시스템 프로그래밍

소프트웨어학과 32151671 박민혁

컴퓨터 시스템은 하드웨어와 시스템 소프트웨어로 구성되며, 이들이 함께 작동하여 응용프로그램을 실행한다.

1.1 정보는 비트와 컨텍스트로 이루어진다.

```
1. #include <stdio.h>
2.
3. int main()
4. {
5.         pirntf(“hello, world₩n”);
6.         return 0;
7. }
```

hello 프로그램은 프로그래머가 에디터로 작성한 소스 프로그램으로 생명을 시작하면 hello.c라는 텍스트 파일로 저장된다. 소스 프로그램은 0 또는 1로 표시 되는 비트들의 연속이며 바이트라는 8비트 단위로 구성된다. 각 바이트는 프로그램의 텍스트 문자를 나타낸다.

hello.c의 아스키 텍스트로 표시.

```
35 105 110 99 708 100 101 32 60 115 116 100 105 111 46 104 62 10
10
105 110 116 32 109 97 105 110 40 41 10
123 10
32 32 32 32 112 114 105 110 116 102 40 34 104 101 108 108 111 44 32 119
111 114 108 100 92 110 32 41 59 10
32 32 32 32 114 101 116 117 114 110 32 48 59 10
125 10
```

hello.c 프로그램은 연속된 바이트들로 파일에 저장된다. 각 바이트는 특정 문자에 대응되는 정수 값을 갖는다. 예를 들어 첫 번째 바이트는 35인데, 이것은 문자 ‘a'에 대응된다. 각 텍스트 라인이 눈에 보이지 않는 newline 문자 '₩n'으로 종료되는 것에 주목하고, 이 문자는 정수값 10으로 표시한다. hello.c처럼 오로지 아스키 문자들로만 이루어진 파일들은 텍스트 파일이라고 부르고 다른 모든 파일들은 바이너리 파일이라고 한다. hello.c의 표시 방법은 기본개념을 분명히 보여준다. 모든 시스템 내부의 정보-디스크 파일, 메모리상의 프로그램, 데이터, 네트워크를 전송되는 데이터는 비트들로 표시된다.

C 프로그래밍 언어의 기원

C는 1969년부터 1973년까지 벨 연구소 Dennis Riychic에 의해 개발되었다.

- c는 유닉스 운영체제와 밀접하게 연결되어 있다. c는 초기부터 유닉스를 위한 시스템 프로그램 언어로 개발 되었다. 유닉스 커널 대부분(운영체제의 핵심부분)과 지원 도구들, 그리고 라이브러리 대부분이 c로 작성되었다.
- c는 작고 간단한 언어다. 설계는 위원회가 아닌 한 명의 개발자에 의해 관리되었으며,

그 결과 군더더기가 거의 없는 깔끔하고 일관된 설계가 가능하다.

- c는 실용적 목적으로 설계되었다. c는 유닉스 운영체제를 만들기 위해 설계되었다.

1.2 프로그램은 다른 프로그램에 의해 다른 형태로 번역된다.

hello 프로그램은 인간이 그 형태로 바로 이해하고 읽을 수 있기 때문에 고급 C프로그램으로 일생을 시작한다. 그러나 시스템에서 실행시키려면 저급 기계어 인스트럭션들로 번역되어야 한다. 이 인스트럭션들은 실행가능 목적 프로그램이라고 하는 형태로 합쳐져서 바이너리 디스크 파일로 저장된다. 목적프로그램은 실행가능 목적 파일이라고도 부른다. 컴파일러 드라이버는 유닉스 시스템에서 다음과 같이 소스파일에서 오브젝트 파일로 번역한다.

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

여기서 gcc 컴파일러 드라이버는 소스파일 hello.c를 읽어서 실행파일인 hello로 번역한다. 번역은 네 단계로 실행된다. (전처리기, 컴파일러, 어셈블러, 링커) 이들을 합쳐 컴파일 시스템이라고 부른다.

