5

운영체제 개요

만일 당신이 학부 운영체제 수업을 듣고 있다면 컴퓨터 프로그램이 실행될 때 어떤 일을 하는지 대략은 알고 있을 것이다. 그렇지 않다면 이 책을 읽는 것과 관련 수업을 듣는 것은 힘든 일이 될 것이다. 일단 읽기를 멈추고 가까운 서점에 가서 필요한 배경 지식을 제공하는 책을 구입하여 읽은 후 다시 시작하기 바란다. Patt/Patel [PP03]과 특히 Bryant/O'Hallaron [BO10]은 매우 훌륭한 책이다.

프로그램이 실행될 때 어떤 일이 일어날까?

프로그램은 매우 단순한 일을 한다: 명령어를 실행한다. 프로세서는 명령어를 초당 수백만 번 (요즘은 수십억 번) **반입**(**fetch**)하고, **해석**(**decode**)하고 (즉, 무슨 명령어인지 파악하고), **실행**(**execute**)한다(즉, 두 수를 더하고, 메모리에 접근하고, 조건을 검사하고, 함수로 분기하는 등의 정해진 일을 한다). 명령어 작업을 완료한 후 프로세서는 다음 명령어로, 또 그 다음 명령어로 프로그램이 완전히 종료될 때까지 실행을 계속한다[1].

우리는 방금 **Von Neumann** 컴퓨팅 모델의 기초를 설명하였다[2]. 간단하게 들리지, 그렇지? 우리는 이 수업에서 시스템을 사용하기 **쉽게하기 위해**(**easy to use**) 프로그램 실행 시 다양한 일들이 발생한다는 것을 배우게 될 것이다.

프로그램을 쉽게 실행하고 (심지어 동시에 여러 개의 프로그램을 실행시킬 수도 있음), 프로그램 간의 메모리 공유를 가능케 하고, 장치와 상호작용을 가능케하고, 다양 흥미로운 일을 할 수 있게 하는 소프트웨어가 있다. 시스템을 사용하기 편리하면서 정확하고 올바르게 동작시킬 책임이 있기 때문에 소프트웨어를 **운영체제**(**operating system**, **OS**)라고 부른다[3].

1) 물론 현대 프로세서는 프로그램을 빨리 실행시키기 위해서 안 보이는 곳에서 이상하고 기괴한 일을 한다. 예를 들면, 동시에 여러 명령어를 실행시키고 심지어는 명령어의 순서를 바꾸어서 실행시키기도 한다! 이런 일들은 우리의 관심사가 아니다. 우리는 대부분의 프로그램이 가정하고 있는 한 번에 하나의 명령어만 실행되고 순차적으로 실행되는 단순한 모델에만 관심을 둔다.

2) Von Neumann은 컴퓨팅 시스템의 초기 개척자 중의 한 명이었다. 그는 게임 이론과 원자 폭탄 분야에서도 선구자적 역할을 하였으며 NBA에서 6년 동안 선수 생활도 하였다. 이 중 하나는 사실이 아니다. 뭘까?

3) OS를 부르는 다른 이름에는 슈퍼바이저 (supervisor) 또는 심지어 주 제어 프로그램 (master control program)도 있다. 분명히 후자는 약간 열정이 지나친 것처럼 들리기 때문에 (자세한 내용은 영화 Tron을 보시오) 따라서 대신 감사하게도 "운영체제"라는 용어가 유행하게 되었다.

운영체제는 앞에서 언급한 일을 하기 위하여 **가상화**(**virtualization**)라고 불리는 기법을 사용한다. 운영체제는 프로세서, 메모리, 또는 디스크와 같은 **물리적** (**physical**)인 자원을 이용하여 일반적이고, 강력하고, 사용이 편리한 가상 (virtual) 형태의 자원을 생성한다. 때문에 운영체제를 때로는 **가상 머신**(**virtual machine**)이라고 부른다.

사용자 프로그램의 프로그램 실행, 메모리 할당, 파일 접근과 같은 가상 머신과 관련된 기능들을 운영체제에게 요청할 수 있도록, 운영체제는 사용자에게 API를 제공한다. 보통 운영체제는 응용 프로그램이 사용 가능한 수백 개의 **시스템 콜**을 제공한다. 운영체제가 프로그램 실행, 메모리와 장치에 접근, 기타 이와 관련된 여러 작업을 진행하기 위해 이러한 시스템 콜을 제공하기 때문에, 우리는 운영체제가 표준 라이브러리 (standard library)를 제공한다고 일컫기도 한다.

마지막으로, 가상화는 많은 프로그램들이 CPU를 공유하여, 동시에 실행될 수 있게 한다. 프로그램들이 각자 명령어와 데이터를 접근할 수 있게 한다. 프로그램들이 디스크 등의 장치를 공유할 수 있게 한다. 이러한 이유로 운영체제는 **자원 관리자**(**resource manager**)라고도 불린다. CPU, 메모리, 및 디스크는 시스템의 **자원**이다. 효율적으로, 공정하게, 이들 자원을 **관리**하는 것이 운영체제의 역할이다. 운영체제의 역할을 좀 더 잘 이해하기 위해서 몇 가지 예를 살펴 보기로 하자.

5.1 CPU 가상화

핵심 질문: 자원을 어떻게 가상화시키는가

이 책에서 우리가 대답할 가장 중요한 질문은 간단하다: 운영체제는 어떻게 자원을 가상화하는가? 이게 바로 우리의 핵심 문제이다. 답변이 너무 뻔하기 때문에 운영체제가 가상화시키는 *이유*는 궁금하지 않다. 시스템을 사용하기 편리하게 만들기 때문이다. 우리는 그 *방법에* 초점을 맞춘다. 가상화 효과를 얻기 위하여 운영체제가 구현하는 기법과 정책은 무엇인가? 운영체제는 이들을 어떻게 효율적으로 구현하는가? 어떤 하드웨어 지원이 필요한가?

운영체제를 구현할 때 해결해야 하는 특정 문제를 강조하는 방법으로 지금과 같이 음영 상자 형식으로 "*핵심 질문*"을 나타낼 것이다. 특정 주제를 다루면서 하나 이상의 핵심 질문을 만날 것이다. 그리고 그 장에서는 해당 질문에 답할 수 있는 해결책 또는 해결책의 기본 특징을 제시한다.

그림 5.1은 우리의 첫 번째 프로그램이다. 이 프로그램은 많은 일을 하지 않는다. 하는 일은 `Spin()`을 호출하는 것이다. `Spin()`은 1초 동안 실행된 후 리턴하는 함수이다. 그런 후 사용자가 명령어 라인으로 전달한 문자열을 출력한다. 이러한 일련의 작업을 무한히 반복한다.

