운영체제 2장 요약 및 배우고 싶은 목표

32183698 이현기

실행중인 프로그램은 프로그램의 구조대로 실행된다. 프로세서는 메모리로부터 명령을 가져오고**(fetch),** 각 명령이 어떤 것인지 알기 위해 **해독하고(decode),** 두 숫자를 더하거나, 메모리에 접근하고, 상태를 확인하고, 기능으로 이동하는 등 프로그램이 해야할 것들을 수행한다**(execute)**. 이 명령이 끝나면, 프로세서는 프로그램이 완전히 끝날때까지 다음명령으로 계속 이동한다. 이 교재를 통해서 프로그램이 실행중일 때, 많은 다듬어지지 않은 것들이 시스템을 사용하기 쉽게 하기 위한 주요 목적을 가지고 일어나고 있다는 것을 배운다. 소프트웨어의 body부분이 사실은 프로그램을 실행하고, 프로그램들이 메모리를 공유하게 하고, 다른 장치들과 통신하게 하고, 그 외 다른 것들을 쉽게 하는 것에 책임진다. 소프트웨어의 body부분을 운영체제라 부르며, 시스템이 사용하기 쉬운 방식으로 정확하고 효율적이게 운영되는 것을 보장하는 것을 담당한다. 운영체제가 이것을 하기 위한 주요 방법은 **가상화(virtualization)**라 불리는 일반적인 기술을 통하는 것이다. 운영체제는 프로세서나 메모리, 디스크와 같은 물리적인 자원을 이용하여 더 일반적이고, 힘쎄고, 사용하기 쉬운 그것의 가상형태로 변환한다. 그래서 가끔 운영체제를 가상기계라고도 얘기한다. 또한, 사용자들이 운영체제에게 무엇을 해야할지와 프로그램을 실행하거나, 메모리 할당, 파일접근과 같은 가상기계의 특징들을 이용하기 위해, 운영체제는 또한 내가 이용할 수 있는 API를 제공한다. 사실 전형적인 운영체제는 애플리케이션에서 이용할 수 있는 수백개의 시스템 콜들을 이용한다. 우리는 가끔 운영체제가 애플리케이션에 대한 표준 라이브러리를 제공한다고 한다. 마지막으로, 가상화가 많은 프로그램들을 실행할 수 있게 하고, 동시에 프로그램들의 구조와 데이터에 접근하게 하고, 프로그램들이 장치에 접근할 수 있게 하기 때문에, 운영체제는 가끔 **자원 관리자**로도 알려져있다. 각각 CPU, 메모리, 그리고 디스크는 시스템에서 자원이다: 그래서 이러한 자원들을 효율적으로 사용하거나 공정하게 또는 확실한 많은 다른 가능한 목표들을 염두하여 관리하는 것이 운영체제의 역할이다.

2.1 CPU의 가상화

Figure 2.1에서는 Spin()을 부르는데, 이 함수는 반복적으로 시간을 확인하고 1초동안 실행했으면 return한다. 그런 후에 유저가 command line에 입력한 문자열을 출력하고 이것을 반복한다. 이 파일을 컴파일하고 1개 프로세서인 시스템에서 실행하면 시간이 경과할때까지 반복적으로 시간을 확인한다. 이 프로그램은 계속 실행되지만 Ctrl+c를 통해 프로그램을 중단할 수 있다. 이 프로그램을 다른 방식으로 실행해본다. figure 2.2는 2.1보다 더 복잡한 결과를 보여준다. 여기서 한 가지 흥미로운 것을 알 수 있는데 한 프로세서를 가지고 있음에도 불구하고, 이 4개의 프로그램들이 같은 시간에 실행되고 있는 것을 볼 수 있다. 운영체제가 하드웨어의 도움을 받아 시스템이 매우 많은 수의 가상 CPU들을 가지고 있다는 환상을 만든다. 1개 CPU를 많은 프로그램들을 동시에 실행하게 하는 무한의 CPU로 보이게 하는 것을 **CPU를 가상화(virtualizing the CPU)**한다고 한다. 또한, 프로그램을 실행하고 멈추고 운영체제에게 어떤 프로그램들이 실행될지 알려주기 위해서, 나의 요구를 운영체제에게 의사소통하는데 사용할 수 있는 인터페이스(API)가 필요하다. 그리고 여러 프로그램들을 실행할 능력이 모든 분야의 새로운 질문들을 야기한다는 것을 알았을 것이다.

