과목명	운영체제
담당교수	최종무 교수님
학과	소프트웨어학과
학번	32153180
이름	이상민
제출일자	2019.03.19

2. 운영체제 개요(Introduction to Operating Systems)

프로그램이 하는 일은 단지 명령어를 실행하는 것으로 아주 단순하다. 즉, 초당 수백만에서 수십억 번 명령어를 반입(fetch)하고, 어떠한 명령어인지 해석(decode)하고, 실행(execute)한다. 이 과정을 프로그램이 완전히 종료될 때까지 계속한다. 이것이 본 노이만(Von Neumann) 컴퓨터 모델의 기초이다.

프로그램을 쉽게 실행하고, 프로그램 간의 메모리 공유를 가능케 하고, 여러 장치와 상호작용을 가능케 하는 등 다양한 일을 할 수 있게 하는 소프트웨어가 있다. 그것을 운영체제(OS, operating system)이라고 한다.

운영체제는 위와 같은 일을 하기 위해서 가상화(virtualization) 기법을 사용한다. 그렇기 때문에 운영체제를 가상 머신(virtual machine)이라고 부른다.

가상화는 많은 프로그램들이 CPU를 공유하여 동시에 실행될 수 있도록 한다. 프로그램들이 각자 명령어와 데이터를 접근할 수 있게 해주고, 디스크 등의 장치를 공유할 수 있게 해준다. 그래서 운영체제는 자원 관리자(resource manager)라고도 불린다.

2.1 CPU 가상화(Virtualizing the CPU)

```c
#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;
}
```

위의 프로그램을 여러 인스턴스를 동시에 실행시킨 결과는 아래와 같은데, 프로세서는 하나뿐인데 프로그램 4개 모두 동시에 실행되는 것처럼 보인다.

```
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
```

하드웨어의 도움을 받아 운영체제가 시스템에 수많은 가상 CPU가 존재하는 듯한 환상(illusion)을 만들어 낸 것이다. 하나의 CPU를 무한대의 CPU가 존재하는 것처럼 변환하여 동시에 여러 프로그램을 실행시키는 것을 CPU 가상화라고 한다.

다수의 프로그램을 동시에 실행시킬 수 있는 기능은 새로운 종류의 문제를 발생시킨다. 예를 들어, '어느 한순간에 두 개의 프로그램이 실행되기를 원한다면 누가 먼저 실행되어야 하는가' 하는 문제이다. 운영체제의 여러 부분에서 이러한 문젱 답하기 위한 정책(policy)이 사용된다. 운영체제가 구현한 동시에 다수의 프로그램을 실행시키는 기본적인 기법(mechanism)에 대해 다룰 것이다.

2.2 메모리 가상화(Virtualizing memory)

현재 우리가 사용하고 있는 물리 메모리(physical memory) 모델은 매우 단순히 바이트의 배열이다. 메모리를 읽기 위해서는 데이터에 주소(address)를 명시해야 한다.

메모리는 프로그램이 실행되는 동안 항상 접근된다. 프로그램은 실행 중 자신 자신의 모든 자료 구조를 메모리에 유지하고 load, store 또는 기타 명령어를 통하여 자료 구조에 접근한다. 명령어 역시 메모리에 존재하며 명령어를 반입할 때마다 메모리가 접근된다.

```
1  #include <unistd.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include "common.h"
5
6  int
7  main(int argc, char *argv[])
8  {
9      int *p = malloc(sizeof(int));                    // a1
10     assert(p != NULL);
11     printf("(%d) address pointed to by p: %p\n",
12            getpid(), p);                             // a2
13     *p = 0;                                          // a3
14     while (1) {
15         Spin(1);
16         *p = *p + 1;
17         printf("(%d) p: %d\n", getpid(), *p);        // a4
18     }
19     return 0;
20 }
```