- 전처리 단계: 전처리기(cpp)는 본래의 c 프로그램을 #문자로 시작하는 디렉티브에 따라 수정한다.
- 컴파일 단계: 컴파일러(cc1)는 텍스트파일 hello.i를 텍스트파일인 hello.s로 번역하며, 이 파일에는 어셈블리어 프로그램이 저장된다.

```
1   main:
2       subq    $8, %rsp
3       movl    $.LCO, %e야
4       call     puts
5       movl    $0, %eax
6       addq    $8, %rsp
7       ret
```

2~7줄에서는 한 개의 저수준 기계어 명령어를 텍스트 형태로 나타내고 있다. 어셈블리어는 여러 상위 수준 언어의 컴파일러들을 위한 공통의 출력언어를 제공하기 때문에 유용하다. 예를 들어 c와 Fortran 컴파일러는 둘 다 동일한 어셈블리어로 출력파일을 생성한다.

- 어셈블리 단계: 다음에는 어셈블러(as)가 hello.s를 기계어 인스트럭션으로 번역하고, 이들을 재배치가능 목적프로그램의 형태로 묶어서 hello.o라는 목적파일에 그 결과를 저장한다.
- 링크 단계: hello 프로그램이 c 컴파일러에서 제공하는 표준 c 라이브러리에 들어있는 printf 함수를 호출하는 것에 주목할 필요가 있다.

GNU 프로젝트

GCC는 GNU 프로젝트에서 개발된 유용한 도구들 중의 하나이다. GNU 프로젝트는 1984년에 Richard Stallman이 시작한 비과세 자선사업으로 소스코드가 어떻게 수정되고, 배포될 수 있는가에 관한 제한사항에 상관 없는 유닉스와 유사한 시스템을 개발하겠다는 야심찬 목표를 가졌다. 커널은 리눅스 프로젝트에서 별도로 개발되었다. GNU 환경은 EMACS 편집기, GCC 컴파일러, GDB 디버거, 어셈블러, 링커, 바이너리 파일 관리를 위한 유틸리티, 그

리고 다른 컴포넌트들을 포함하였다. 지원하는 언어로는 C, C++, Fortran, java, 파스칼, Object-C, Ada가 있다.

1.3 컴파일 시스템이 어떻게 동작하는지 이해하는 것은 중요하다.

- 프로그램 성능 최적화하기 : 프로그래머는 기계어 수준의 코드에 대해 기본적인 이해를 하고 있는 것이 좋고 컴파일러가 어떻게 C 문장들을 기계어 코드로 번역하는지 아는 것이 좋다. 그렇게 된다면 C 프로그램을 작성 할 때 더 나은 판단으로 프로그램을 작성할 수 있을 것이다.
  시스템 프로그래밍에서는 컴파일러가 어떻게 C 구조들을 기계어로 번역하는지를 배우고, 어떻게 C 컴파일러가 메모리 시스템에 배열을 계층적으로 저장하는지를 배워서, 프로그래머가 작성한 C 프로그램이 보다 효율적으로 실행될 수 있도록 해준다.
- 링크 에러 이해하기 : 앞으로 배울 시스템(링커가 참조를 풀어낼 수 없다고 할 때는 무엇을 의미하는지, 정적변수와 전역변수의 차이는 무엇인지, 다른 파일에 동일한 이름의 두 개의 전역변수를 정의하면 무슨 일이 일어나는지, 왜 링커와 관련된 에러들은 실행하기 전까지는 나타나지 않는지 등)을 통해 가장 당혹스러운 프로그래밍 에러인 링커의 동작을 이해하고 예방할 수 있다.
- 보안 약점 피하기 : 버퍼 오버플로우 취약성은 인터넷과 네트워크상의 보안 약점의 주요 원인으로 설명되었는데, 이는 프로그래머들이 신뢰할 수 없는 곳에서 획득한 데이터 형태를 주의 깊게 제한해야 할 필요성을 느끼지 못하기 때문에 생겨난다.
  시스템 프로그래밍 과목에서는 스택에 데이터와 제어 정보가 저장되는 방식과 그러 인해 생겨나는 영향을 배워서 안전한 프로그래밍을 할 수 있도록 한다.

1.4 프로세서는 메모리에 저장된 인스트럭션을 읽고 해석한다.