이 코드를 `cpu.c`라는 이름으로 저장하고 단일 프로세서(또는 **CPU** 라는 용어도 섞어서 사용할 것임) 시스템에서 컴파일하고 실행시킨다고 가정하자. 다음과 같은 출력을 볼 수 있을 것이다.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <assert.h>
#include "common.h"
int
main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "usage: cpu <string>\n");
        exit(1);
    }
    char *str = argv[1];
    while (1) {
        Spin(1);
        printf("%s\n", str);
    }
    return 0;
}
```

〈그림 5.1〉 **간단한 예 : 반복해서 출력하는 코드 (cpu.c)**

```
prompt> gcc -o cpu cpu.c -Wall
prompt> ./cpu "A"
A
A
A
A
^C
prompt>
```

그렇게 재미있는 결과는 아니다—시스템은 프로그램 실행 후 1초가 지나면 사용자가 전달한 입력 문자열을 (이 경우, 문자 "A") 출력한다. 출력 후 실행을 계속한다. 프로그램은 계속 실행된다는 것에 주목하자. "Control-c"를 (UNIX-계열 시스템에서 전위 프로그램을 종료시키는 키) 눌러 프로그램을 종료시킬 수 있다.

이번에는 같은 작업에 대한 여러 인스턴스를 동시에 실행시켜 보자. 그림 5.2에 약간 복잡해 보이는 실행 결과가 나와 있다.

```
prompt> ./cpu A & ; ./cpu B & ; ./cpu C & ; ./cpu D &
[1] 7353
[2] 7354
[3] 7355
[4] 7356
A
B
D
C
A
B
D
C
A
C
B
D
...
```

〈그림 5.2〉 **동시에 많은 프로그램 실행시키기**

이제 좀 더 흥미로워졌다. 프로세서가 하나밖에 없음에도 프로그램 4개 모두 동시에

실행되는 것처럼 보인다! 어떻게 이런 마술 같은 일이 일어날까?[4]

하드웨어의 도움을 받아 운영체제가 시스템에 매우 많은 수의 가상 CPU가 존재하는 듯한 **환상**(**illusion**)을 만들어 낸 것이다. 하나의 CPU 또는 소규모 CPU 집합을 무한 개의 CPU가 존재하는 것처럼 변환하여 동시에 많은 수의 프로그램을 실행시키는 것을 **CPU 가상화**(**virtualizing the CPU**)라 한다. 이 책의 첫 부분의 주제이다.

프로그램을 실행하고, 멈추고, 어떤 프로그램을 실행시킬 것인가를 운영체제에게 알려주기 위해서는 원하는 바를 운영체제에 전달할 수 있는 인터페이스 (API)가 필요하다. 이 책 전반에 걸쳐 이런 API들에 대해 논의한다. API는 운영체제와 사용자가 상호작용할 수 있는 주된 방법이다.

다수의 프로그램을 동시에 실행시킬 수 있는 기능은 새로운 종류의 문제를 발생시킨다는 것을 인지했을 것이다. 예를 들어, 특정 순간에 두 개의 프로그램이 실행되기를 원한다면, *누가 실행되어야 하는가*? 이 질문은 운영체제의 정책 (policy)에 달려있다. 운영체제의 여러 부분에서 이러한 유형의 문제에 답하기 위한 정책들이 사용된다. 운영체제가 구현한 동시에 다수의 프로그램을 실행시키는 **기본적인 기법**(**mechanism**)에 대해 다룰 것이다. 즉, **자원 관리자**로서의 운영체제의 역할을 다룬다.

```
1  #include <unistd.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include "common.h"
5  int
6  main(int argc, char *argv[])
7  {
8      int *p = malloc(sizeof(int)); // a1
9      assert(p != NULL);
10     printf("(%d) memory address of p: %08x\n",
11         getpid(), (unsigned) p); // a2
12     *p = 0; // a3
13     while (1) {
14         Spin(1);
15         *p = *p + 1;
16         printf("(%d) p: %d\n", getpid(), *p); // a4
17     }
18     return 0;
19 }
```

〈그림 5.3〉 **메모리 접근 프로그램 (mem.c)**

5.2 메모리 가상화

메모리에 대해 생각해 보자. 현재 우리가 사용하고 있는 컴퓨터에서의 **물리 메모리**(**physical memory**) 모델은 매우 단순하다. 바이트의 배열이다. 메모리를 읽기 위

4) & 문자를 사용하여 동시에 4개의 프로그램을 실행시키는 방법에 주목하자. 이렇게 하면 tcsh 쉘 상에서 작업을 백그라운드로 실행시킬 수 있다. 이는 사용자가 다음 명령어를 곧바로 실행시킬 수 있다는 것을 의미하고, 이 경우에는 다른 인스턴스를 실행시켰다. 명령어 사이의 세미콜론은 `tcsh`에서 동시에 여러 프로그램을 실행시킬 수 있게 한다. `bash`와 같은 다른 쉘을 사용하고 있다면 약간은 다르게 실행시켜야 한다. 자세한 사항에 대해서는 온라인 문서를 읽기 바란다.

해서는 데이터에 **주소(address)**를 명시해야 한다. 메모리에 **쓰기** (혹은 **갱신**) 위해서는 주소와 데이터를 명시해야 한다.

메모리는 프로그램이 실행되는 동안 항상 접근된다. 프로그램은 실행 중에 자신의 모든 자료 구조를 메모리에 유지하고 load와 store 또는 기타 메모리 접근을 위한 명령어를 통하여 자료 구조에 접근한다. 명령어 역시 메모리에 존재한다는 사실을 잊지 말자. 명령어를 반입할 때마다 메모리가 접근된다.

`malloc()`을 호출하여 메모리를 할당하는 그림 5.3의 프로그램을 살펴보자. 이 프로그램의 출력은 다음과 같다.

```
prompt> ./mem
(2134) memory address of p: 00200000
(2134) p: 1
(2134) p: 2
(2134) p: 3
(2134) p: 4
(2134) p: 5
^C
```

이 프로그램은 몇 가지 작업을 수행한다. 우선 메모리를 할당받는다 (a1 행). 그런 후 할당받은 메모리의 주소를 출력한다 (a2). 새로 할당받은 메모리의 첫 슬롯에 숫자 0을 넣는다 (a3). 마지막으로 루프로 진입하여 1초 대기 후, 변수 `p`가 가리키는 주소에 저장되어 있는 값을 1 증가시킨다. 출력할 때마다 실행 중인 프로그램의 프로세스 식별자(PID)라 불리는 값이 함께 출력된다. PID는 프로세스의 고유의 값이다.