위의 프로그램은 malloc()을 호출하여 메모리를 할당하는 프로그램으로 우선우선 (a1)메모리를 할당받는다. 그 후 (a2)할당받은 메모리의 주소를 출력한다. (a3)새로 할당받은 메모리의 첫 슬롯에 0을 넣는다. 반복문으로 진입하여 1초 대기 후, 변수 p가 가리키는 주소에 저장되어 있는 값을 1 증가시킨다. 그리고 출력할 때마다 실행 중인 프로그램 고유의 값인 PID 값을 함께 출력한다.

```
prompt> ./mem &; ./mem &
[1] 24113
[2] 24114
(24113) address pointed to by p: 0x200000
(24114) address pointed to by p: 0x200000
(24113) p: 1
(24114) p: 1
(24114) p: 2
(24113) p: 2
(24113) p: 3
(24114) p: 3
(24113) p: 4
(24114) p: 4
```

같은 프로그램을 여러 번 실행시켜 보면 위와 같다. 프로그램들은 00200000이라는 같은 주소에 메모리를 할당받지만, 각각이 독립적으로 00200000 번지의 값을 갱신한다. 각 프로그램은 물리 메모리를 다른 프로그램과 공유하는 것이 아니라 각자 자신의 메모리를 가지고 있는 것처럼 보인다.

운영체제가 메모리 가상화를 하기 때문에 위와 같은 현상이 생긴다. 각 프로세스는 자신만의 가상 주소 공간(virtual address space)을 갖는다. 운영체제는 이것을 컴퓨터의 물리 메모리로 매핑(mapping)한다.

2.3 병행성(Concurrency)

병행성이라는 용어는 프로그램이 한 번에 많은 일을 하려 할 때 발생하는, 반드시 해결해야 하는 문제들을 가리킬 때 사용한다.

병행성 문제는 운영체제 자체에서 발생하는데 운영체제만의 문제는 아니다. 멀티 쓰레드(multi thread) 프로그램도 동일한 문제를 드러낸다.

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include "common.h"
4
5   volatile int counter = 0;
6   int loops;
7
8   void *worker(void *arg) {
9       int i;
10      for (i = 0; i < loops; i++) {
11          counter++;
12      }
13      return NULL;
14  }
15
16  int
17  main(int argc, char *argv[])
18  {
19      if (argc != 2) {
20          fprintf(stderr, "usage: threads <value>\n");
21          exit(1);
22      }
23      loops = atoi(argv[1]);
24      pthread_t p1, p2;
25      printf("Initial value : %d\n", counter);
26
27      Pthread_create(&p1, NULL, worker, NULL);
28      Pthread_create(&p2, NULL, worker, NULL);
29      Pthread_join(p1, NULL);
30      Pthread_join(p2, NULL);
31      printf("Final value   : %d\n", counter);
32      return 0;
33  }
```

위의 프로그램은 Pthread_create()를 사용하여 두 개의 쓰레드를 생성한다. 각 쓰레드는 worker()라는 루틴을 실행하는데, 이 루틴은 loops번 만큼 루프를 반복하면서 카운터 값을 증가시킨다.

loops 변수를 1000으로 설정하여 프로그램을 실행시키면 counter의 최종값은 2000이 된다. 그러나 loops 값을 더 큰 값으로 지정하면 결과는 다음과 같다.