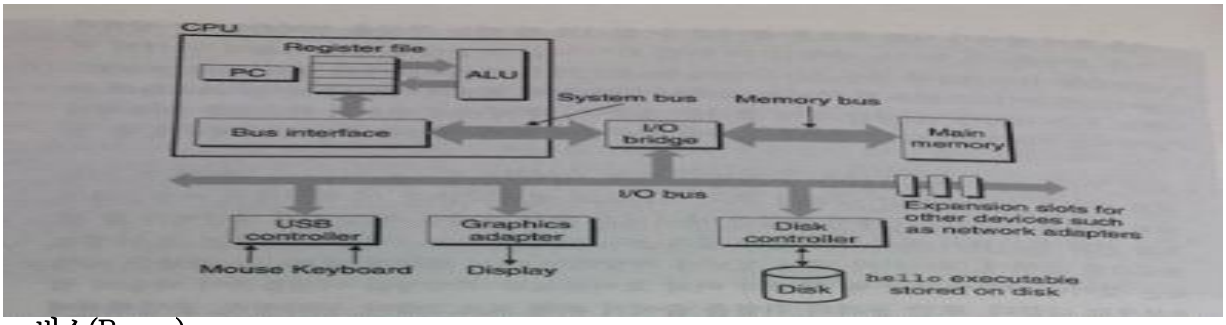
지금까지 hello.c 소스프로그램은 컴파일 시스템에 의해 hello라는 실행 가능한 목적파일로 번역되어 디스크에 저장되었다. 이 실행파일을 유닉스 시스템에서 실행하기 위해서 쉘이라는 응용프로그램에 그 이름을 입력한다.

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

이 경우에 쉘은 hello 프로그램을 로딩하고실행한 뒤에 종료를 기다린다. hello 프로그램은 메시지를 화면에 출력하고 종료한다. 쉘은 프롬프트를 출력해 주고 다음 입력 명령어 라인을 기다린다.

1.4.1 시스템의 하드웨어 조직

전형적인 시스템의 하드웨어 구성, CPU: Central Processing Unit, ALU: Arithmetic Logic Unit, PC: Program Counter, USB: Universal Serial Serial Bus. (그림은 아래 참조)



버스(Buses)

시스템 내를 관통하는 전기적 배선군을 버스(bus)라고 하며, 컴포넌트들 간에 바이트 정보들을 전송한다. 일반적으로 워드(word)라고 하는 고정된 크기의 바이트 단위로 데이터를 전송하는데, 워드를 구성하는 바이트 수는 시스템마다 다르다. 오늘날 컴퓨터는 대부분 4바이트(32비트), 8바이트(64비트)의 워드 크기를 갖는다.

입출력 장치

입출력 장치는 시스템과 외부세계를 연결해주는 것이다. 입력장치인 마우스와 키보드, 출력장치인 디스플레이, 데이터와 프로그램의 장기 저장을 위한 디스크 드라이브 등이 있다. 각 입출력 장치는 입출력 버스와 컨트롤러나 어댑터를 통해 연결되어 있다. 컨트롤러나 어댑터는 입출력 버스와 입출력 장치들 간에 정보를 전달 해주는 역할을 한다.

메인 메모리

메인 메모리는 프로세서가 프로그램을 실행하는 동안 데이터와 프로그램을 저장해 두는 임시 장치이다. 메인 메모리는 DRAM(Dynamic Random Access Memory) 칩들로 구성되어 있다. 메모리는 바이트들의 배열이고 각 배열은 각자의 주소를 가지고 있다.

프로세서

프로세서(CPU)는 메인 메모리에 저장되어 있는 인스트럭션(명령)들을 실행하는 역할을 한다. CPU 중심에는 레지스터인 PC(프로그램 카운터)가 있다.

시스템에 전원이 공급되는 순간부터 꺼질 때까지 CPU는 프로그램 카운터가 가리키는 명령어를 수행한다. CPU는 인스트럭션을 수행하고 프로그램 카운터는 다음 수행할 인스트럭션이 있는 메모리를 가리킨다.

이와 같은 단순한 동작을 반복하고, 이들은 메인 메모리, 레지스터 파일, 수식/논리 처리기(ALU) 주위를 순환한다. 레지스터 파일은 고유 이름을 갖는 워드 크기의 레지스터의 집합이고, ALU는 새 데이터와 주소 값을 계산하는 아이이다.