예를들어, 2개의 프로그램이 특정 시간에 실행되기를 원한다면 어떤 프로그램을 실행할지? 와 같은 것이다. 이러한 질문은 운영체제의 정책(policy)에 의해 결정된다. 나중에 운영체제가 구현하는 기본적인 메커니즘들에 대해 배운다. 그러므로 운영체제의 역할은 자원 관리자이다.

2.2 메모리 가상화

이제는 메모리를 고려해본다. 현대의 기계들에 있는 물리적인 메모리 모델은 매우 단순한데 bytes의 배열로 이루어져 있다. 메모리를 읽고, 데이터가 저장된 곳에 접근하기 위한 주소와, 메모리에 작성하고, 주어진 주소에 데이터를 명확하게 하기 위함이다. 메모리는 프로그램이 실행중일 때 항상 접근된다. 프로그램은 모든 데이터 구조들을 메모리에 유지하고, load하고 저장하거나 작업을 수행할 때 메모리에 접근하는 명백한 명령들과 같은 다양한 명령들을 통해 메모리에 접근한다. 프로그램의 명령들 또한 메모리에 있으므로 각각의 명령어 패치에도 메모리에 접근한다. malloc()을 이용함으로써 몇몇의 메모리에 할당하는 프로그램을 보면 몇 가지 것들을 한다. 먼저, 메모리를 할당하고, 메모리의 주소를 출력한다. 그 후에 새로 할당받은 메모리의 첫 번째 자리에 숫자 0을 넣는다. 그러고나서, 루프를 돌고, 1초의 딜레이가 발생하고 p에 있는 주소에 저장된 값이 증가한다. 모든 출력 상황에, 실행한 프로그램에 PID(process identifier)라 불리는 것을 출력한다. 이 PID는 실행중인 프로세스별로 있는 고유한 번호이다. 첫 번째 할당받은 주소를 보여주는 것은 그리 흥미롭지 않지만, 여러 프로세스를 실행했을 때, 각각의 실행중인 프로그램이 같은 주소의 메모리를 할당받았고, 각각의 주소에 있는 값이 독립적으로 갱신되고 있다. 이것은 각각의 실행중인 프로그램이 다른 실행중인 프로그램들과 같은 물리메모리를 공유하는 것 대신에 각각의 실행중인 프로그램들이 인 메모리를 가지고 있는 것처럼 보인다. 실제로 이것은 정확히 운영체제가 메모리를 가상화할 때 일어나는 일이다. 각각 프로세스는 운영체제가 어떻게든 기계의 물리메모리에 map한 개인적인 가상 주소공간에 접근한다. 그러나, 현실은 물리메모리는 운영체제에 의해 관리되는 공유된 자원이다.