```
prompt> ./mem &; ./mem &
[1] 24113
[2] 24114
(24113) memory address of p: 00200000
(24114) memory address of p: 00200000
(24113) p: 1
(24114) p: 1
(24114) p: 2
(24113) p: 2
(24113) p: 3
(24114) p: 3
(24113) p: 4
(24114) p: 4
...
```

〈그림 5.4〉 **메모리 프로그램 여러 번 실행하기**

첫 번째 실행 결과는 별로 흥미롭지 않다. 새로 할당된 메모리의 주소는 `00200000`이다. 프로그램이 진행되면서 천천히 값을 갱신하고 그 결과를 출력한다.

같은 프로그램을 여러 번 실행시켜 보자(그림 5.4). 프로그램들은 같은 주소에 메모리를 할당받지만 (`00200000`), 각각이 독립적으로 `00200000` 번지의 값을 갱신한다. 각 프로그램은 물리 메모리를 다른 프로그램과 공유하는 것이 아니라 각자 자신의 메모리를 가지고 있는 것처럼 보인다[5].

5) 이 예제 코드가 동작하려면, 주소 공간 난수화 기능을 꺼야 한다. 앞으로 보겠지만 난수화는 특정 유형의 보안 취약점을 방어하는 데 좋은 방어법이다. 스택-스매싱 공격(stack-smashing attack)을 이용하여

운영체제가 **메모리 가상화**(**virtualizing memory**)를 하기 때문에 이런 현상이 생긴다. 각 프로세스는 자신만의 **가상 주소 공간**(**virtual address space**, 때로 그냥 **주소 공간**(**address space**)이라고 불림)을 갖는다. 운영체제는 이 가상 주소 공간을 컴퓨터의 물리 메모리로 매핑(mapping)한다. 하나의 프로그램이 수행하는 각종 메모리 연산은 다른 프로그램의 주소 공간에 영향을 주지 않는다. 실행 중인 프로그램의 입장에서는, 자기 자신만의 물리 메모리를 갖는 셈이다. 실제로는 물리 메모리는 공유 자원이고, 운영체제에 의해 관리된다. 이러한 일들이 정확히 어떻게 일어나는지 역시 이 책의 첫 주제 **가상화**(**virtualization**)에 포함된다.

5.3 병행성

이 책의 다른 주요 주제는 **병행성**(**concurrency**)이다. 프로그램이 한 번에 많은 일을 하려 할 때 (즉, 동시에) 발생하는 그리고 반드시 해결해야 하는 문제들을 가리킬 때 이 용어를 사용한다. 병행성 문제는 우선 운영체제 자체에서 발생한다. 가상화에 관한 앞의 예에서 알 수 있듯이 운영체제는 한 프로세스 실행, 다음 프로세스, 또 다음 프로세스 등의 순서로 여러 프로세스를 실행시켜 한 번에 많은 일을 한다. 이러한 행동은 심각하고 흥미로운 문제를 발생시킨다.

병행성 문제는 운영체제만의 문제는 아니다. **멀티 쓰레드** 프로그램도 동일한 문제를 드러낸다. **멀티 쓰레드** 프로그램을 예로 들어 확인해 보자 (그림 5.5).

기본 아이디어는 간단하다(이에 대해서는 이 책의 병행성 부분에서 더 자세히 배우게 될 것이다). 메인 프로그램은 `Pthread_create()`를 사용하여 두 개의 쓰레드를 생성한다[6]. 쓰레드를 동일한 메모리 공간에서 함께 실행 중인 여러 개의 함수라고 생각할 수 있다. 이 예제 코드에서 각 쓰레드는 `worker()`라는 루틴을 실행한다. `worker()` 루틴은 `loops`번 만큼 루프를 반복하면서 카운터 값을 증가시킨다.

`loops` 변수를 1000으로 설정하여 프로그램을 실행시켰을 때 어떤 일이 일어나는지 아래에 나와 있다. `loops` 값은 각 쓰레드가 카운터를 증가시키는 횟수다. `loops` 값을 1000으로 지정한 후 프로그램을 실행시키면, `counter` 변수의 최종 값은 얼마가 될까?

```
prompt> gcc -o thread thread.c -Wall -pthread
prompt> ./thread 1000
Initial value : 0
Final value   : 2000
```

예상했겠지만 각 쓰레드가 1000번씩 counter 값을 증가시켰기 때문에 counter의 최종 값은 2000이 된다. 사실 `loops` 값이 N이면 프로그램의 최종 출력은 $2N$이 되리라고 예상할 수 있다. 곧 밝혀지겠지만 인생은 그리 간단하지 않다. 동일한 프로그램을 다시 실행시켜 보자. 이번에는 `loops`의 값을 더 큰 값으로 지정해 보자.

컴퓨터 시스템에 침입하는 법에 대해 알아보고 싶은 사람은 스스로 관련 자료를 읽어 보기 바란다. 비록 우리는 그러한 공격을 추천하지는 않지만...

6) 실제는 소문자로 시작하는 `pthread_create()`가 호출되어야 한다. 대문자로 시작하는 버전은 `pthread_create()`를 호출하고 반환 코드가 호출 성공을 나타내는지 검사하도록 우리가 만든 래퍼 (wrapper) 함수이다. 자세한 내용은 코드를 확인하기 바란다.

```
#include <stdio.h>
#include <stdlib.h>
#include "common.h"
volatile int counter = 0;
int loops;
void *worker(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        counter++;
    }
    return NULL;
}

int
main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "usage: threads <value>\n");
        exit(1);
    }
    loops = atoi(argv[1]);
    pthread_t p1, p2;
    printf("Initial value : %d\n", counter);

    Pthread_create(&p1, NULL, worker, NULL);
    Pthread_create(&p2, NULL, worker, NULL);
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("Final value : %d\n", counter);
    return 0;
}
```

〈그림 5.5〉 **멀티 쓰레드 프로그램 (threads.c)**

