2장. 소프트웨어의 기초

프로그램이란 특정한 작업을 수행하는 컴퓨터 명령어들의 집합이다. 프로그램은 어셈블리어 와 같이 저급 컴퓨터 언어로 작성할 수도 있고, C 프로그래밍 언어처럼 기계와 무관한 고급 언어로 작성할 수도 있다. 운영체제는 사용자가 스프레드시트나 워드 프로세서와 같은 응용 프로그램을 실행할 수 있도록 해주는 특별한 프로그램이다. 이 장에서는 프로그래밍의 기본 원칙과 운영체제의 목표와 기능에 대한 개요를 제시하고자 한다.

2.1 컴퓨터 언어(Computer Language)

2.1.1 어셈블리어(Assembly Language)

CPU가 메모리에서 가져와 실행하는 명령어는 사람이 전혀 이해할 수 없는 것이다. 이들은 컴퓨터가 정확히 무엇을 해야할 지 말해주는 기계어 코드이다. 인텔 80486 CPU에서 십육진 수 0x89E5는 ESP 레지스터의 내용을 EBP 레지스터로 복사하라는 명령이다. 초창기 컴퓨터 를 위해 개발된 최초의 소프트웨어 도구 중 하나는 어셈블러였다. 어셈블러는 사람이 읽을 수 있는 형태의 소스 파일을 어셈블하여 기계어 코드를 만드는 프로그램이다. 어셈블리어는 레지스터와 자료에 대한 연산을 명시적으로 다루며, 마이크로프로세서마다 다르다. 인텔 x86 마이크로프로세서용 어셈블리어와 알파 AXP 마이크로프로세서용 어셈블리어는 완전히 다르 다. 다음 알파 AXP용 어셈블리 코드는 프로그램이 수행할 수 있는 연산의 예를 보여준다.

```
        ldr r16, (r15)  ; Line 1
        ldr r17, 4(r15) ; Line 2
        beq r16, r17, 100       ; Line 3
        str r17, (r15)  ; Line 4
100:                            ; Line 5
```

첫번째 문장(Line 1)은 레지스터15가 가진 주소에 있는 값을 레지스터16으로 읽어들인다. 그 다음 명령은 메모리 다음 위치의 내용을 레지스터17로 읽어들인다. 세 번째 줄에서는 레지 스터16과 레지스터17의 내용을 비교하여, 이 값이 같으면 레이블100으로 분기한다. 두 레지 스터에 들어있는 값이 같지 않다면, 프로그램은 네 번째 줄로 계속 진행하여 레지스터17의 내용을 메모리에 저장한다. 두 레지스터가 같은 값을 갖고 있다면, 그 값을 저장할 필요가 없다. 어셈블리 수준의 프로그램은 따분하고, 작성하는데 잔꾀가 많이 필요하며, 오류를 범 하기 쉽다. 리눅스 커널 중에서 어셈블리어로 작성된 부분은 극히 일부에 지나지 않는다. 이 들은 단시 효율성을 위해 어셈블리어로 작성되었으며, 특정 마이크로프로세서에 고유하다.

2.1.2 C 프로그래밍 언어와 컴파일러(Compiler)

어셈블리어로 큰 프로그램을 작성하는 것은 어려울 뿐만 아니라 시간도 많이 필요하다. 게 다가 오류를 범하기 쉽고, 특정 프로세서에만 국한되므로 이식성도 없다. 그래서 C같이 기 계에 무관한 언어를 사용하는 것이 훨씬 좋다. C는 프로그램을 처리할 논리적인 자료와 논 리적인 알고리즘으로 표현할 수 있게 해준다. 컴파일러라고 하는 특수한 프로그램은 이 C 프로그램을 읽어서 어셈블리로 변환하여, 특정 기계에 해당하는 코드를 만들어낸다. 좋은 컴 파일러는 훌륭한 어셈블리 프로그래머가 작성한 것에 가깝게 효율적인 어셈블리 코드를 만 들어낸다. 리눅스 커널의 대부분은 C언어로 되어 있다. 다음 C 코드는 앞에 예로 든 어셈블 리 코드와 똑같은 연산을 수행한다.

```
if (x != y)
        x = y;
```

이는 변수 x의 값과 변수 y의 값이 다르면 x에 y의 값을 복사할 것이다. C 코드는 각기 다 른 일을 수행하는 여러개의 루틴들로 이루어진다. 루틴은 어떤 값이나, C언어에서 지원하는 자료형을 리턴할 수 있다. 리눅스 커널같이 큰 프로그램은 많은 수의 C 모듈로 이루어지며, 각 모듈은 자신만의 자료구조와 루틴들로 구성되어 있다. 이런 C 소스 코드 모듈이 모여서 파일 시스템을 다루는 것같은 논리적인 기능을 하게 된다.