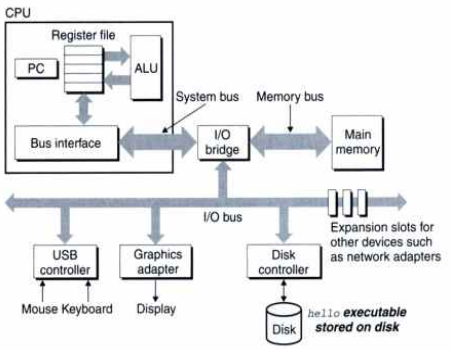
인스트럭션의 요청에 의해 CPU가 실행하는 작업의 예

- 적재(Load) : 메인 메모리에서 레지스터에 워드 또는 한 바이트 단위로 이전 값에 덮어쓰는 방식으로 복사한다.
- 저장(Store) : 레지스터에서 메인 메모리로 한 바이트 또는 워드 단위로 이전 값에 덮어쓰는 방식으로 복사한다.
- 작업(Operate) : 두 레지스터 값을 ALU로 복사한 뒤, 수식연산을 수행하고, 결과를 레지스터에 덮어쓰는 방식으로 저장한다.

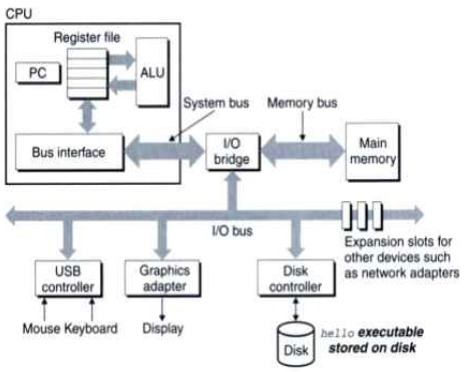
• 점프(Jump) : 인스트럭션 자신으로부터 한 개의 워드를 추출하고, 이것을 PC에 덮어쓰는 방식으로 복사한다.
프로세서가 자신의 인스트럭션 집합 구조를 단순 구현한 것처럼 보인다고 말하지만, 사실 최신 프로세서들은 프로그램의 실행 속도를 높이기 위해 훨씬 더 복잡한 방식을 사용한다. 그래서 각 기계어 인스트럭션의 효과를 설명하고, 프로세서가 실제 어떻게 구현되었는지 설명하면서 프로세서의 마이크로 구조와 인스트럭션 집합구조를 구별할 수 있게 된다.

1.4.2 hello프로그램의 실행

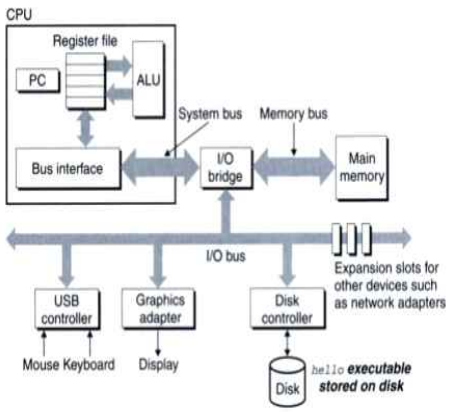
지금까지의 시스템 하드웨어 조직과 동작에 대한 간단한 이해를 바탕으로 예제 프로그램이 실행될 때 무슨일이 일어나는지 비로소 이해할 수 있었다. 여기서는 상당 부분 자세한 내용은 생략해야 하지만, 당분간은 개략적인 큰 그림에 대한 이해만으로도 충분히 이해가 가능하다.



처음에 쉘 프로그램은 자신의 인스트럭션을 실행하면서, 사용자가 명령을 입력하기를 기다린다. “₩hello"를 입력하면 쉘 프로그램은 각각의 문자를 레지스터에 읽어들인 후 그림과 같이 메모리에 저장한다.



키보드에서 엔터 키를 누르면 쉘은 명령 입력을 끝나였다는 것을 알게 된다. 그러면 쉘은 파일 내의 코드와 데이터를 복사하는 일련의 인스트럭션을 실행하여 실행파일 hello를 디스크에서 메인 메모리로 로딩한다. 데이터 부분은 최종적으로 출력되는 문자스트링인 “hello, world₩n"을 포함한다. 직접 메모리 접근이라고 알려진 기법을 이용해서 데이터는 프로세서를 거치지 않고 디스크에서 메인 메모리로 직접 이동한다. 옆에 그림에 나타나 있다.



