#1

2. Introduction to Operating Systems

프로그램을 쉽게 실행(동시에 많은 프로그램을 동시에 실행하는 것처럼 보이는 것)하고, 프로그램이 메모리를 공유하고, 프로그램이 장치와 상호작용할 수 있게 하는 소프트웨어가 있다. 그러한 소프트웨어는 시스템을 사용하기 쉬운 방식으로 정확하고 효율적으로 작동하게 하는데 이런 소프트웨어를 OS(Operating System ; 운영 체제)라고 한다. OS를 수행하는 기본적인 방법으로는 가상화라고 하는 기술이 있다. 즉, OS는 물리적 리소스(예 : 프로세서, 메모리 또는 디스크)를 가져와 보다 일반적이고 강력하며 사용하기 쉬운 가상 형식으로 변환한다. 따라서 때때로 운영 체제를 가상 머신이라고도 한다. 물론 사용자가 OS에게 무엇을 해야하는지 알려주어야 하고 이로서 OS는 프로그램 실행, 메모리 할당 또는 파일 액세스와 같은 가상 머신의 기능을 활용할 수 있도록 OS는 일부 인터페이스(호출 할 수 있는 API)를 제공한다. 실제로 일반적인 OS는 응용 프로그램에서 사용할 수 있는 수백 개의 시스템 호출을 내보낸다. OS는 프로그램 실행, 메모리 및 장치 액세스 및 기타 관련 작업에 대한 이러한 호출을 제공하기 때문에 OS가 응용 프로그램에 표준 라이브러리를 제공한다고 말한다. 마지막으로 가상화를 통해 많은 프로그램을 실행(CPU 공유)하고 많은 프로그램이 자체 명령 및 데이터에 동시에 액세스(메모리 공유)하고 많은 프로그램이 장치에 액세스(디스크 공유) 하므로 OS 자원 관리자라고도 한다. 각 CPU, 메모리 및 디스크는 시스템의 자원이다. 따라서 다른 많은 가능한 목표를 염두해 두고 효율적이고 공정하게 리소스를 관리하는 것이 운영 체제의 역할이다.

2.1 Virtualizing The CPU

시스템에 매우 많은 가상 CPU가 환상을 보여준다. 단일 CPU (또는 작은 세트)를 겉보기에 무한한 수의 CPU로 바꾸어 많은 프로그램이 한 번에 실행되는 것처럼 보이게 하는 것이 CPU 가상화라는 것이다. 이 책의 첫 번째 주요 부분의 초점입니다. 물론, 프로그램을 실행하고 중지하고 운영 체제에 실행할 프로그램을 알리려면 원하는 인터페이스를 OS에 전달하는데 사용할 수 있는 인터페이스(API)가 필요하다. 실제로 대부분의 사용자가 운영 체제와 상호 작용하는 주요 방법이다. 또 한 번에 여러 프로그램을 실행할 수 있으면 모든 종류의 새로운 질문이 생길 수 있다. 예를 들어, 특정 시간에 두 프로그램을 동시에 실행하려면 어떤 프로그램을 실행해야 할까? 이 질문은 OS의 정책에 의해 답변된다. 정책은 OS 내의 여러 위치에서 이러한 유형의 질문에 답변하는 데 사용되므로 운영 체제가 구현하는 기본 메커니즘(예 : 여러 프로그램을 한 번에 실행할 수 있는 기능)에 대해 배우면서 정책을 연구한다.