C는 여러 가지 변수형을 지원한다. 변수란 심볼 이름으로 참조할 수 있는 메모리 상의 한 위치이다. 프로그래머는 이런 변수가 메모리 상의 어디에 있는지 신경 쓸 필요가 없다. 이 일은 밑에서 설명할 링커가 알아서 해준다. 변수는 각각 정수, 실수, 포인터 등의 다른 종류 의 자료를 가질 수 있다.

포인터는 어떤 자료의 메모리 상의 위치인 주소를 값으로 가지는 변수이다. 어떤 변수 x가 메모리 상의 주소 0x80010000에 있다고 하자. 여기서 x를 가리키는 포인터 변수 - 이것을 px라고 하자 - 를 만들 수 있고, 이 px는 0x80010030 번지에 있다고 하자. 그러면 px의 값은 변수 x의 주소인 0x80010000이게 된다.

C에서는 서로 관련된 변수 여러개를 묶어 하나의 자료구조로 묶을 수 있다. 예를 들어,

```
struct {
        int i;
        char b;
} my_struct;
```

는 i라는 정수(32비트 자료공간을 차지한다)와, b라는 문자(8비트 자료), 이 두 개의 원소를 가진 my_struct라는 자료구조를 정의한다.

2.1.3 링커(Linker)

링커는 여러개의 오브젝트 모듈과 라이브러리를 연결하여 하나의 완결된 프로그램을 만들어 내는 프로그램이다. 오브젝트 모듈은 어셈블러나 컴파일러가 만들어 낸 기계어 코드 출력물 로, 기계어 코드와 자료, 그리고 링커가 다른 모듈과 결합하여 하나의 프로그램을 만들어 내 는데 필요한 정보를 포함한다. 예를 들어 어떤 프로그램에서, 필요한 데이터베이스 함수를 모두 어떤 하나의 모듈이 가지고 있고, 명령행 인자를 처리하는 함수를 다른 모듈이 가지고 있다고 하자. 링커는 하나의 오브젝트 모듈에서 실제로 다른 모듈에 있는 자료구조나 루틴 을 참조하고 있을 때, 이들 모듈 사이의 참조를 맞추어 준다. 리눅스 커널은 많은 요소의 오 브젝트 모듈들을 링크하여 만든, 하나의 거대한 프로그램이다.

2.2 운영체제(Operating System)란 무엇인가?

소프트웨어가 없다면 컴퓨터는 그저 열이나 내는 전자제품 덩어리에 지나지 않는다. 하드웨 어를 컴퓨터의 심장이라고 한다면, 소프트웨어는 컴퓨터의 영혼이라 할 수 있다. 운영체제는 사용자가 응용 프로그램을 실행할 수 있도록 해주는 시스템 프로그램들을 모아놓은 것이다. 운영체제는 실제 하드웨어를 추상화하여 시스템의 사용자와 응용프로그램에게 가상 기계 (virtual machine)를 제공한다. 그래서 실제로 운영체제가 시스템의 특성을 제공해주는 것처럼 느껴진다. 대부분의 PC는 하나 이상의 운영체제를 돌릴 수 있으며, 각 운영체제는 매우 다 른 모습과 느낌을 갖고 있다. 리눅스는 운영체제를 구성하는 여러개의 기능적으로 분리된 조각들로 만들어진다. 리눅스에서 명백하게 구분되는 부분은 커널이지만, 라이브러리나 쉘이 없다면 커널은 무용지물이다.