```
prompt> ./thread 100000
Initial value : 0
Final value   : 143012    // huh??
prompt> ./thread 100000
Initial value : 0
Final value   : 137298    // what the??
```

값을 100,000으로 주었더니 최종값이 200,000이 아닌 143,012가 되었다. 이러한 결과의 원인은 명령어가 한 번에 하나씩만 실행된다는 것과 관련 있다. 위의 프로그램에서 counter를 증가시키는 부분은 세 개의 명령어로 이루어진다. 이 세 개의 명령어가 한 번에 3개 모두 실행되지 않기 때문에 이상한 일이 발생할 수 있다.

2.4 영속성(Persistence)

DRAM과 같은 장치는 데이터를 휘발성(volatile) 방식으로 저장하기 때문에 데이터가 쉽게 손실될 수 있다. 전원이 꺼지거나 시스템이 고장나면 메모리의 모든 데이터는 사라진다. 따라서 데이터를 영속적으로 저장할 수 있는 하드웨어와 소프트웨어가 필요하다.

하드웨어는 입력/출력(input/output) 혹은 I/O 장치 형태로 제공된다. 장기간 보존할 정보를 저장하는 장치로는 일반적으로 하드 드라이브가 사용된다.

디스크를 관리하는 운영체제 소프트웨어를 파일 시스템(file system)이라고 부르는데, 파일 시스템은 사용자가 생성한 파일을 시스템의 디스크에 안전하고 효율적인 방식으로 저장할 책임이 있다.

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <assert.h>
4  #include <fcntl.h>
5  #include <sys/types.h>
6
7  int
8  main(int argc, char *argv[])
9  {
10     int fd = open("/tmp/file", O_WRONLY | O_CREAT | O_TRUNC, S_IRWXU);
11     assert(fd > -1);
12     int rc = write(fd, "hello world\n", 13);
13     assert(rc == 13);
14     close(fd);
15     return 0;
16 }
```

위의 코드는 문자열 “hello world”를 포함한 파일 /tmp/file을 생성하는 코드이다. 여기서 프로그램은 운영체제를 세 번 호출한다. 첫째, open()콜로 파일을 생성하고 연다. 둘째, write()콜로 파일에 데이터를 쓴다. 셋째, close()콜로 단순히 파일을 닫는데, 프로그램이 더 이상 해당 파일을 사용하지 않는다는 것을 나타낸다. 이들 시스템

콜은 운영체제에서 파일 시스템이라 불리는 부분으로 전달된다.
운영체제는 시스템 콜이라는, 표준화된 방법으로 장치들을 접근할 수 있게 한다. 운영체제는 표준 라이브러리(standard library)처럼 보이기도 한다.

2.5 설계 목표(Design Goals)
앞서 설명한 시스템을 구현하려면 몇 가지 목표를 세워야 한다. 이러한 목표는 설계와 구현에 집중하고, 필요한 경우 절충안을 찾는 데 필수적이다.
가장 기본적인 목표는 시스템을 편리하고 사용하기 쉽게 만드는 데 필요한 개념들을 정의하는 것이다. 추상화를 통해 큰 프로그램을 이해하기 쉬운 작은 부분들로 나누어 구현할 수 있다.
운영체제의 설계와 구현에 중요한 목표는 성능이다. 또 다른 목표는 응용 프로그램 간의 보호, 그리고 운영체제와 응용 프로그램 간의 보호이다. 여러 프로그램들이 동시에 실행되기 때문에, 운영체제는 한 프로그램의 행위가 다른 프로그램에게 피해를 주지 않는다는 것을 보장해야 한다. 운영체제는 계속 실행되어야 하며, 이러한 종속성 때문에 운영체제는 높은 수준의 신뢰성(reliability)을 제공해야 한다.