```
prompt> ./thread 100000
Initial value : 0
Final value   : 143012 // 어?
prompt> ./thread 100000
Initial value : 0
Final value   : 137298 // 뭐라고?
```

이번 실행에서는 입력 값을 100,000으로 주었더니 최종 값이 200,000이 아니라 143,012가 되었다. 다시 한 번 동일한 조건으로 실행시켰을 때에는 또 잘못된 값이 출력되었을 뿐 아니라 직전 실행과도 다른 결과가 출력되었다. 사실 `loops` 값을 큰 값으로 지정하여 반복해서 실행시키면 몇 번은 올바른 결과가 출력된다는 것을 발견할 수 있을 것이다. 왜 이런 일이 생기는 걸까?

예상하지 못한 결과의 원인은 명령어가 한 번에 하나씩만 실행된다는 것과 관련 있다. 앞 프로그램의 핵심 부분인 counter를 증가시키는 부분은 세 개의 명령어로 이루어진다. counter 값을 메모리에서 레지스터로 탑재하는 명령어 하나, 레지스터를 1 증가시키는 명령어 하나, 레지스터의 값을 다시 메모리에 저장하는 명령어 하나 이렇게 3개의 명령어로 구성된다. 이 세 개의 명령어가 **원자적**(**atomically**)으로(한 번에 3개 모두) 실행되지 않기 때문에 이상한 일이 발생할 수 있다. 이 책의 후반부에서 자세하게 논의할 주제가 바로 이 **병행성**(**concurrency**) 문제이다.

핵심 질문: 올바르게 동작하는 병행 프로그램은 어떻게 작성해야 하는가

같은 메모리 공간에 다수의 쓰레드가 동시에 실행한다고 할 때, 올바르게 동작하는 프로그램을 어떻게 작성할 수 있는가? 운영체제로부터 어떤 기본 기법들을 제공받아야 하는가? 하드웨어는 어떤 기능을 제공해야 하는가? 병행성 문제를 해결하기 위하여 기본 기법들과 하드웨어 기능을 어떻게 이용할 수 있는가?

5.4 영속성

이 책의 세 번째 주요 주제는 **영속성**(**persistence**)이다. DRAM과 같은 장치는 데이터를 **휘발성**(**volatile**) 방식으로 저장하기 때문에 메모리의 데이터는 쉽게 손실될 수 있다. 전원 공급이 끊어지거나 시스템이 갑자기 고장나면(crash) 메모리의 모든 데이터는 사라진다. 데이터를 영속적으로 저장할 수 있는 하드웨어와 소프트웨어가 필요하다. 저장 장치는 모든 시스템에 필수적이다.

하드웨어는 **입력/출력**(**input/output**) 혹은 **I/O** 장치 형태로 제공된다. 요즘에는 **solid-state drives**(**SSDs**)가 많이 사용되고 있기는 하지만 장기간 보존할 정보를 저장하는 장치로는 일반적으로 **하드 드라이브**(**hard drive**)가 사용된다.

디스크를 관리하는 운영체제 소프트웨어를 **파일 시스템**(**file system**)이라고 부른다. 파일 시스템은 사용자가 생성한 **파일**(**file**)을 시스템의 디스크에 안전하고 효율적인 방식으로 저장할 책임이 있다.

CPU나 메모리 가상화와는 달리 운영체제는 프로그램 별로 가상 디스크를 따로 생성하지 않는다. 오히려 사용자들이 종종 파일 정보를 공유하기 원한다고 가정한다. 예를 들어, C 프로그램을 작성할 때, 우선 에디터(예, Emacs[7])를 사용하여 C 파일을 생성하고 편집한다(`emacs -nw main.c`). 편집이 끝나면 소스 코드를 컴파일한다(예, `gcc -o main main.c`). 컴파일이 완료되면 새로 생성된 실행 파일을 실행할 수 있다(예, `./main`). 파일이 여러 다른 프로세스 사이에서 공유된다. 우선, emacs라는 편집기가 컴파일러가 사용할 파일을 생성한다. 컴파일러는 입력 파일을 사용하여 새로운 실행 파일을 생성한다(여러 단계를 거쳐서, 자세한 사항은 컴파일러 강좌를 수강하기 바란다). 마지막으로 실행 파일이 실행된다. 자 새로운 프로그램이 탄생했다!

핵심 질문: 데이터를 영속적으로 저장하는 방법은 무엇인가

파일 시스템은 데이터를 영속적으로 관리하는 운영체제의 일부분이다. 올바르게 일하기 위해서는 어떤 기법이 필요할까? 이러한 작업의 성능을 높이기 위해서 어떤 기법과 정책이 필요한가? 하드웨어와 소프트웨어가 실패하더라도 올바르게 동작하려면 어떻게 해야 하는가?

7) 반드시 Emacs를 사용하고 있어야 한다. 만일 vi를 사용하고 있는 중이라면 문제가 있을 가능성이 있다. 코드 에디터가 아닌 일반 편집기를 사용하는 중이라면 문제는 더 심각하다.

코드를 살펴보자. 그림 5.6은 문자열 "hello world"를 포함한 파일 `/tmp/file`을 생성하는 코드이다.

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <assert.h>
4  #include <fcntl.h>
5  #include <sys/types.h>
6  int
7  main(int argc, char *argv[])
8  {
9      int fd = open("/tmp/file", O_WRONLY | O_CREAT | O_TRUNC, S_IRWXU);
10     assert(fd > -1);
11     int rc = write(fd, "hello world\n", 13);
12     assert(rc == 13);
13     close(fd);
14     return 0;
15 }
```

〈그림 5.6〉 **입출력을 수행하는 프로그램 (`io.c`)**

여기서 프로그램은 운영체제를 세 번 호출한다. 첫째, `open()` 콜은 파일을 생성하고 연다. 둘째, `write()` 콜은 파일에 데이터를 쓴다. 셋째, `close()` 콜은 단순히 파일을 닫는데, 프로그램이 더 이상 해당 파일을 사용하지 않는다는 것을 나타낸다. 이들 시스템 콜(**system call**)은 운영체제에서 **파일 시스템**(**file system**)이라 불리는 부분으로 전달된다. 파일 시스템은 요청을 처리하고 경우에 따라 사용자에게 에러 코드를 반환한다.

데이터를 디스크에 쓰기 위해서 운영체제가 실제로 하는 일이 무엇인지 궁금할 것이다. 설명하겠지만, 그렇게 간단하지 않다. 파일 시스템은 많은 작업을 해야 한다. 먼저 새 데이터가 디스크의 어디에 저장될지 결정해야 하고, 파일 시스템이 관리하는 다양한 자료 구조를 통하여 데이터의 상태를 추적해야 한다. 이런 작업을 하기 위해서는 저장 장치로부터, 기존 자료 구조를 읽거나 갱신해야 한다. **장치 드라이버**(**device driver**)[8]를 작성해 본 사람은 알겠지만 당신을 대신하여 장치가 무언가 하게 하는 일은 복잡한 작업이다. 이를 위해서는 저수준의 장치 인터페이스와 그 시맨틱에 대한 깊은 이해가 필요하다. 운영체제는 시스템 콜이라는, 표준화된 방법으로 장치들을 접근할 수 있게 한다. 운영체제는 **표준 라이브러리**(**standard library**)처럼 보이기도 한다.

장치를 접근하는 방법과 파일 시스템이 데이터를 영속적으로 관리하는 방법은 이보다 훨씬 더 복잡하다. 성능향상을 위해서 대부분의 파일 시스템은 쓰기요청을 지연시켜 취합된 요청들을 한 번에 처리한다. 쓰기 중에 시스템의 갑작스런 고장에 대비해서 많은 파일 시스템들이 **저널링**(**journaling**)이나 **쓰기-시-복사**(**Copy-On-Write**)와 같은 복잡한 쓰기 기법을 사용한다. 이런 기법들은 쓰기 순서를 적절히 조정하여 고장이 발생하더라도 정상적인 상태로 복구될 수 있게 한다. 효율적인 디스크 작업을 위해 단순 리스트에서 복잡한 B-트리까지 다양한 종류의 자료 구조를 사용한다. 지금까지 설명한 내용을 이해 못해도 괜찮다! 이 책의 **영속성**(**persistence**) 부분에서 자세히 설명할

8) 디바이스 드라이버는 운영체제 코드의 일부분으로, 특정 장치의 처리를 담당한다. 장치와 장치 드라이버에 대해서는 나중에 더 자세하게 논의할 것이다.