운영체제가 무엇인지 이해를 할 수 있도록, 다음과 같이 간단한 명령을 쳤을 때 어떤 일이 나는지 생각해보자.

```
$ ls
Mail            c               images          perl
docs            tcl
$
```

여기서 $는 로그인 쉘(이 경우에는 bash)이 내보내는 프롬프트이다. 이는 사용자가 어떤 명 령을 내리기를 기다리고 있다는 것을 의미한다. ls라고 쳐 넣으면 키보드 드라이버는 무슨 글자가 입력되었는지 인식하고 인식한 글자들을 쉘에 넘겨준다. 쉘은 그런 이름을 가진 실 행 이미지가 있는지 찾고, 여기서는 /bin/ls라는 이미지를 찾게 된다. 커널 서비스를 호출 하여 ls라는 실행 이미지를 가상 메모리에 올리고, 이를 실행하게 된다. ls 이미지는 커널의 파일 서브시스템의 함수를 호출하여 어떤 파일들이 있는지 찾는다. 파일 시스템은 캐시된 파일 시스템 정보를 이용하거나, 디스크 디바이스 드라이버를 사용하여 디스크에서 이 정보 를 읽어올 수도 있다. 또는 파일 시스템이 네트웍 파일 시스템(Network File System, NFS)을 통하여 원격으로 마운트된 경우, 액세스해야 하는 원격 파일들의 세부정보를 찾기 위해 네 트웍 드라이버를 이용하여 원격 기계와 정보를 교환할 수도 있다. 어떤 방법으로 정보를 찾 았던 간에, ls는 그 정보를 출력하고, 비디오 드라이버는 이를 화면에 표시한다.

얘기가 좀 복잡해진 것 같지만, 어쨌든 이런 간단한 명령을 통해서도, 운영체제는 사실상 서 로 협동하는 여러 기능들이 모여서 사용자에게 시스템의 일관된 모습을 보여준다는 것을 알 수 있다.

2.2.1 메모리 관리(Memory Management)

자원 - 예를 들어 메모리 - 이 무한히 있다면 운영체제가 하는 일의 상당 부분은 필요없는 일이 될 것이다. 모든 운영체제의 기본기 중의 하나는 적은 양의 실제 메모리(physical memory)를 많이 있는 것처럼 보이게 하는 것이다. 겉으로 보기에 많아 보이는 이 메모리를 가상 메모리(virtual memory)라고 부른다. 이 아이디어는 시스템 내에서 실행중인 소프트웨어 를 속여서 메모리가 많이 있는 것처럼 믿게 만드는 것이다. 시스템은 메모리를 쉽게 다룰 수 있도록 페이지(page)로 쪼개고, 시스템이 실행되면서 이들 페이지를 하드디스크로 스왑 (swap)한다. 소프트웨어는 멀티프로세싱이라는 또 다른 트릭 때문에 이 사실을 깨닫지 못한 다.

2.2.2 프로세스(Process)

프로세스란 실행중인 프로그램이며, 각 프로세스는 각기 하나의 프로그램을 실행하는 구분 된 개체이다. 현재 사용하는 리눅스 시스템에 동작하고 있는 프로세스를 살펴본다면, 상당히 많은 수의 프로세스가 있음을 알 수 있을 것이다. ps라고 타이핑하면 시스템에 있는 프로세 스들을 보여주는데, 예를 들어 다음과 같은 결과가 나온다.

```
$ ps
  PID   TTY STAT        TIME    COMMAND
  158   pRe     1       0:00    -bash
  174   pRe     1       0:00    sh /usr/X11R6/bin/startx
  175   pRe     1       0:00    xinit /usr/X11R6/lib/X11/xinit/xinitrc --
  178   pRe     1 N     0:00    bowman
  182   pRe     1 N     0:01    rxvt -geometry 120x35 -fg white -bg black
  184   pRe     1 <     0:00    xclock -bg grey -geometry -1500-1500 -padding 0
  185   pRe     1 <     0:00    xload -bg grey -geometry -0-0 -label xload
  187   pp6     1       9:26    /bin/bash
  202   pRe     1 N     0:00    rxvt -geometry 120x35 -fg white -bg black
  203   ppc     2       0:00    /bin/bash
 1796   pRe     1 N     0:00    rxvt -geometry 120x35 -fg white -bg black
 1797   v06     1       0:00    /bin/bash
 3056   pp6     3 <     0:02    emacs intro/introduction.tex
 3270   pp6     3       0:00    ps
$
```