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 복사하고 거기로부터 디스플레이 장치로 전송하여 화면에 글자들이 표시된다. 이과정은 옆에 그림과 같다.

What is the purpose of studying System Programming?

시스템 프로그래밍에 앞서 이제 군대를 전역하고 복학한 상태이다. 대학 입학전 컴퓨터에 대해 잘 몰랐다. 오류가 뜨면, 오류가 나는 대로 살아왔다. 그리고 대학 입학을 했다. 사실 나는 같은 과 사람들 보다 컴퓨터에 대한 배경지식도 없는 상태에서 시작했다. 나는 1학년 2학년 1학기 공부를 하고 군대를 갔다. 군대를 가서 곰곰이 생각해봤다. 내가 과연 이 과가 적성에 맞고 흥미가 있는지? 사실 나는 의류 디자인과를 가고 싶었다. 하지만 성적에 맞추다보니 단국대 소프트웨어학과에 진학했다. 그래서, 현재에 최선을 다하고 싶다는 생각을 했다.

나는 프로그래밍언어 공부를 하면서 과연 내가 이걸 배운다고 컴퓨터를 잘 만질 수 있을까? 라는 의문이 들었다. 그래도 공부는 해야 돼서 공부를 해왔다. 그리고 복학해서 처음 시스템 프로그래밍을 듣게 되었다. 그리고 수업시간에 배운 시스템 프로그램 정의는 이렇다. “응용 프로그램을 돌 수 있는 환경” 그리고 컴퓨터 구성엔 무엇이 있는지에 대해서도 조금은 알게 되었다. 사실 C언어, C++을 공부하면서, 내가 컴퓨터 구성요소도 잘 모르는데 이거를 공부한다고 해서 컴퓨터를 잘 다룰 수 있을까란 의문이 들었었다. 주변에서 이런 질문도 받아봤다. “CPU는 뭐가 좋니? 램과 하드디스크 차이는 뭐니?” 단순히 컴퓨터와 관련 있는 과라 그런지 질문을 수 없이 많이 받았었다. 그래서 요즘엔 컴퓨터 사양도 높아져 그런 질문을 예전보다 더 많이 받고 있다. 하지만 시스템 프로그래밍을 듣기 전에는 뭐가 무엇인지 CPU가 하는 역할이 무엇인지 잘 몰랐다. 하지만 이 과목을 듣고 나서는 CPU가 무엇인지 어떤 역할을 하는지, RAM과 하드디스크에 차이는 무엇인지 정확히 설명을 할 수 있게 되었다. 아직 자세히 시스템 프로그래밍을 배우기 전이지만, 드디어 내가 설명 할 수 있는 것이 생겼다. 사실 그 사람들은 C언어 C++언어 보다는 그런 것에 관해 관심이 더 많기 때문에 유용하다고 생각했다.

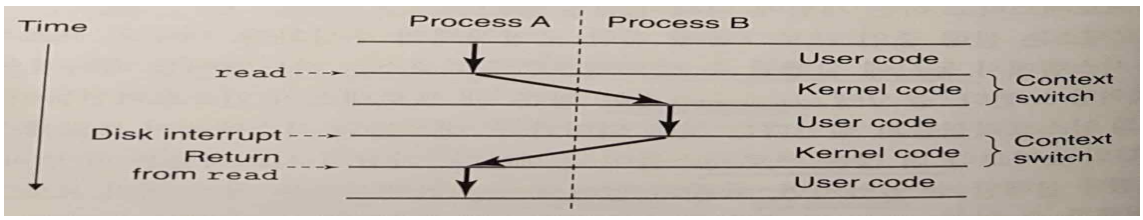
아직 시스템 프로그래밍을 자세히 배우기 전이다. 내가 뭘 배워야 될지, 아직까지 나도 잘 모르겠다. 그러면 목적을 과연 알까? 교수님이 왜 이것을 배워야 되는지 설명은 해주셨지만, 내가 직접 느끼지 못하면 설명을 할 수가 없었다. 그러나 하나 깨달은 것이 있다. C언어와 C++을 공부했을 때 보다는 더 관심이 간다는 것, 내가 시스템프로그래밍 말고도 다른 과목들 또한 열심히 해서 남들에게 설명할 수 있을 정도에 실력을 가지고 싶다고 생각했다. 그럴려면, 내가 지금 수강하고 있는 과목이 뭐를 배우는지에 대해 한 번 더 생각하는 계기가 되었다. 단순히 나에게 이 과제는 점수 채우기 목적이 아닌, 내가 어떻게 이번 학기를 어떻게 다녀야 될지, 어떤 식으로 공부를 해야 될지 생각하는 계기가 되었다. 그래서 나는 시스템 프로그래밍을 스터디 할 정도의 실력을 가지고 싶어 졌고, 그것이 공부하는 목적이 되었다. 비록 어렵지만 후배들에게 어렵다고 소문난 시스템 프로그래밍 언어를 가르쳐 주고 싶다.