것이다. 장치와 입출력에 관한 전반적인 내용과 디스크, RAID와 파일 시스템에 대해 매우 상세하게 다룰 것이다.

5.5 설계 목표

이제 운영체제가 실제로 어떤 일을 하는지 어느 정도 감을 잡았을 것이다. 운영체제는 CPU, 메모리, 디스크와 같은 물리 **자원**을 **가상화**(**virtualize**)한다. 운영체제는 병행성과 관련된 복잡한 문제를 처리한다. 파일을 **영속적**으로 저장하여 아주 오랜 시간 동안 안전한 상태에 있게 한다. 그런 시스템을 구현하려면 몇 가지를 목표를 세워야 한다. 이러한 목표는 설계와 구현에 집중하고, 필요한 경우 절충안을 찾는 데 필수적이다. 시스템 개발 시 적절한 절충안을 찾는 것은 매우 중요하다.

가장 기본적인 목표는 시스템을 편리하고 사용하기 쉽게 만드는 데 필요한 **개념**(abstraction)[9]들을 정의하는 것이다. 컴퓨터 과학에서 추상화는 모든 일에 근간이다. 추상화를 통해 큰 프로그램을 이해하기 쉬운 작은 부분들로 나누어 구현할 수 있다. 어셈블리 코드를 몰라도 C[10]와 같은 고수준 언어로 큰 프로그램을 작성하는 것이 가능하다. 추상화는 논리 게이트를 고려하지 않고도 어셈블리 코드를 작성할 수 있게 하며, 트랜지스터에 대한 지식없이도 게이트를 이용하여 프로세서를 만들 수 있게 한다. 추상화는 너무 근본이기 때문에 중요성을 쉽게 잊을 수 있다. 이 강의를 진행하는 동안은 잊지 않도록 하자. 각 절에서 오랫동안 개발되어 온 주요 개념을 설명하며, 운영체제의 각 부분을 쉽게 이해할 수 있는 단서를 제공할 것이다.

운영체제의 설계와 구현에 중요한 목표는 **성능**이다. 다른 말로 표현하면 **오버헤드를 최소화**(**minimize the overhead**)하는 것이다. "가상화"와 "사용하기 쉬운 시스템을 만드는것"은 의미가 있지만 반드시 해야 하는 것은 아니다. 가상화 및 다른 운영체제 기능을 과도한 오버헤드 없이 제공해야 한다. 오버헤드는 시간(더 많은 명령어)과 공간(메모리 또는 디스크)의 형태로 나타난다. 시간이나 공간 혹은 둘 다 최소로 하는 해결책을 찾을 것이다. 항상 완벽한 해답을 얻을 수는 없다. 앞으로 완벽한 해결책을 찾는 법과 절충하는 것에 대해 배우게 될 것이다.

또 다른 목표는 응용 프로그램 간의 **보호**, 그리고 운영체제와 응용 프로그램 간의 **보호**이다. 다수 프로그램들이 동시에 실행되기 때문에, 운영체제는 한 프로그램의 악의적인 또는 의도치 않은 행위가 다른 프로그램에게 피해를 주지 않는다는 것을 보장해야 한다. 당연히 응용 프로그램이 운영체제에게 해를 끼치지 않기를 원한다(운영체제에게 해를 가하면 모든 프로그램에게 영향을 주기 때문이다). 보호는 운영체제의 원칙 중 하나인 **고립**(**isolation**) 원칙의 핵심이다. 프로세스를 다른 프로세스로부터 고립시키는 일은 보호의 핵심이고 운영체제가 해야 하는 일 중 많은 부분의 근간이 된다.

운영체제는 계속 실행되어야 한다. 운영체제가 실패하면 그 위에서 실행되는 모든 응용 프로그램도 실패하게 된다. 이러한 종속성 때문에 운영체제는 높은 수준의 **신뢰**

9) 역자 주: abstraction은 추상화라는 사전적 의미보다 "개념"에 더 가까움.

10) 어떤 일부는 C를 고수준 언어라고 부르는 데에 반대할지 모른다. 그러나 이 수업은 운영체제 수업이라는 것을 기억하라. 우리는 단지 어셈블리 언어로만 코딩하지 않아도 된다는 것만으로도 너무 행복하다.

성(**reliability**)을 제공해야 한다. 운영체제가 복잡해질수록 (때론 수백만 라인의 코드로 이루어짐), 신뢰성 있는 운영체제를 구현하는 일이 매우 어려워진다. 운영체제 분야의 많은 연구(우리 연구를 포함하여 [Bai+09; Sun+10])들이 바로 이 문제에 초점을 맞추고 있다.

다른 중요한 목표들도 존재한다. **에너지-효율성**(**energy-efficiency**)은 녹색 세상을 위해 중요하다. 악의적인 응용 프로그램에 대한 **보안**(**security**, 사실은 보호의 확장)은 현재와 같은 네트워크 환경에서 특히 중요하다. **이동성**(**mobility**)은 운영체제가 작은 장치에서 사용될수록 중요해지고 있다. 시스템의 목적에 따라, 운영체제는 다른 목표를 지향하게 되고, 구현이 달라진다. 이 책에서 다루는 많은 주제들은 다양한 기기에서 모두 유용하다.

5.6 역사 약간

운영체제 소개를 마치기 전에 운영체제가 발전해 온 간단한 역사를 살펴보기로 하자. 인간이 만든 다른 것들(예: 자동차, 신발 등 아무거나)과 마찬가지로 이전의 사례로부터 학습하기 때문에, 시간이 지나면서 좋은 아이디어가 운영체제에 축적되었다. 몇 가지 주요 발전에 대해서 논의한다. 더 자세한 내용은 Brinch Hansen의 훌륭한 운영체제의 역사를 보기 바란다 [Han00].

초창기 운영체제: 단순 라이브러리

초창기 운영체제는 그렇게 많은 일을 하지 않았다. 기본적으로는 자주 사용되는 함수들을 모아 놓은 라이브러리에 불과했다. 예를 들어 프로그래머들이 각자 저수준 입출력 처리 코드를 작성하는 것이 아니라 운영체제가 그런 API를 제공하였다. 이러한 방식은 개발자들의 삶을 편하게 해준다.

옛날 메인프레임 시스템에서는 컴퓨터를 조작하는 사람이 프로그램을 한 번에 하나씩 실행하였다. 현대 운영체제가 하는 많은 작업, 예를 들어 작업의 순서를 정하는 것과 같은 것들을 컴퓨터 관리자가 담당했다. 현명한 개발자라면 컴퓨터 관리자에게 잘 보여야 한다. 그래야 당신의 작업을 우선 실행해주니까 말이다.