만약 시스템에 CPU가 여러개 있다면 각 프로세스는 각기 다른 CPU에서 실행될 수 있을 것 이다 (최소한 이론적으로는 그렇다). 하지만 불행히도 CPU는 보통 하나밖에 없기 때문에 운 영체제는 각각의 프로세스를 돌아가며 짧은 시간 실행하는 또 다른 트릭을 사용해야 한다. 이 짧은 시간을 타임 슬라이스(time-slice)라고 한다. 이런 트릭을 멀티프로세싱(multi- processing) 또는 스케쥴링(scheduling)이라고 부르며, 이는 각 프로세스가 자신만이 유일한 프 로세스인 것처럼 생각하도록 속이는 것이다. 프로세스 간에는 서로 보호가 되기 때문에 한 프로세스가 박살이 나거나 오동작을 해도 다른 프로세스에 영향을 미치지 않는다. 운영체제 는 각 프로세스에게 자신만이 액세스 할 수 있는 분리된 주소공간을 줌으로써 이 기능을 달 성한다.

2.2.3 디바이스 드라이버(Device Driver)

디바이스 드라이버는 리눅스 커널의 주요 부분을 구성한다. 디바이스 드라이버는 운영체제 의 다른 부분들과 마찬가지로 특권층에서 동작하므로, 잘못될 경우 심각한 결과를 가져온다. 디바이스 드라이버는 자신이 제어하는 하드웨어 장치와 운영체제 간의 상호작용을 제어한다. 예를 들어, 파일 시스템은 IDE 디스크에 블럭을 기록할 때 일반적인 블럭 장치 인터페이스 를 사용하는데, 디바이스 드라이버는 장치의 세세한 부분까지 챙기며, 장치마다 다른 일들을 실행한다. 디바이스 드라이버는 구동하려는 컨트롤러 칩에 따라 다르다. 그래서 NCR810 SCSI 컨트롤러가 있다면 NCR810 SCSI 드라이버가 필요한 것이다.

2.2.4 파일 시스템(File System)

유닉스와 마찬가지로, 리눅스에서도 시스템이 사용할 수 있는 구분된 파일 시스템에 접근하 는데 장치 식별자(드라이브 번호나 드라이브 이름같은)를 사용하지 않는다. 대신 파일 시스 템 전체를 하나의

계층적인 트리 구조로 연결하여 하나의 개체로 보여준다. 리눅스는 각각 의 새로운 파일 시스템을 /mnt/cdrom같은 마운트 디렉토리에 마운트하여, 하나의 파일 시 스템 트리 구조에 추가한다. 예를 들면 CD-ROM을 /mn/cdrom으로 마운트하는 것이다. 여 러 가지 파일 시스템을 지원하는 것은 리눅스의 가장 중요한 특징 중의 하나이다. 이는 리 눅스를 매우 유연하게 만들며, 다른 운영체제와 잘 공존할 수 있게 한다. 리눅스에서 가장 많이 사용하는 파일 시스템은 EXT2 파일시스템으로, 대부분의 리눅스 배포판이 EXT2를 지 원한다.

파일 시스템에 의해 사용자는 파일 시스템의 형태나 그 하부의 물리적인 장치의 특징에 상 관없이 시스템의 하드 디스크에 있는 파일이나 디렉토리를 인식할 수 있게 된다. 리눅스는 MS-DOS나 EXT2 등의 많은 다른 파일 시스템을 투명하게 지원하며, 마운트되어 있는 모든 파일과 파일 시스템을 하나의 통합된 가상 파일 시스템(Virtual File System, VFS)으로 제공한 다. 따라서, 사용자와 프로세스는 일반적으로 어떤 파일이 무슨 파일 시스템에 속해 있는지 알 필요 없이 사용하기만 하면 된다.

블럭 디바이스 드라이버는 실제 블럭 장치의 유형(IDE와 SCSI같은)에 따른 차이점을 숨겨주 기 때문에, 파일 시스템에 있어서는 이 물리적 장치는 그저 연속된 데이터 블럭의 모음일 뿐이다. 블럭의 크기는 장치에 따라 다를 수 있다. 예를 들어, 플로피 장치는 공통적으로 512바이트를 사용하는데 반해, IDE 장치는 1024바이트를 사용한다. 이 차이는 시스템 사용자 에겐 보이지 않는다. EXT2 파일 시스템이 어떤 장치에 들어있든 간에 사용자에겐 모두 똑같 이 보인다.