이 책의 또 다른 주요 테마는 **병합성(concurrency)**이다. 우리는 같은 프로그램에서 많은 것들을 작업할 때 발생하고 반드시 기록되어야하는 여러 문제들에 대해 지칭하는 개념의 용어로 사용한다. 병합성의 문제들은 운영체제 자체에 먼저 발생했다. 가상화를 예로 볼 수 있는데, 운영체제가 먼저 한 프로세스를 실행하고, 다른 것을 하고, 기타 등등 많은 것들을 한번에 한다. 불행하게도, 병합성의 문제들은 이제 운영체제 자체에서만 제한되는 것이 아니라 현대의 멀티쓰레드(multi-threaded) 프로그램들에서도 같은 문제가 제기된다. 비록 이 예시를 완전히 이해하지 못해도 나중에 배울 것이고, 기본적인 아이디어는 간단하다. 메인 프로그램은 Pthread_create()를 이용하여 2개의 쓰레드를 생성한다. 쓰레드는 다른 함수와 같은 메모리 공간에서 실행중인 함수로, 한번에 한 개 이상의 함수들이 있다고 생각하면 된다. 이 예에서, 각각의 쓰레드는 루프의 수만큼 카운트하는 worker()라 불리는 routine으로 실행되기 시작한다. 루프의 값은 루프 안에서 공유된 카운트 수를 증가할 각각 2개의 worker들이 얼마나 많은 수를 증가시킬지 결정한다. 루프의 값을 1000으로 하고 프로그램을 실행하였을 때, 마지막 카운트 값은 생각한대로 2000이 된다. 즉, 루프의 값이 N으로 설정하면, 마지막 출력값은 2N이 된다. 그러나 루프의 값을 매우 큰 값을 줬을 때는 이대로 되지 않을 수 있다. 예에서 본 것처럼 루프값을 100000을 줬을 때 원래대로라면 200000이 마지막 출력값이 되야 하는데 20000보다 작은 143012가 출력되었다. 이 프로그램을 2번 돌려도 계속 틀린 값을 얻을 뿐만 아니라, 마지막 값과 다른 값을 출력한다. 왜 이런일이

발생하는 걸까?
밝혀진 바와 같이, 이상하고 특이한 결과들은 한번에 하나씩인 어떻게 명령들을 수행하는지와 관련되있다. 불행하게도, 앞서 공유된 카운터가 실행되는 곳인 프로그램의 중요 부분은 세가지 명령들을 따른다: 메모리에서 레지스터로 카운터 값을 load하는 것과, 그것을 실행하는 것과, 다시 메모리에 저장하는 것이다. 이 3개의 명령들이 즉시 실행되지 않기 때문에, 이상한 것들이 발생할 수 있다. 이것은 나중에 자세하게 다룰 병합성의 문제이다.

2.4 영속성(Persistence)
 이 코스의 세 번째 중요 테마는 영속성이다. 시스템 메모리에서, 데이터는 DRAM이 휘발성 방식으로 값들을 저장하는 장치들과 같이 쉽게 없어질 수 있다. 따라서, 하드웨어와 소프트웨어가 데이터를 지속적으로 저장하는 것이 필요하다. 이러한 저장공간은 유저들이 그들의 데이터에 대해 많은 관심을 가질 수 있는 시스템이기 때문에 중요하다.
하드웨어는 입출력 장치와 같은 종류의 형식으로, 하드드라이브는 장기간 정보를 보관하는 흔한 저장장치이고, SSD는 이 분야에서 크게 발전하고 있다.
주로 디스크를 관리하는 운영체제 안의 소프트웨어는 파일 시스템이라 불린다. 이것은 유저가 시스템에서 디스크들의 믿을 수 있고 효율적인 형식으로 생성한 파일들을 저장하는 것에 책임이 있다. CPU와 메모리에 대한 운영체제에서 제공하는 추상화와는 달리, 운영체제가 각각 애플리케이션에 개인적이고 가상화된 디스크를 생성하지 않는다. 오히려, 유저들은 파일 안에 있는 정보를 공유하길 원할 것이다. 이것을 쉽게 이해하기 위해, 몇몇 코드를 볼 것인데 figure 2.6에서는 hello world라는 문자열이 포함된 파일을 /tmp/file에 생성하는 코드이다.
이 일을 하기 위해선 프로그램은 운영체제에게 3가지 콜을 해야한다. 첫 번째로 open()콜인데, 파일을 열고 생성한다. 두 번째로 write()콜인데, 파일에 데이터를 작성한다. 세 번째로 close()콜인데, 간단히 파일을 닫는 것으로 다른 데이터들이 작성되지 않게 한다. 이런 시스템 콜들은 파일시스템이라 불리는 운영체제의 한 부분이며, 요청을 관리하고 유저에게 에러 코드를 돌려준다.
파일 시스템은 일의 bit단위로 수행하는데, 먼저 디스크에 새로운 데이터가 저장될 곳을 파악하고, 파일 시스템이 유지하는 다양한 구조들을 찾는다. 운영체제가 시스템 콜을 통해 장치들에 접근하는 표준적이고 간단한 방법을 제공한다. 운영체제는 가끔 표준 라이브러리로 보기도 한다. 물론 어떻게 장치들이 접근하는지, 파일 시스템들이 지속적으로 데이터를 관리하는지에 대한 많은 세부사항들이 있다. 이해하기 어렵지만, 나중에 3번째 부분에서 영속성에 대해 다시 배운다.