2.6 역사 약간(Some History)
초창기 운영체제 : 단순 라이브러리
- 기본적으로 자주 사용되는 함수를 모아 놓는 라이브러리에 불과했다.
작업들이 준비되면 컴퓨터 관리자가 일괄적으로 처리한다. 이러한 방식의 컴퓨팅을 일괄 처리(batch)라고 부른다.
라이브러리를 넘어서 : 보호
- 단순한 라이브러리를 넘어서 운영체제는 컴퓨터 관리 면에서 더 중심적인 역할을 하게 된다.
사용자 응용 프로그램은 사용자 모드(user mode)라고 불리는 상태에서 실행되는데, 여기에서는 응용 프로그램이 할 수 있는 일을 하드웨어적으로 제한한다. 시스템 콜은 trap이라 불리는 특별한 하드웨어 명령어를 이용하여 호출된다. 시스템 콜 시작 시, 하드웨어는 미리 지정된 트랩 핸들러 함수에게 제어권을 넘기고 특권 수준(privilege level)을 커널 모드(kernel mode)로 격상시킨다.
멀티프로그래밍 시대
- 컴퓨터 자원의 효율적 활용을 위해 멀티프로그래밍(multiprogramming) 기법이 대중으로 사용되었다. 운영체제는 여러 작업을 메모리에 탑재하고 작업들을 빠르게 번갈아 가며 실행하여 CPU 사용률을 향상시킨다.
현대
- 더 빠르고 대중적인 컴퓨터가 등장하였다. 현재 개인용 컴퓨터 또는 PC라고 불리는 컴퓨터이다. Apple사의 초기 컴퓨터와 IBM PC에 의해 주도된 컴퓨터는 책상마다 하나의 컴퓨터를 놓을 수 있을만큼 쌌기 때문에 컴퓨팅의 주도 세력이 되었다.

운영체제 수업을 들으면서 나의 목표

운영체제 과목은 작년 2학기 때 들었던 시스템 프로그래밍의 연장선이다. 시스템 프로그램은 시스템을 제어하는 프로그램으로, 시스템을 알아가는 과정이었다. 윈도우나 리눅스 같은 운영체제도 시스템 프로그램에 속해 있다. 따라서 작년에 큰 틀을 잡아놓고 이번에 좀 더 세세하게 배운다고 생각한다.

우선 가장 큰 목표는 작년처럼 좋은 학점을 받는 것이다. 어쩌면 너무 딱딱하고 특징 없는 목표일 수도 있겠지만 나는 학생이고 대부분의 학생들이 그렇듯이 좋은 학점을 받고 싶다.

가장 상위 목표야 위와 같은 것이고, 이상적으로는 리눅스에 대해 공부하는 것이다. 작년 수업 때 putty를 사용했었는데 C언어랑은 또 다른 재미가 있었다. 그래서 리눅스를 배워보고 싶어 겨울방학 때 스터디 신청도 했었다. 결과적으로는 중도하차 했지만 다시 리눅스를 만질 수 있게 되어 기쁘다.

이번 과제를 하면서도 본 노이만이라든지 가상화, load와 store 같은 단어를 보며 작년 한 학기동안 여러 과목에서 배운 내용들이 내 머릿속에 희미하게 남아있는 거 같아 신기했다. 병행성에 있는 멀티쓰레드 프로그래밍에서 pthread()를 보고 작년에 했던 팀 프로젝트 과제도 생각났다. 우리가 다루었던 주제가 프로그래밍 잘하는 사람 입장에서 보면 정말 아무것도 아닌 주제였을 수도 있지만 코드를 짜고, 성공했을 때 팀원과 그 누구보다 행복했었다. 어떻게 보면 이처럼 내가 학습한 내용을 가지고 도전해서 성공하는 것, 이러한 사소한 것 또한 목표가 될 수 있다고 생각한다.

이제 개강한 지 3주차인데 이 과제를 통해 방학이라 풀려있던 마음을 다잡을 수 있어서 좋았다. 기본 틀을 잡은 것을 바탕으로 열심히 공부해 더 효과적인 프로그램 개발을 할 수 있도록 해야겠다. 한 발 나아가서, 이제 3학년이기 때문에 진로에 대해서도 고찰해보는 시간을 가져야겠다.