2.3 커널 자료구조(Kernel Data Structure)

운영체제는 시스템의 현재 상태에 대한 매우 많은 양의 정보를 갖고 있어야 한다. 시스템 내부에서 어떤 일이 일어나면 현재 상태를 반영하기 위해 이들 자료구조를 변경해야 한다. 예를 들어, 한 사용자가 시스템에 로그인하면 새로운 프로세스가 만들어지게 되는데, 커널은 이 새로운 프로세스를 나타내는 자료구조를 만들고, 이를 시스템 내의 다른 프로세스를 나 타내는 모든 자료구조와 연결하여야 한다.

이들 자료구조의 대부분은 실제 메모리 상에 존재하는 것이며, 커널과 커널의 서브시스템만 이 액세스할 수 있다. 자료구조는 데이터와 포인터를 포함하며, 이 포인터는 다른 자료구조 나 루틴을 가리킨다. 리눅스 커널이 사용하는 자료구조를 한번에 뭉쳐서 보면 매우 혼동스 러울 수도 있다. 모든 자료구조는 고유의 목적을 갖고 있으며, 일부는 여러 커널 서브시스템 에서 사용하지만, 실제로는 처음 보기보다는 더 단순하다.

리눅스 커널을 이해하는 것은 리눅스 커널의 자료구조와 커널에 있는 여러 함수들이 이를 어떻게 활용하는지 이해하는데 달려 있다. 이 책은 리눅스 커널을 자료구조에 기반하여 설 명한다. 각 커널 서브시스템을 원하는 일을 어떻게 처리하는지를 나타내는 알고리즘과, 커널 의 자료구조를 어떻게 사용하는지를 중심으로 설명한다.

2.3.1 연결 리스트(Linked List)

리눅스는 자료구조를 서로 연결하기 위하여 여러 가지 소프트웨어 공학적 기법을 사용한다. 많은 경우 리눅스는 연결된(linked), 또는 연쇄된(chained) 자료구조를 사용하고 있다. 각 자료 구조가 어떤 것 - 예를 들어 프로세스나 네트웍 장치 - 의 한 존재나 경우를 나타낸다면, 커 널은 이들 모두를 찾아낼

수 있어야 한다. 연결 리스트에서는 루트 포인터가 리스트에 있는 첫 번째 자료구조(또는 원소)의 주소를 가지고, 각 자료구조는 리스트의 다음 원소의 주소를 가진다. 마지막 원소의 다음 원소를 가리키는 포인터는 리스트의 끝임을 나타내기 위해 0 또는 NULL 값을 가진다. 이중 연결 리스트(Doubly Linked List)에서는 각 원소가 다음 원소를 가리키는 포인터와 함께, 이전 원소를 가리키는 포인터도 가진다. 이중 연결 리스트를 사용 하면 메모리 액세스 횟수가 더 많아지긴 하지만, 리스트의 중간에 원소를 추가하거나 삭제 하는 것이 더 쉽다. 이는 운영체제에서 가장 전형적인 트레이드 오프(trade off)[1]이다. 메모리 액세스를 더 할 것인가, 아니면 CPU 사이클을 더 쓸 것인가.

2.3.2 해시 테이블(Hash Table)

연결 리스트는 자료구조를 묶는 손쉬운 방법이지만, 연결 리스트를 탐색하는 것은 비효율적 일 수 있다. 어떤 특정 원소를 찾으려고 할 때, 원하는 걸 발견할 때까지 리스트 전체를 쭉 훑어보아야 하기 때문이다. 이런 제한을 피하기 위해 리눅스는 해싱(hashing)이라는 기법을 사용한다. 해시 테이블은 포인터의 배열, 즉 포인터의 벡터(vector)이다. 배열, 즉 벡터는 어 떤 것들이 메모리 상에 하나씩 이어져 있는 것을 말한다[2]. 즉 책꽂이는 책의 배열이라고 할 수 있다. 배열은 배열에서의 위치를 나타내는 인덱스(index)를 가지고 액세스한다. 책꽂이 비 유를 조금 더 확장한다면, 각각의 책을 '다섯번째 책'과 같은 방식으로 책꽂이에서의 위치로 표현하는 것이다.