2.5 Design Goals
 이제 운영체제가 실제로 어떤 일을 하는지에 대해 알았을 것이다. CPU, 메모리 또는 디스크와 같은 물리적인 자원들을 가상화하고, 병합성과 관련된 거칠고 까다로운 문제들을 다루고, 긴 시간동안 파일들을 지키기 위해 지속적으로 파일들을 저장한다. 이러한 시스템을 구축하기 위해, 필수적으로 우리 디자인과 실행에 집중하고 trade-off를 만드는 것을 도와주기 위해 몇몇 목표들을 가지고 있어야 한다. 적절한 trade-off을 찾는 것이 시스템을 구성하는데 키가 된다.
가장 기본적인 목표들 중 하나는 시스템을 편리하고 사용하기 쉽게 만들기 위해 몇몇 추상화를 구성하는 것이다. 추상화는 우리가 컴퓨터 과학에서 하는 모든 것에 기본이다.

추상화는 큰 프로그램을 작고 이해하기쉬운 조각들로 나눌 수 있게 하고, 어셈블리언어를 생각하지 않고 C와 같은 고급언어로 프로그램을 작성할 수 있게 하고, 로직 게이트에 대해 생각하지 않고 어셈블리 코드로 작성할 수 있고, 트랜지스터들에 대해 많은 생각하지 않고 게이트 밖에 있는 프로세서를 만들 수 있게 한다. 각 부분마다, 운영체제의 조각들에 대해 생각할 방법을 주는 시간이 흐름에 따라 발전된 주요 추상화들에 대해 얘기한다.
운영체제를 설계하고 실행하는 목표는 높은 수행력을 제공하는 것이다. 다르게 얘기하면 우리 목표는 운영체제의 오버헤드를 최소화하는 것이다. 가상화와 시스템을 쉽게 사용하게 만드는 것은 가치 있으나, 비용 없이는 안된다. 따라서, 과도한 오버헤드 없이 가상화와 다른 운영체제 특징들을 제공하기 위해 노력해야한다. 오버헤드들은 몇가지 형식들로 발생한다. 추가 시간(많은 명령들), 추가공간(메모리 또는 디스크). 우리는 이것들을 가능한 최소화하는 해결책들을 찾을 것이다.
또다른 목표는 애플리케이션 사이뿐만 아니라 운영체제와 애플리케이션 사이에서 보호를 제공하는 것이다. 많은 프로그램들을 같은 시간에 수행하는 것을 바라기 때문에, 악의적이고 돌발적인 나쁜 행동은 다른 것들을 해할 수 없다. 보호는 운영체제의 근본적인 주요 원칙들 중 하나인 분리의 핵심이다. 프로세스들을 또 다른 것으로부터 분리하는 것은 보호의 키이며 운영체제가 해야할 것이다.
운영체제는 또한 멈춤 없이 계속 실행해야 한다. 이것에 실패하면, 시스템에서의 모든 애플리케이션 실행은 실패한다. 이런 의존성 때문에, 운영체제는 높은 수준의 신뢰성을 제공하기 위해 노력한다.
다른 목표들도 마찬가지다. 에너지 효율성은 증가하는 친환경 세상에서 중요하다. 악의적인 애플리케이션에 대응하는 보안은 특히 네트워크 시대에서 중요하다. 운영체제가 작아진 장치들에서 실행됨에 따라 mobility도 점차 중요해지고 있다. 어떻게 시스템이 사용되는지에 따라, 운영체제는 다른 목표들을 가지고 있고 최소한 다른 방법들로 실행할 가능성이 있다. 그러나, 여기서 알 수 있듯이, 많은 원칙들이 운영체제를 다른 장치들의 범위 내에서 유용하게 구성할 수 있도록 한다.