작업들이 준비되면 컴퓨터 관리자가 일괄적으로 처리한다. 이러한 방식의 컴퓨팅을 **일괄 처리**(**batch**)라고 부른다. 그 당시 컴퓨터는 비용 때문에 대화 방식으로 사용되지 않았다. 컴퓨터가 너무 고가였기 때문에 사용자가 그 앞에 앉아서 사용하게 할 수 없었다. 아무 것도 하지 않은 채 대부분의 시간을 보내게 되고 그러면 설치 기관은 시간당 수만 달러씩 낭비하게 되기 때문이다 [Han00].

라이브러리를 넘어서: 보호

단순한 라이브러리를 넘어서 운영체제는 컴퓨터 관리 면에서 더 중심적인 역할을 하게 된다. 이러한 방식이 등장하게 된 원인 중 하나는 운영체제가 실행하는 코드는 특별하다는 인식이었다. 운영체제 코드는 장치를 제어하였 때문에 일반 응용 프로그램 코드와는

다르게 취급되어야 한다. 왜 이래야 할까? 모든 응용 프로그램이 디스크에서 원하는 지점을 읽을 수 있다고 상상해 보자. 어떤 프로그램이든 원하는 모든 파일을 읽을 수 있기 때문에 사생활이라는 개념은 사라지게 될 것이다. 때문에, 라이브러리 형태로 **파일 시스템**을 구현하는 것은 의미가 없다. 다른 대체 방안이 필요하게 되었다.

Atlas 컴퓨팅 시스템 [Kil+61; Lav78]에 의해 **시스템 콜**이라는 아이디어가 발명되었다[11]. 운영체제를 라이브러리가 아니라 특별한 하드웨어 명령어와 하드웨어 상태를 결합하여 운영체제로 전환하기 위해서는 정해진 규칙에 따라 제어 가능한 과정을 거치도록 만들었다.

시스템 콜과 프로시저 호출의 결정적 차이는 시스템 콜은 제어를 운영체제에게 넘길 때 (예, 분기) **하드웨어 특권 수준**(**hardware privilege level**)을 상향 조정한다는 것이다. 사용자 응용 프로그램은 **사용자 모드**(**user mode**)라고 불리는 상태에서 실행된다. 사용자 모드에서는 응용 프로그램이 할 수 있는 일을 하드웨어적으로 제한한다. 예를 들어 사용자 모드에서 실행 중인 응용 프로그램은 디스크 입출력, 물리 메모리 페이지 접근 또는 네트워크 패킷 송신 등의 작업을 할 수 없다. 시스템 콜은 보통 **trap**이라고 불리는 특별한 하드웨어 명령어를 이용하여 호출된다. 시스템 콜 시작 시, 하드웨어는 미리 지정된 **트랩 핸들러**(**trap handler**) 함수에게 제어권을 넘기고 특권 수준을 **커널 모드**(**kernel mode**)로 격상시킨다. 트랩 핸들러 함수는 운영체제가 미리 구현해 놓는다. 커널 모드에서 운영체제는 시스템의 하드웨어를 자유롭게 접근할 수 있으며 입출력 또는 메모리 할당 등과 같은 작업을 할 수 있다. 운영체제가 서비스를 완료하면 **return-from-trap** 특수 명령어를 사용하여 제어권을 다시 사용자에게 넘긴다. 이 명령어는 응용 프로그램이 출발했던 지점으로 제어권을 넘기는 동시에 사용자 모드로 다시 전환한다.

멀티프로그래밍 시대

운영체제 다운 운영체제의 실제 등장은 메인프레임 이후 **미니컴퓨터**(**minicompu-ter**) 시대에 이루어졌다. Digital Equipment 사의 PDP 계열 컴퓨터의 등장으로 컴퓨터 가격이 크게 낮아졌다. 큰 기관이 하나의 메인프레임 컴퓨터를 갖는 대신 기관의 소그룹마다 하나의 컴퓨터를 가질 수 있었다. 컴퓨터 가격의 하락으로 개발자들의 활동이 활발해졌다[12]. 더 많은 유능한 인력이 컴퓨터 분야로 모여 들었고 컴퓨터 시스템이 더 흥미롭고 멋진 일들을 하게 만들었다.

컴퓨터 자원의 효율적 활용을 위해 **멀티프로그래밍**(**multiprogramming**)기법이 대중으로 사용되었다. 한 번에 하나의 프로그램만 실행시키는 대신 운영체제는 여러 작업을 메모리에 탑재하고 작업들을 빠르게 번갈아 가며 실행하여 CPU 사용률을 향상시킨다. 입출력 장치가 느리기 때문에 전환(switching) 능력이 특히 중요하였다. 입출력 요청이 서비스 되고 있는 동안 CPU가 대기하는 것은 CPU 시간 낭비를 초래한다. 대신 그동안 다른 작업으로 전환하여 잠시라도 실행하는 것이 낫지 않을까?

11) 역자 주: 실로 역사적인 사건이 아닐 수 없다.
12) 역자 주: 매우 중요한 현상이다.

멀티프로그래밍 지원의 필요와 인터럽트를 통한 입출력 작업 처리 등이 운영체제의 발전에 여러 가지의 혁신을 가져왔다. **메모리 보호**(**memory protection**)와 같은 주제가 중요하게 되었다. 한 프로그램이 다른 프로그램의 메모리에 접근하는 것을 원하지 않는다. 멀티프로그래밍에서 발생하는 **병행성**(**concurrency**) 문제에 대한 이해도 중요하다. 인터럽트가 발생하더라도 운영체제가 올바르게 동작한다는 것을 보장하는 것도 매우 어려운 일이다. 이와 관련된 여러 주제에 대해서는 이 책의 후반부에서 살펴본다.

그 당시 사건 중 하나는 Unix 운영체제의 등장이었다. Unix 운영체제는 Bell 연구소(예, 전화 회사 맞음)의 Ken Thompson과 Dennis Ritchie가 만들었다. Unix는 다른 운영체제로부터 좋은 아이디어를 많이 도입하였다. 특히 Multics [Org72], TENEX [Bob+72]와 Berkeley Time-Sharing System [Inc68] 같은 시스템으로부터도 아이디어를 받아들였다. Unix는 이 아이디어들을 더욱 단순하고 사용하기 쉽도록 변형하였다. 얼마 있어 이 팀은 Unix 소스 코드가 담긴 테이프를 전세계 사람들에게 보내기 시작했다. 소스 코드를 받은 사람 중 많은 사람들이 구현에 참여했으며, 시스템에 새로운 기능을 추가하였다. 더 자세한 사항에 대해서는 다음 페이지의 **여담** 상자를 보기 바란다[13].

