상주한다. 위에서 말했듯이 프로그램이 운영체제에 작업을 요청하면 커널은 파일 일기/쓰기 같은 특정 시스템 콜을 실행해 제어권을 일단 커널이 가져오고 커널에서 필요한 작업 수행 후 프로그램에게 제어권을 넘겨준다. 커널은 어떤 별도의 프로세스가 아닌 모든 프로세스를 관리하기 위해 컴퓨터 시스템이 이용하는 코드와 자료 구조의 집합이다.

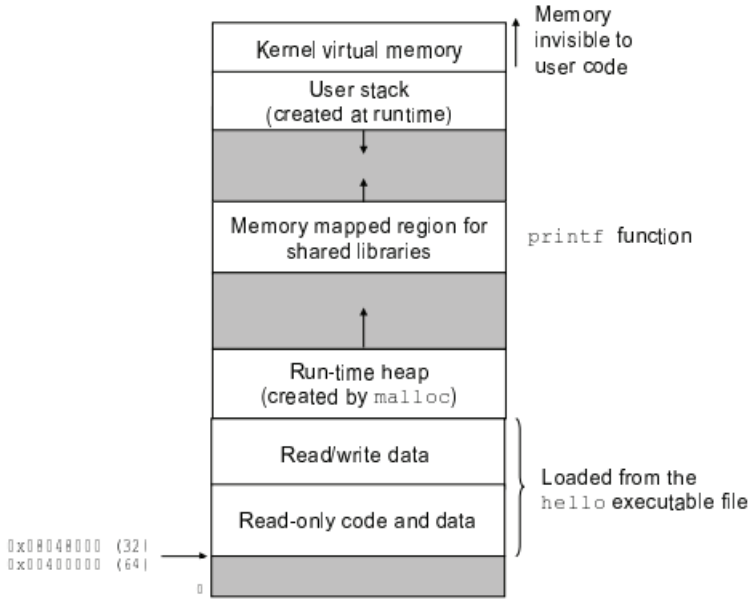
쓰레드(Thread)

프로세스가 한 개의 제어흐름을 관장한다고 생각할 수 있지만 실제로는 프로세스 안에 쓰레드 라는 다수의 실행 유닛이 실행된다. 쓰레드는 해당 프로세스의 컨텍스트에서 실행되며 동일한 코드

와 전역데이터를 공유한다. 이전에 진행했던 파이썬 게임 만들기 프로젝트에서 하나의 동작을 하는 기능을 만드는데 그 안에서 세부적으로 여러 타임아웃 기능을 구현하기 위해서 쓰레드를 사용하곤 했던 기억이 난다. 이것처럼 전체적으로는 동일한 흐름제어 안에서 쓰레드가 작동하기 때문에 데이터의 공유가 더 쉽다. 다중 프로세서를 활용한 다중 쓰레딩도 프로그램 실행을 빠르게 하는 방법이다.

가상메모리

한 프로세스가 마치 하드웨어를 독점하고 있듯이 보이는 것처럼 각 프로세스들은 메인 메모리를 독점적으로 사용하는 것처럼 보인다. 하지만 이 또한 사실은 가상 메모리라는 개념을 사용해 추상화를 진행한 것이다.



그림처럼 리눅스 프로세스들의 가상주소 공간이 있고, 최상위 주소공간에는 운영체제 코드와 데이터가, 하위 영역에는 사용자가 프로세스의 코드와 데이터를 저장한다. (* 위쪽으로 갈수록 주소 증가)

커널 가상메모리: 오로지 커널을 위하여 예약되어 있는 공간이다. 이 영역의 함수를 호출하는 것 또한 함부로 할 수 없고, 반드시 커널을 호출해야 한다.

스택(Stack): 컴파일러가 함수호출을 구현하기 위해 사용되는 공간. 프로그램이 실행되는 동안 동적으로 늘어났다 줄어든다. (함수 호출 시, 늘어남)

공유 라이브러리: 표준 라이브러리나 기타 등등 라이브러리의 코드와 데이터를 저장하는 영역이다.

힙(Heap) : 크기가 고정되어 있지 않고 실행되면서 동적으로 그 크기가 늘어나는 영역이다. C언어

에서 사용하는 malloc 같은 동적 메모리 할당 함수를 사용하면 이 공간에 영역이 생겼다 없어진다 하는 것이다.

프로그램 코드/데이터: 프로그램의 코드와 데이터가 저장되는 것인데 코드의 경우 모든 프로세스들은 같은 고정 주소에서 시작하고, 그 다음에 데이터들이 따라온다. 이 영역은 실행 가능한 목적 파일일 때부터 초기화 된다.

파일

파일은 그저 연속된 바이트들이다. 입출력 장치를 추상화한 것이 바로 파일인데, 컴퓨터 시스템에서 입출력을 진행하면 유닉스 I/O 라는 시스템 콜들을 이용해 파일을 읽고 쓴다. 매우 간단해 보이지만 시스템에 존재하는 다양한 입출력 장치들에게 통일성을 부여하는 아주 강력한 개념이다.

이 수업에서 배우고 싶은 목표

한창 입시를 할 당시 목표했던 진로와는 전혀 다른 길을 선택한 나에게 이 분야의 가장 큰 난제는 하드웨어, 혹은 하드웨어의 느낌이 나는 이론들이었다. 하지만 이진수가 무엇인지도 잘 몰랐던 내가 이 길을 선택한 이유는 이 난제를 해결하고 이 분야를 어느정도 정복하고 싶다고 생각했기 때문이다. 물론 전문가 수준의 정복을 원하는 것은 아니지만 적어도 하드웨어의 구조, 소프트웨어가 하드웨어에서 동작하는 대략적인 원리, 운영체제의 개념, 운영체제가 하는 일 과 같은 내용들을 확실히 이해하고 누군가가 나에게 이에 대해서 물어보면 쉽고 재미있게 가르쳐줄 수 있는 수준에 도달하고 싶다. 또한 저번 학기에 3학년 과목인 네트워크 수업을 수강한 적이 있는데 내가 모르는 개념들이 너무 많아 따라가기 힘들다는 생각이 들었고, 내가 많이 부족하다는 생각에 속이 상하기도 했다. 그런데 지금까지 시스템 프로그래밍 수업을 들으면서 네트워크 수업에서 들었던 프로세스, 스레드, IP/TCP와 같은 개념들이 등장하는 것을 발견했고 차근차근 배워가니 공부할 만하다는 생각이 들면서 굉장히 즐거웠다. 그래서 이번 학기 시스템 프로그래밍 수업을 잘 수강한다면 내년에 3,4학년에 되어서 네트워크 수업을 포함해 다른 수업까지 연계하여 많은 지식과 경험을 쌓을 수 있을 것이라고 기대하고 있다. 마지막으로 개인적인 소소한 기대가 한 가지 있다. 윈도우 컴퓨터만 사용하던 내가 최근에 맥os가 깔린 맥북을 구입해 새로운 운영체제를 경험해 보고 싶다는 생각을 강하게 하게 되었는데 보통 윈도우를 사용하다 맥북을 사용하면 어려움을 호소하는 사람이 많다고 한다. 하지만 나는 이 수업을 듣고 맥북을 구입하면 리눅스 명령어도 쉽게 사용하고 새로 경험하는 운영체제 시스템을 사용하면서 수업 시간에 배웠던 것들을 떠올리며 즐길 수 있을 것 같다는 생각이 들어 많은 기대가 된다.