cpu.c

```
os-lecture@os-lecture:~/OS_study/ostep-code/intro$ ./cpu "A"
A
A
A
A
A
A
A
^C
os-lecture@os-lecture:~/OS_study/ostep-code/intro$
os-lecture@os-lecture:~/OS_study/ostep-code/intro$
os-lecture@os-lecture:~/OS_study/ostep-code/intro$ ./cpu A & ./cpu B & ./cpu C & ./cpu D &
[1] 4474
[2] 4475
[3] 4476
[4] 4477
A
os-lecture@os-lecture:~/OS_study/ostep-code/intro$ B
C
D
A
B
C
D
A
B
C
D
A
B
C
D
A
B
C
```

mem.c

```
os-lecture@os-lecture:~/OS_study/ostep-code/intro$ ./mem
(4660) address pointed to by p: 0x2463010
(4660) p: 1
(4660) p: 2
(4660) p: 3
(4660) p: 4
(4660) p: 5
(4660) p: 6
^C
os-lecture@os-lecture:~/OS_study/ostep-code/intro$
os-lecture@os-lecture:~/OS_study/ostep-code/intro$
os-lecture@os-lecture:~/OS_study/ostep-code/intro$ ./mem & ./mem &
[1] 4663
[2] 4664
(4663) address pointed to by p: 0x1e1b010
os-lecture@os-lecture:~/OS_study/ostep-code/intro$ (4664) address pointed to by p: 0x1c03010
(4663) p: 1
(4664) p: 1
(4663) p: 2
(4664) p: 2
(4663) p: 3
(4664) p: 3
(4663) p: 4
(4664) p: 4
(4663) p: 5
(4664) p: 5
```

thread.c

```
os-lecture@os-lecture:~/OS_study/ostep-code/intro$ ./threads 1000
Initial value : 0
Final value   : 2000
os-lecture@os-lecture:~/OS_study/ostep-code/intro$
os-lecture@os-lecture:~/OS_study/ostep-code/intro$ ./threads 10000
Initial value : 0
Final value   : 20000
os-lecture@os-lecture:~/OS_study/ostep-code/intro$
os-lecture@os-lecture:~/OS_study/ostep-code/intro$ ./threads 100000
Initial value : 0
Final value   : 110113
os-lecture@os-lecture:~/OS_study/ostep-code/intro$
os-lecture@os-lecture:~/OS_study/ostep-code/intro$ ./threads 100000
Initial value : 0
Final value   : 171132
os-lecture@os-lecture:~/OS_study/ostep-code/intro$
```

io.c

```
os-lecture@os-lecture:~/OS_study/ostep-code/intro$ ls
Makefile   common.h          cpu    io    mem    threads
README.md  common_threads.h  cpu.c  io.c  mem.c  threads.c
os-lecture@os-lecture:~/OS_study/ostep-code/intro$
os-lecture@os-lecture:~/OS_study/ostep-code/intro$ ./io
os-lecture@os-lecture:~/OS_study/ostep-code/intro$
os-lecture@os-lecture:~/OS_study/ostep-code/intro$ ls
Makefile   common.h          cpu    io    mem    threads    tmp.txt
README.md  common_threads.h  cpu.c  io.c  mem.c  threads.c
os-lecture@os-lecture:~/OS_study/ostep-code/intro$
os-lecture@os-lecture:~/OS_study/ostep-code/intro$ cat tmp.txt
hello world
os-lecture@os-lecture:~/OS_study/ostep-code/intro$
```

고찰

이번 예제는 코드도 ppt나 github에 다 있고 그대로 가져와서 쓰면 되는건데도 불구하고 상당히 많은 어려움을 겪었다. virtual box를 이용해 리눅스를 쓰는 것이 작년에 쓰던 putty와는 완전히 달랐다.

처음 접해서인지 프로그램을 사용하는 것이 너무나 익숙하지 않았다. 그래도 이번 예제를 다 실행시켜 보면서 많은 공부가 되었다. putty의 가장 큰 단점으로 코드를 전부 쓰는 것이 있었는데 make 명령어 하나로 깔끔하게 처리되는 것이 너무 좋았다.

ppt에 있는 mem.c를 백그라운드에서 여러개 실행하면 같은 주소가 뜨는데 내가 프로그램을 돌리면 왜 계속 다른 주소가 뜨는 것이 의문이다.