현대

미니컴퓨터 시대를 지나 더 싸고, 더 빠르고 대중적인 컴퓨터가 등장하였다. 현재 **개인용 컴퓨터**(**personal computer**) 또는 **PC**라고 불리는 컴퓨터이다. Apple사의 초기 컴퓨터와 (예를 들면 Apple II) IBM PC에 의해 주도된 이 새로운 종류의 컴퓨터는 그룹당 하나의 미니컴퓨터 대신 책상 마다 하나의 컴퓨터를 놓을 수 있을만큼 가격이 쌌기 때문에 컴퓨팅의 주도 세력이 되었다.

불행하게도 운영체제의 입장에서는 PC의 등장이 퇴보를 의미하였다. 초기 PC들은 미니컴퓨터 기술로부터 얻었던 소중한 경험들을 무시하거나 아예 이에 대해 무지했기 때문이다. 예를 들어, **DOS**(**Disk Operating System**, **Microsoft** 제품) 같은 초기 운영체제는 메모리 보호가 중요하다는 생각조차 하지 않았다. 악의적인 혹은 잘못 개발된 응용 프로그램이 메모리 전체를 손상시킬 수 있었다. 1세대 **Mac OS**(v9과 그 이전 버전)는 작업 스케줄링에 협업 스케줄링(Cooperative Scheduling)을 채용하였다. 쓰레드가 우연히 무한루프에 빠지면, 전체 시스템을 정지시키게 되고, 결국 재부팅을 할 수밖에 없었다. 이 세대에 사라져버린 운영체제 기술들이 너무 많아 여기서 전부 논의할 수 없을 정도이다.

다행히 암흑기가 지난 뒤 예전 미니컴퓨터 운영체제 기법들이 데스크톱 컴퓨터용 운영체제에 등장하기 시작했다. 예를 들면, Mac OS X는 Unix를 근간으로 하며 Unix의 모든 특성을 가지고 있다. Windows 역시 컴퓨팅 역사의 위대한 아이디어 중 많은 것들을 채택하였다. 이러한 노력은 특히 Widows NT부터 시작되었는데, NT는 Microsoft

13) 우리는 여담 상자와 다른 관련 텍스트 상자들을 텍스트의 주제와는 맞지 않는 다양한 사항에 대해 주의를 환기하기 위해 사용한다. 때론 단지 농담하기 위해서도 그 상자들을 사용할 것이다. 진행하면서 약간의 재미를 즐기는 것은 괜찮지 않나? 예, 많은 농담들이 그리 재미있지는 않다.

여담: Unix의 중요성

운영체제의 역사에서 UNIX의 중요성은 아무리 강조해도 지나치지 않다. 초기 시스템들로부터, 특히 MIT의 **Multics**는 위대한 아이디어들을 집대성하여 간단하고 강력한 시스템을 만들었다.

"Bell Labs"에서 시작된 UNIX의 핵심은 서로 통합 사용이 가능한 작고 강력한 프로그램들이다. 명령어를 입력하는 **쉘**(**shell**)은 그러한 메타-수준 프로그래밍을 할 수 있는 **pipes**와 같은 기본 기법을 제공한다. 프로그램을 연결하는 것이 용이하다. 예를 들어, 텍스트 파일에서 "foo"라는 단어를 포함하고 있는 행을 찾고 행의 개수를 세고 싶은 경우, 다음과 같이 입력하면 된다: `grep foo file.txt | wc -l`. `grep`과 `wc`(word count) 프로그램을 이용하여 목적을 달성할 수 있다. UNIX 환경은 프로그래머와 개발자에 친화적이며 새로운 C 언어를 위한 컴파일러를 제공하였다. 프로그래머가 프로그램을 쉽게 작성하고 공유할 수 있었기 때문에 UNIX는 엄청난 인기를 끌었다. 저자들이 소스 코드를 요청한 누구에게나 공짜로 배포한 것도 크게 한 몫 하였다. 이는 **공개-소스 소프트웨어**(**open-source software**)의 초창기 형태라고 할 수 있다.

코드를 쉽게 접할 수 있고 가독성이 좋았다는 것도 매우 중요한 특징이다. C 언어로 작성된 아름답고 작은 크기의 커널은 많은 사람들이 커널에 새로운 기능과 멋진 특징을 추가할 수 있게 하였다. 예를 들어, **Bill Joy**가 이끌었던 Berkeley의 사업화 그룹은 환상적인 배포판(**Berkeley Systems Distribution**, **BSD**)을 만들었다. 이 배포판은 발전된 가상 메모리, 파일 시스템 및 네트워킹 서브시스템을 가지고 있었다. Joy는 나중에 **Sun Microsystems**를 설립하였다.

불행하게도 회사들이 소유권을 주장하고 이익을 추구하면서 조금씩 느려졌다. 변호사가 관여하면 항상 발생하는 흔한 결과였다. 많은 회사들이 자신의 고유 버전을 소유하고 있었다. Sun Microsystems의 **SunOS**, IBM의 **AIX**, HP의 **HPUX**("H-Pucks"라고도 불림), SGI의 **IRIX** 등이 대표적인 예이다. AT&T/Bell Labs와 다른 회사 간의 법적 다툼은 UNIX에 먹구름을 드리웠다. 많은 사람들은 Windows가 등장하여 PC 시장을 장악함에 따라 UNIX가 살아남을 수 있을까 하는 의문을 가지게 되었다.

운영체제 기술을 크게 발전시켰다. 오늘날 이동전화조차 Linux와 같은 운영체제를 실행하고 있다. 이 운영체제는 1980년대 PC 용 운영체제보다 1970년대 미니컴퓨터 용 운영체제에 가깝다 (정말 다행이다). 운영체제 전성기에 개발된 좋은 아이디어들이 현재 시스템에서도 여전히 적용되는 것을 보면 사뭇 즐겁다. 더욱이 이 아이디어들은 계속 발전하고 있다. 이러한 발전으로 운영체제에 많은 기능들이 추가되고 사용하기 좋은 시스템이 되고 있다.

5.7 요약

운영체제에 대한 소개를 마쳤다. 오늘날 운영체제는 시스템을 사용하기 쉽게 만들었다. 거의 모든 운영체제가 이 책에서 논의하게 될 기술을 사용한다.

시간적인 제약 때문에 이 책에서 다루지 못한 운영체제 내용이 존재한다. 예를

여담: 그리고 LINUX가 나왔다

UNIX에게는 다행하게도 **Linus Torvalds**라는 이름의 젊은 핀란드 해커가 자신만의 UNIX 버전을 개발하기로 마음 먹었다. UNIX의 원리와 개념은 사용하였지만 코드는 사용하지 않아 법적 문제는 발생하지 않았다. 그는 전세계의 개발자들에게 도움을 요청하였고 곧 **Linux**가 탄생하였다. 동시에 현재의 공개-소스 소프트웨어 운동도 시작되었다.