1.7.1 프로세스

hello 같은 프로그램이 최신 시스템에서 실행될 때 운영체제는 시스템에서 이 한 개의 프로그램만 실행되는 것 같은 착각에 빠지도록 해준다. 프로그램이 프로세서, 메인 메모리, 입출력장치를 모두 독차지하고 있는 것처럼 보인다. 프로세서는 프로그램 내의 인스트럭션들을 다른 방해 없이 순차적으로 실행하는 것처럼 보인다. 이러한 환상은 전산학에서 가장 중요한 프로세스라고 하는 개념에 의해서 만들어진다.

프로세스는 실행중인 프로그램에 대한 운영체제의 추상화다. 다수의 프로세스들은 동일한 시스템에서 동시에 실행될 수 있으며, 각 프로세스는 하드웨어를 배타적으로 사용 하는 것처럼 느낀다. 동시에 concurrently라는 말은 한 프로세스의 인스트럭션들이 다른 프로세스의 인스트럭션들과 섞인다는 것을 의미한다. 대부분의 시스템에서 프로세스를 실행할 CPU의 숫자보다 더 많은 프로세스들이 존재한다. 이전의 시스템들은 한 번에 한 개의 프로그램만 실행할 수 있었지만, 요즘의 멀티코어 프로세서들은 여러 개의 프로그램을 동시에 실행할 수 있다. 어느 쪽이건 프로세서가 프로세스들을 바꿔주는 방식으로 한 개의 CPU가 다수의 프로세스를 동시에 실행하는 것처럼 보이게 해준다.

운영체제는 문맥 전환 context switching이라는 방법을 사용해서 이러한 교차실행을 수행한다. 운영체제는 프로세스가 실행하는 데 필요한 모든 상태정보의 변화를 추적한다. 이 컨텍스트라고 부르는 상태정보는 PC, 레지스터 파일, 메인 메모리의 현재 값을 포함하고 있다. 어느 한순간에 단일 프로세서 시스템은 한 개의 프로세스의 코드만을 실행할 수 있다. 운영체제는 현재 프로세스에서 다른 새로운 프로세스를 제어를 옮기려고 할 때 현재 프로세스의 컨텍스트를 저장하고 새 프로세스의 컨텍스트를 복원시키는 문맥전환을 실행하여 제어권을 새 프로세스로 넘겨준다. 새 프로세스는 이전에 중단했던 바로 그 위치부터 다시 실행된다.



그림과 같이, 하나의 프로세스에서 다른 프로세스로의 전환은 운영체제 커널에 의해 관리된다. 커널은 운연체제 코드의 일부분으로 메모리에 상주한다. 응용프로그램이 운영체제에 의한 어떤 작업을 요청하면, 컴퓨터는 파일 읽기나 쓰기와 같은 특정 시스템 콜 system call을 실행해서 커널에 제어를 넘겨준다. 그러면 커널은 요청 된 작업을 수행하고 응용프로그램으로 리턴 한다. 커널은 별도의 프로세스가 아니라는 점에 유의해야 한다. 대산, 커널은 모든 프로세스를 관리하기 위해 시스템이 이용하는 코드와 자료구조의 집합이다.

1.7.2 쓰레드(Thread)

각각의 쓰레드는 해당 프로세스의 컨텍스트에서 실행되며 동일한 코드와 전역 데이터를 공유한다. 다중 쓰레딩도 다중 프로세서를 활용할 수 있다면 프로그램의 실행속도를 빠르게 하는 한 가지 방법이다.