2.2 Virtualizing Memory

현대 기계가 제시하는 물리적 메모리 모델은 매우 간단하다. 메모리는 단지 바이트 배열이다. 메모리를 읽으려면 저장된 데이터에 액세스 할 수 있는 주소를 지정해야 한다. 메모리를 쓰거나 업데이트하려면 주어진 주소에 쓸 데이터를 지정해야 한다. 프로그램이 실행될 때 항상 메모리에 액세스한다. 프로그램은 모든 데이터 구조를 메모리에 유지하고 로드 및 저장과 같은

다양한 명령 또는 작업을 수행 할 때 메모리에 액세스하는 기타 명시적 명령을 통해 액세스한다. 따라서 메모리는 각 명령 패치에서 액세스된다.

```
prompt> ./mem
(2134) address pointed to by p: 0x200000
(2134) p: 1
(2134) p: 2
(2134) p: 3
(2134) p: 4
(2134) p: 5
^C
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
```

이 프로그램을 몇 가지 기능을 한다. 먼저 일부 메모리를 할당한다. 그런 다음 메모리 주소를 출력한 다음 새로 할당된 메모리의 첫 번째 슬롯에 숫자 0을 넣는다. 마지막으로 1초 동안 지연되고 p에 보유된 주소에 저장된 값이 증가한다. 모든 print문과 함께 실행중인 프로그램의 프로세스 식별자(pid)를 출력한다. 이 pid는 실행중인 프로세스마다 고유하다. 새로 할당된 메모리는 주소 0x200000에 있다. 프로그램이 실행되면 천천히 값을 업데이트하고 결과를 출력한다. 이제 동일한 프로그램의 여러 인스턴스를 다시 실행하여 어떻게 되는지 확인해보자. 예제에서 실행중인 각 프로그램이 동일한 주소(0x200000)에 메모리를 할당했지만 각각 0x200000의 값을 독립적으로 업데이트하는 것으로 보인다. 마치 실행중인 각 프로그램에 다른 실제 프로그램과 동일한 실제 메모리를 공유하는 대신 자체 전용 메모리가 있는 것처럼 보인다. 실제로, OS가 메모리를 가상화한다는 것이 여기서 보여진다. 각 프로세스는 자체 가상 주소 공간에 액세스하며, OS는 어떻게든 기계의 물리적 메모리에 매핑된다. 하나의 실행중인 프로그램 내의 메로리 참조는 다른 프로세스(또는 OS 자체)의 주소 공간에 영향을 미치지 않는다. 실행중인 프로그램에 관한 한 그것은 실제 메모리를 가지고 있다. 그러나 실제 메모리는 운영 체제에서 관리하는 공유 리소스이다. 이 모든 것이 정확히 어떻게 이루어지는가는 이 주제의 첫 번째 주제인 가상화에 관한 주제이기도 하다.

2.3 Concurrency

이 책의 또 다른 주요 주제는 동시성이다. 우리는 이 개념적인 용어를 사용하여 동일한 프로그램에서 동시에 많은 일을 할 때 발생하는 여러 가지 문제를 언급할 것이다. 동시성 문제는 운영 체제 자체에서 먼저 발생했다. 위의 가상화에 대한 예에서 볼 수 있듯이 OS는 한 번에 많은 프로세스를 저글링하고 먼저 하나의 프로세스를 실행한 다음 다른 프로세스를 실행한다. 동시성 문제는 물론 OS 자체에만 국한되지 않는다. 현대의 멀티 스레드 프로그램은 동일한 문제를 보인다. 다중 스레드 프로그램의 예를 보자. 메인 프로그램은 Pthread create()를 사용하여 두 개의 스레드를 만든다. 스레드는 다른 함수와 동일한 메모리 공간 내에서 실행되는 함수로 생각할 수 있다. 한 번에 둘 이상의 함수가 활성화되어 있다. 이 예제에서 각 스레드는 worker()라는 루틴에서 실행을 시작한다. 이 루틴은 for루프 횟수만큼 카운터를 증가시킨다.

```c
#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;
}
```

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

변수 루프의 입력 값이 1,000으로 설정된 상태에서 이 프로그램을 실행하면 for루프를 1000번 돌기 때문에 변수의 마지막 값은 2,000이 된다. 즉 입력 값이 N이라면 최종 출력은 2N이 된다는 것을 확인 할 수 있다. 하지만 여기서 문제가 발생한다.