인터넷 시대가 도래함에 따라 Google, Amazon, Facebook 등 대부분의 회사가 공짜이면서 자신들의 입맛대로 쉽게 수정할 수 있는 Linux를 사용하기로 결정하였다. 사실 이런 시스템이 없었다면 이런 새 회사들의 성공은 상상하기 힘들다. 스마트폰(Android)이 사용자가 직접 사용하는 주된 플랫폼이 되면서 역시 같은 이유 때문에 Linux는 입지를 확장하였다. 그리고 Steve Jobs는 자신의 UNIX-기반 NeXTStep 운영 환경을 Apple에 가지고 가서 UNIX를 데스크톱에서도 대중적으로 만들었다. 많은 Apple 사용자들은 이러한 사실 조차 알지 못할 것이다. 이리하여 UNIX는 그 어느 때보다도 건재하다. 이런 환상적인 결과에 대해 감사해야 한다.

들면, 운영체제에는 많은 **네트워킹** 코드가 있다. 네트워킹 수업의 수강 여부는 당신에게 맡기겠다. 비슷하게 **그래픽**(**graphic**) 장치는 특히 중요하다. 이 분야의 지식을 쌓고 싶으면, 그래픽 수업을 듣기 바란다. 마지막으로 어떤 운영체제 교재는 **보안**을 상세히 다룬다. 운영체제는 실행 중인 프로그램들 간에 보호를 제공해야 하고 응용 프로그램들이 자신의 파일을 보호할 수 있는 능력을 주어야 한다는 의미에서 보안을 다룬다. 그러나 보안 수업에서 발견할 수 있는 심화된 보안 쟁점에 대해서는 깊이 들어가지 않는다.

다루어야 할 중요한 주제가 많다. CPU와 메모리 가상화의 기본 원리, 병행성 및 장치와 파일 시스템을 통한 영속성이 바로 그것이다. 걱정하지 말기 바란다! 공부해야 할 것이 많지만 강좌가 끝나면 컴퓨터 시스템의 실제 동작 방식에 관한 새로운 이해를 가지게 될 것이다. 자 출발하자!

참고 문헌

[Bai+09] **"Tolerating File-System Mistakes with EnvyFS"**
LakshmiN. Bairavasundaram, Swaminathan Sundararaman, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau
USENIX '09, San Diego, CA, June 2009
동시에 여러 파일 시스템을 사용하면 그 중 하나의 실수를 감내할 수 있다는 재미있는 논문

[Bob+72] **"TENEX, A Paged Time Sharing System for the PDP-10"**
Daniel G. Bobrow, Jerry D. Burchfiel, Daniel L. Murphy, and Raymond S. Tomlinson
CACM, Volume 15, Number 3, March 1972
TENEX는 현대 운영체제에서 발견되는 많은 기법을 가지고 있다; 얼마나 많은 혁신이 이미 1970년대에 일어났는지 알고 싶은 사람은 읽어 보기 바란다.

[BO10] **"Computer Systems: A Programmer's Perspective"**
Randal E. Bryant and David R. O'Hallaron
Addison-Wesley, 2010
컴퓨터 시스템이 어떻게 동작하는가에 관한 또 하나의 훌륭한 소개. 이 책과 약간 겹치는 부분이 존재하므로 원한다면 그 책의 마지막 몇 장을 건너 뛰거나 또는 그 책을 읽어서 같은 주제에 대해 다른 시각을 경험하기 바란다. 결국 자신만의 지식을 쌓는 좋은 방법은 가능한 많은 다른 사람의 관점을 듣고 자신만의 의견과 생각을 개발해야 함. 생각 또 생각.

[Han00] **"The Evolution of Operating Systems"**
P. Brinch Hansen
In Classic Operating Systems: From Batch Processing to Distributed Systems Springer-Verlag, New York, 2000
이 에세이는 역사적으로 중요한 시스템에 관한 논문 모둠을 소개한다.

[Inc68] **"SDS 940 Time-Sharing System"**
Scientific Data Systems Inc.
TECHNICAL MANUAL, SDS 90 11168 August 1968
URL: `http://goo.gl/EN0Zrn`
예, 찾을 수 있는 가장 좋은 기술 매뉴얼. 옛날 시스템 문서를 읽고 이미 1960년대 후반에 얼마나 많은 것들이 나타났는지 알아보는 것은 환상적인 일이다. Berkeley Time-Sharing System(후에 SDS 시스템이 됨)을 만든 사람 중에 하나는 Butler Lampson이었다. 그는 나중에 시스템 분야에 공헌한 공로로 Turing-award를 수상한다.

[Kil+61] **"One-Level Storage System"**
T. Kilburn, D.B.G. Edwards, M.J. Lanigan, and F.H. Sumner
IRE Transactions on Electronic Computers, April 1962
Atlas는 현대 시스템에서 볼 수 있는 많은 작품들을 만들었다. 그러나 이 논문이 최고는 아니다. 하나만 읽어야 한다면 아래의 역사적 관점을 설명한 책 [Lav78]을 읽는 것이 낫다.

[Lav78] **"The Manchester Mark I and Atlas: A Historical Perspective"**
S.H. Lavington
Communications of the ACM archive Volume 21, Issue 1 (January 1978), pages 4-12
컴퓨터 시스템의 초창기 발전 역사와 Atlas의 선구적 노력에 관한 좋은 논문. 물론 Atlas 원 논문을 읽는 것도 괜찮지만 이 논문은 좋은 개관과 역사적 관점을 추가하였다.

[Org72] **"The Multics System: An Examination of its Structure"**
Elliott Organick

*Multics의 좋은 개괄. 너무 좋은 아이디어들이 많았지만 과설계된 시스템. 너무 많은 목적을 가지고 있어 실제로는 하나도 제대로 동작하지 않음. Fred Brooks가 "second-system effect"라고 부른 고전적인 예 [**B75**].*

[PP03] **"Introduction to Computing Systems: From Bits and Gates to C and Beyond"**
Yale N. Patt and Sanjay J. Patel
McGraw-Hill, 2003
컴퓨팅 시스템에 대한 소개로 저자들이 좋아하는 책 중 하나. 트랜지스터와 게이트로부터 출발하여 C까지 차례대로 설명함. 앞 부분 내용이 특히 좋음.

[RT74] **"The Unix Time-Sharing System"**
Dennis M. Ritchie and Ken Thompson
CACM, Volume 17, Number 7, July 1974, pages 365-375
컴퓨팅 세계를 장악하고 나서 개발자들에 의해 쓰여진 훌륭한 요약

[Sun+10] **"Membrane: Operating System Support for Restartable File Systems"**
Swaminathan Sundararaman, Sriram Subramanian, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift
FAST '10, San Jose, CA, February 2010
강의 노트를 쓸 때 좋은 점은 당신의 연구를 홍보할 수 있다는 것이다. 그러나 이 논문은 실제 매우 잘된 논문이다. 파일 시스템이 버그를 만나 갑자기 고장날 때, Membrane은 마술과 같이 재시동 된다. 이때 응용 프로그램이나 시스템의 다른 부분에는 전혀 영향을 주지 않는다.