해시 테이블은 자료구조에 대한 포인터의 배열이며, 인덱스는 자료구조의 내용으로부터 만 들어진다. 어떤 마을의 인구 분포를 나타내는 자료구조가 있다면, 이를 표현하는데 사람의 나이를 인덱스 값으로 쓸 수 있을 것이다. 이 경우 어떤 사람의 자료를 찾으려고 한다면 그 사람의 나이를 인덱스로 하여 인구 해시 테이블로부터 포인터를 얻고, 그 포인터를 따라가 면 그 사람의 상세자료가 들어있는 자료구조가 나올 것이다. 불행히도 마을에는 같은 나이 를 가진 사람이 많이 있을 수 있다. 그런 경우에는 그 포인터가 같은 나이를 가진 사람들의 연결 리스트를 가리키는 포인터가 된다. 물론 이 짧은 리스트를 찾는 것이 자료구조 전체를 뒤지는 것보다는 여전히 빠를 것이다.

해시 테이블은 자주 사용하는 자료구조로의 액세스 속도를 높여주기 때문에, 리눅스는 캐시 를 구현하기 위해 해시 테이블을 종종 사용한다. 캐시는 빨리 액세스되어야 하는 바로 쓸 수 있는 정보이며, 대개 참조할 수 모든 정보의 일부만을 가지고 있다. 자료구조를 캐시에 넣어두는 것은 커널이 그 자료구조를 자주 액세스하기 때문이다. 캐시는 간단한 연결 리스 트나 해시 테이블에 비하여 사용하고 관리하기가 복잡하다는 단점이 있다. 찾으려는 자료구 조가 캐시에 있다면 (이를 캐시 히트라고 부른다) 아주 좋은 일이다. 그러나 만약 캐시에 없 으면 관련된 자료구조를 모두 뒤져야 하고, 원하는 자료구조가 실제로 있으면 그것을 캐시 에 추가하여야 한다. 새로운 자료구조를 캐시에 넣으려면 옛날 것은 버려야 할 수도 있다. 리눅스는 어떤 것을 버려야 할 지 정해야 하는데, 이번에 버린 자료가 바로 다음에 필요한 것이 되는 위험도 있다.

2.3.3 추상 인터페이스(Abstract Interface)

리눅스는 종종 자신의 인터페이스를 추상화한다. 인터페이스란 특정 방법으로 동작하는 루 틴과 자료구조의 모음이다. 예를 들어, 모든 네트웍 디바이스 드라이버는 특정한 자료구조를 이용하여 정해진 루틴들을 제공해야 한다. 이런 방법으로 장치마다 다른 코드로 된 아래 계 층에서 제공하는 서비스(또는 인터페이스)를 사용하는 일반적인 코드 계층이 있게 된다. 네 트웍 계층은 일반화 되어있고, 장치마다 고유한 코드는 표준 인터페이스를 제공하여 이를 지원한다.

종종 이들 하위 계층은 부팅할 때 상위 계층에 자신을 등록한다. 이러한 등록 과정은 대개 어떤 연결 리스트에 자료구조를 추가하는 일을 수반한다. 예를 들어, 커널에 들어 있는 각각 의 파일 시스템은 부팅할 때 자신을 커널에 등록하며, 모듈을 사용하는 경우에는 처음으로 그 파일 시스템이 사용될 때 등록된다. 어떤 파일 시스템이 등록되어 있는 지를 보려면 /proc/filesystems를 들여다보면 된다. 때로 등록된 자료구조가 함수에 대한 포인터를 가지고 있는 경우도 있다. 이들 포인터는 특정한 업무를 수행하는 소프트웨어 함수의 주소 이다. 다시 파일 시스템 등록을 예로 들어보면, 각 파일 시스템이 등록할 때 리눅스 커널에 넘겨주는 자료구조에는, 파일 시스템이 마운트될 때마다 불리는 파일 시스템에 고유한 루틴 의 주소가 들어있다.

번역 : 고양우, 신문석
정리 : 이호

역주 1) 메모리를 더 액세스하고 CPU 사이클을 적게 쓸 것인가, 또는 메모리 액세스를 적 게하고 CPU 사이클을 더 쓸 것인가 하는 갈림길에서 둘 사이의 타협점을 찾는 것을 말 한다. (flyduck)

역주 2) 이후에 벡터라는 용어는 배열과 같은 의미로 쓰인다. (flyduck)