2.6 역사

인간이 만든 시스템과 같이, 엔지니어들이 그들의 디자인에 중요한 것이 무엇인지 배웠기 때문에 좋은 생각들이 시간이 지남에 따라 운영체제에 축적되었다. 여기서 몇가지 주요 발전들에 대해 얘기한다.

Early Operating Systems: Just Libraries

초창기에, 운영체제는 많은 일을 하지 않았다. 기본적으로, 운영체제는 단지 자주 사용하는 함수들의 라이브러리들의 집합이었다. 주로, 오래된 mainframe 시스템들은 한 프로그램이 실행되고 인간 조종사에 의해 통제되었다. 운영체제가 하는 것은 이 조종사에 의해 수행되었다. 만약 똑똑한 개발자였다면, 이 조종사에게 친절히 대하며, 그들이 내 직업을 큐 앞으로 이동시킬 것이다. 이런 컴퓨팅 모드는 조종사에 의해 일들의 수가 정해지고 실행되기 때문에 **batch processing**이라고 알려져있다.

Beyond Libraries: Protection

주로 사용하는 서비스들의 간단한 라이브러리가 되는 것을 넘어, 운영체제 시스템은 기계들을 관리하는데 더 중요한 역할을 맡았다. 운영체제를 대신하여 코드가 실행되는 것이 특별하다고

깨달은 것이 중요한 부분이다. 운영체제는 장치들을 통제하기 위해 일반적인 애플리케이션 코드와 다르게 다뤄진다. 따라서, 파일시스템을 라이브러리로 실행하는 것은 거의 의미가 없다. 그래서 Atlas 컴퓨팅 시스템에 의해 개척된 시스템 콜이 개발되었다. 시스템 콜과 프로시저 콜 사이의 다른 점은 시스템 콜은 하드웨어 권한 수준을 올림과 동시에 운영체제에 통제를 전달한다. 유저 애플리케이션들은 하드웨어가 애플리케이션이 할 수 있는 것을 제한하는 것을 의미하는 유저 모드에 적용된다. 시스템 콜이 초기화될 때(주로 trap이라 불리는 특별한 하드웨어 명령을 통해), 하드웨어는 미리 지정된 trap handler에 신호를 전달함과 동시에 특권 수준을 커널모드로 올린다. 커널모드에서, 운영체제는 시스템의 하드웨어에 대한 접근을 하여 I/O 요청을 초기화하거나 프로그램이 이용가능한 메모리를 더 만드는 것과 같은 것을 할 수 있다.

The Era of Multiprogramming

운영체제가 실제로 도약한 것은 메인프레임을 넘어 미니컴퓨터의 시대이다. 멀티프로그래밍은 기계 자원들을 더 잘 사용하는 욕망에 의해 보편화되었다. 시간에 단지 한 일을 실행하는 것 대신에, 운영체제는 메모리에 여러개의 일들을 load하고 그것들 사이로 빠르게 바꿔가면서 CPU 활용률을 향상시켰다. 메모리 보호와 같은 문제들이 중요해졌다. 멀티프로그래밍에 의해 소개된 병합성 문제를 다루는 것에 대해 이해하는 것 또한 중요하다. 그 시대의 주요 실용적은 진보중 하나는 Bell Labs에서 Ken Thompson이 개발한 UNIX 운영체제이다. UNIX는 다른 운영 시스템들로부터 좋은 아이디어들을 가져왔고, 이것들을 간편하고 사용하기 쉽게 만들었다.