1.7.3 가상메모리

가상메모리는 각 프로세스들이 메인 메모리 전체를 독점적으로 사용하고 있는 것 같은 환상을 제공하는 추상화이다.

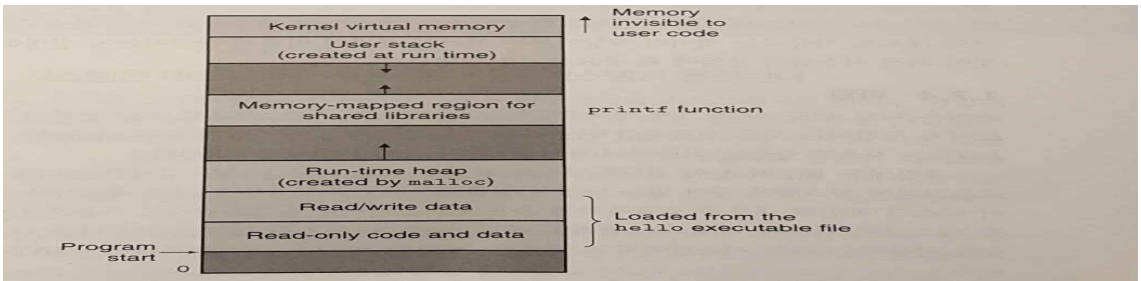


그림 1.13 프로세스 가상주소공간(각 영역은 크기에 맞춰서 그린 것은 아니다)

리눅스에서, 주소공간의 최상위 영역은 모든 프로세스들이 공통으로 사용하는 운영체제의 코드와 데이터를 위한 공간이다. 주소공간의 하위 영역은 사용자 프로세스의 코드와 데이터를 저장한다. 그림에서 위쪽으로 갈수록 주소가 증가하는 점에 주의하라. 낮은 주소부터 위로 올라가면서 간단히 살펴보는 것도 도움이 될 것이다.

- 프로그램 코드와 데이터: 코드는 모든 프로세스들이 같은 고정 주소에서 시작하며, 다음에 C전역변수에 대응되는 데이터 위치들이 따라온다. 코드와 데이터 영역은 실행 가능 목적파일인 hello로부터 직접 초기화 된다.
- 힙heap: 코드와 데이터 영역 다음으로 런타임 힙이 따라온다. 크기가 고정되어 있는 코드, 데이터 영역과 달리, 힙은 프로세스가 실행되면서 C표준함수인 malloc이나 free를 호출하면서 런타임에 동적으로 그 크기가 늘었다 줄었다 한다.
- 공유 라이브러리: 주소공간의 중간 부근에 C 표준 라이브러리나 수학 라이브러리와 같은 공유 라이브러리의 코드와 데이터를 저장하는 영역이 있다.
- 스택stack: 사용자 가상메모리 공간의 맨 위에 컴파일러가 함수 호출을 구현하기 위해 사용하는 사용자 스택이 위치한다. 힙과 마찬가지로 사용자 스택은 프로그램이 실행 되는 동안에 동적으로 늘어났다 줄어들었다 한다. 특히, 함수를 호출할 때마다 스팩이 커지며, 함수에서 리턴될 때는 줄어든다.
- 커널 가상메모리: 주소공간의 맨 윗부분은 커널을 위해 예약되어 있다. 응용프로그램들은 이 영역의 내용을 읽거나 쓰는 것이 금지되어 있으며 마찬가지로 커널 코드 내에 정의된 함수를 직접 호출하는 것도 금지되어 있다. 대신, 이런 작업을 수행하기 위해서는 커널을 호출해야 한다.

가상메모리가 작동하기 위해서는 프로세서가 만들어내는 모든 주소를 하드웨어로 번역하는 등의 하드웨어와 운영체제 소프트웨어 간의 복잡한 상호작용이 필요하다.

1.7.4 파일

파일은 더도 덜도 말고 그저 연속된 바이트들이다. 디스크, 키보드, 디스플레이, 네트워크까지 포함하는 모든 입출력장치는 파일로 모델링한다. 시스템의 모든 입출력은 유닉스 I/O라는 시스템 콜들을 이용하여 파일을 읽고 쓰는 형태로 이루어진다. 파일은 다양한 입출력 장치들의 통일된 간점으로 제공한다.