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

입력 값을 100,000으로 했을 때 최종 값이 200,000이 아닌 143,012가 출력된다. 다시 실행해 보았을 때는 값이 잘못된 것 뿐 만이 아니라 다른 결과 값이 나온다. 이는 높은 루프 값으로 프로그램을 반복해서 실행하면 때로는 올바른 값을 얻지 못한다는 것을 보여준다. 이러한 결과는 이유는 명령이 실행되는 방식과 관련이 있다. 공유 카운터가 증가하는 위는 프로그램의 핵심 부분은 세 가지 명령을 취한다. 하나는 카운터에서 메모리의 값을 레지스터로 로드하고, 하나는 증가시키고 다른 하나는 메모리에 다시 저장한다. 이 세 가지 명령어는 한 번에 실행되지 않기 때문이다.

2.4 Persistence

이 과정의 세 번째 주요 주제는 지속성이다. 시스템 메모리에서 DRAM과 같은 장치는 값을 휘발성으로 저장하기 때문에 데이터가 쉽게 손실될 수 있다. 전원이 꺼지거나 시스템이 충돌하면 메모리의 모든 데이터가 손실된다. 따라서 데이터를 지속적으로 저장할 수 있으려면 하드웨어와 소프트웨어가 필요하다. 따라서 이러한 스토리는 사용자가 데이터에 대해 많은 관심을 기울이는 모든 시스템에 중요하다. 하드웨어는 일종의 입/출력 또는 I/O 장치 형태로 제공

된다. 최신 시스템에서 하드 드라이브는 수명이 긴 정보를 위한 공통 저장소이지만 SSD도 이 분야에서 앞서 나가고 있다. 일반적으로 디스크를 관리하는 운영 체제의 소프트웨어를 파일 시스템이라고 한다. 따라서 사용자가 생성한 모든 파일을 시스템의 디스크에 안정적이고 효율적인 방식으로 저장해야 한다. OS에서 CPU 및 메모리에 제공한 추상화와 달리 OS는 각 응용 프로그램에 대해 개인 가상화 디스크를 만들지 않는다. 오히려 종종 사용자는 파일에 있는 정보를 공유하기를 원한다고 가정한다. 예를 들어 C프로그램을 작성할 때 먼저 편집기를 사용하여 C파일을 작성하고 편집할 수 있다. 완료되면 컴파일러를 사용하여 소스 코드를 실행 파일로 변환할 수 있다. 완료되면 새 실행 파일을 실행할 수 있다. 따라서 다른 프로세스에서 파일을 공유하는 방법을 볼 수 있다.먼저 편집기는 컴파일러의 입력 역할을 하는 파일을 만든다. 컴파일러는 해당 입력 파일을 사용하여 새 실행 파일을 만든다.

```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;
}
```

이를 더 잘 이해하기 위해 코드를 살펴보자. 이 코드는 “hello world”라는 문자열을 포함하는 파일을 작성하는 코드이다. 이 작업을 수행하기 위해 프로그램은 운영 체제를 세 번 호출한다. 첫 번째는 open()을 호출하여 파일을 열고 만든다. 두 번째는 write()를 사용하여 일부 데이터를 파일에 쓴다. 세 번째로 close()는 단순히 파일을 닫아 프로그램이 더 이상 데이터를 쓰지 않음을 나타낸다. 이러한 시스템 호출은 파일 시스템이라고 하는 운영 체제의 일부로 라우팅된다. 그러면 파일 시스템이 요청을 처리하고 일종의 오류 코드를 사용자에게 반환한다. 실제로 디스크에 쓰기 위해 OS가 무엇을 하는지 궁금할 것이다. 파일 시스템은 상당한 양의 작업을 수행해야 한다. 먼저 디스크에서 이 새 데이터가 상주 할 위치를 파악한 다음 파일 시스템이 유지 관리하는 다양한 구조에서 데이터를 추적한다. 그렇게 기본 스토리지를 읽거나 업데이트하기 위해 기본 스토리지 디바이스에 I/O요청을 발행해야 한다. 장치 드라이버에 무언가를 하도록 요청하는 것은 복잡하고 디테일이 요구되지만 OS는 시스템 호출을 통해 장치에 액세스하는 표준적이고 간단한 방법을 제공한다. 따라서 OS는 때때로 표준 라이브러리로 간주된다. 물로 장치에 액세스하는 방법과 파일 시스템이 해당 장치 위에 데이터를 지속적으로 관리하는 방법에 대한 자세한 내용이 있다. 성능상의 이유로 대부분의 파일 시스템은 이러한 쓰기를 한동안 지연시켜 더 큰 그룹으로 일괄 처리하려고 한다. 쓰기 중 시스템 충돌 문제를 처리하기 위해 대부분의 파일 시스템은 저널링 또는 COW(Copy-On-Write)와 같은 복잡한 쓰기 프로토콜을 통합하여 쓰기 순서 중에 오류가 발생하면 시스템이 디스크에 쓰기 순서를 신중하게 지정하여 나중에 합리적인 상태로 복구할 수 있다. 서로 다른 공통 작업을 효율적으로 만들기 위해 파일 시스템은 간단한 목록에서 복잡한 btree에 이르는 다양한 데이터 구조와 액세스 방법을 사용한다.