The Modern Era

미니컴퓨터를 넘어서 새로운 형태의 싸고 빠른 기계들이 나왔다. 오늘날 PC(Personal Computer)라 부른다. 불행하게도, 운영체제는 초기 시스템이 미니컴퓨터의 시대에서 배운 것들을 잊어 초창기의 PC는 하락세를 보였다. 예를 들어, DOS(Disk Operating System)과 같은 초장기 운영 시스템은 메모리 보호를 중요하게 생각하지 않아 악성 애플리케이션이 모든 메모리에 들어갈 수 있다. 다행히도, 몇 년의 침체기 후에, 미니컴퓨터 운영 시스템들의 오래된 특징들은 데스크톱으로 제 자리를 찾아갔다.

이 교과목에서 배우고 싶은 목표

보통 운영체제라고 하면 윈도우, Mac OS, 리눅스 등을 먼저 떠올리게 됩니다. 이러한 운영체제가 정확히 어떤 것이고, 컴퓨터 안에서 하는 역할이 무엇인지, 내부는 어떻게 구성이 되어 있는지, 그리고 하드웨어와 소프트웨어에 어떤 관련이 있는지 등을 배우고 싶습니다. 그리고 전 학기에 수강한 시스템프로그래밍에서 배운 시스템콜이나 가상화 등을 운영체제에서 어떻게 활용하는지에 대해 학습하고 운영체제에 대한 이해를 하는 것이 목표입니다.

OSTEP 2장에서 나온 예제들을 실행해본 스크린샷입니다.

```
lee99@DESKTOP-GR3R6AC:/mnt/c/users/lee99$ cd os
lee99@DESKTOP-GR3R6AC:/mnt/c/users/lee99/os$ vi common.h
lee99@DESKTOP-GR3R6AC:/mnt/c/users/lee99/os$ vi cpu.c
lee99@DESKTOP-GR3R6AC:/mnt/c/users/lee99/os$ vi cpu.c
lee99@DESKTOP-GR3R6AC:/mnt/c/users/lee99/os$ gcc -o cpu cpu.c -Wall
lee99@DESKTOP-GR3R6AC:/mnt/c/users/lee99/os$ ./cpu "A"
A
A
A
A
A
^C
lee99@DESKTOP-GR3R6AC:/mnt/c/users/lee99/os$
```

```
lee99@DESKTOP-GR3R6AC:/mnt/c/users/lee99/os$ gcc -o 2_3 2_3.c
lee99@DESKTOP-GR3R6AC:/mnt/c/users/lee99/os$ ./2_3
(171) address pointed to by p: 0x556ed2a182a0
(171) p: 1
(171) p: 2
(171) p: 3
(171) p: 4
(171) p: 5
(171) p: 6
(171) p: 7
^C
lee99@DESKTOP-GR3R6AC:/mnt/c/users/lee99/os$
```

```
lee99@DESKTOP-GR3R6AC:/mnt/c/users/lee99/os$ gcc -o 2_4 2_4.c -Wall -pthread
lee99@DESKTOP-GR3R6AC:/mnt/c/users/lee99/os$ ./2_4 1000
Initial value : 0
Final value : 2000
lee99@DESKTOP-GR3R6AC:/mnt/c/users/lee99/os$ ./2_4 100000
Initial value : 0
Final value : 125765
lee99@DESKTOP-GR3R6AC:/mnt/c/users/lee99/os$ ./2_4 100000
Initial value : 0
Final value : 114258
lee99@DESKTOP-GR3R6AC:/mnt/c/users/lee99/os$
```

```
lee99@DESKTOP-GR3R6AC:/mnt/c/users/lee99/os$ vi 2_5.c
lee99@DESKTOP-GR3R6AC:/mnt/c/users/lee99/os$ cat 2_5.c
#include <stdio.h>
#include <unistd.h>
#include <assert.h>
#include <fcntl.h>
#include <sys/types.h>

int main(int argc, char *argv[])
{
        int fd = open("/tmp/file", O_WRONLY | O_CREAT | O_TRUNC, S_IRWXU);
        assert(fd > -1);
        int rc = write(fd, "hello world₩n", 13);
        assert(rc == 13);
        close(fd);
        return 0;
}
```