#2

```
sys163006@embedded:~$ ls
addprt.c  back.c  examples.desktop  p13  p13.c  p16  p16.c  p2  p20.c  p24  p24.c  p2.c
sys163006@embedded:~$ whoami
sys163006
sys163006@embedded:~$ date
2020. 03. 27. (금 ) 19:47:21 KST
sys163006@embedded:~$ vi p13.c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <assert.h>

int main(int argc, char *argv[])
{
        if (argc != 2) {
                fprintf(stderr, "usage: cpu <string>\n");
                exit(1);
        }
        char *str = argv[1];
        while (1) {
                wait(100000);
                printf("%s\n", str);
        }
        return 0;
}
```

```
sys163006@embedded:~$
sys163006@embedded:~$ clear
sys163006@embedded:~$ ls
addprt.c  back.c  examples.desktop  p13  p13.c  p16  p16.c  p2  p20.c  p24  p24.c  p2.c
sys163006@embedded:~$ whoami
sys163006
sys163006@embedded:~$ date
2020. 03. 27. (금 ) 19:48:36 KST
sys163006@embedded:~$ vi p16.c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
        int *p = malloc(sizeof(int)); // a1
        if(p != NULL){
                 exit(1);
        }
        printf("(%d) address pointed to by p: %p\n",
        getpid(), p); // a2
        *p = 0; // a3
        while (1) {
                wait(1);
                *p = *p + 1;
                printf("(%d) p: %d\n", getpid(), *p); // a4
        }
        return 0;
}
```

```
sysl63006@embedded:~$ ls
addprt.c  examples.desktop  p13.c  p16.c  p20.c  p24.c
back.c    p13               p16    p2     p24    p2.c
sysl63006@embedded:~$ whoami
sysl63006
sysl63006@embedded:~$ date
2020. 03. 27. (금 ) 19:54:38 KST
sysl63006@embedded:~$ vi p20.c
```

```c
#include <stdio.h>
#include <stdlib.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]);
        int p1, p2;
        printf("Initial value : %d\n", counter);

        pthread_create(&p1, NULL, worker, (void*)NULL);
        pthread_create(&p2, NULL, worker, (void*)NULL);
        pthread_join(p1, (void*)NULL);
        pthread_join(p2, (void*)NULL);
        printf("Final value : %d\n", counter);
        return 0;
}
```

```
sysl63006@embedded:~$ ls
addprt.c  back.c  examples.desktop  p13  p13.c  p16  p16.c  p2  p20.c  p24  p24.c  p2.c
sysl63006@embedded:~$ whoami
sysl63006
sysl63006@embedded:~$ date
2020. 03. 27. (금 ) 19:49:50 KST
sysl63006@embedded:~$ vi p24.c
```

```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;
}
```