운영체제 COSE341(02) 1차과제

주제 : 시스템 콜 추가 및 이해

학과 : 정보대학 컴퓨터학과

학번 : 2017320229

이름 : 심규현

제출 날짜 : 2019.04.19

Freeday 사용 일수 : 0일

환경 : Microsoft Azure Ubuntu 18.04.02 (64bit), Linux kernel 4.20.11 (가상 환경)

목차

1. 리눅스 시스템 콜

가. 시스템 콜(system call)과 트랩(trap)

시스템 콜은 CPU가 프로그램을 실행하면서 발생하는 동기적인 이벤트의 일종이다. 이러한 시스템 콜은 사용자 모드(User mode)에서 작동하는 프로세스가 커널 모드(Kernel)에서 지원하는 서비스를 이용할 때, 커널모드로 실행 모드를 전환하도록 해준다. [그림 1]은 시스템콜이 발생할 때 CPU가 어떻게 이벤트를 처리하는지 보여준다. 동기적인 이벤트인 시스템 콜을 처리하기 위해 트랩을 발생시키는데, 해당 시스템 콜은 커널 공간(Kernel space)에서 트랩 서비스 루틴을 실행시킨 후, 다시 실행 중인 프로세스로 돌아오게 된다.

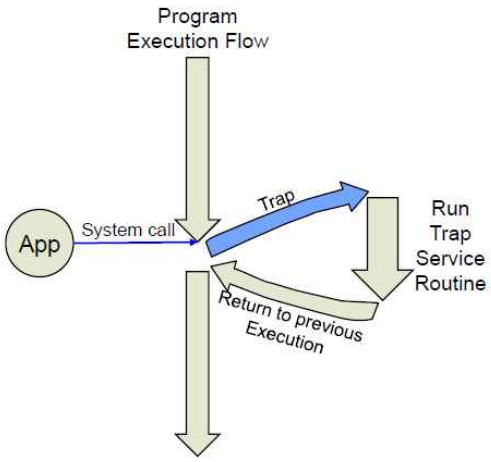
나. 리눅스 시스템 콜 호출 루틴

1) 프로세스에서 시스템 콜 호출 및 소프트웨어 인터럽트(트랩) 발생

[그림 2]는 실행 중인 프로세스에서 커널의 서비스를 이용하기 위해 임의의 시스템 콜인 System_call()을 호출했을 때, 사용자 모드에서의 실행 흐름을 보여준다. [그림 2]의 화살표는 CPU의 실행 흐름을 나타낸다. CPU가 실행 중인 프로세스 User process에서 System_call()을 호출하게 되면, libc.a[1]와 같은 라이브러리에서 System_call()을 찾아 실행한다. 라이브러리에서 호출된 System_call()은 해당 시스템 콜 실행을 위해 필요한 파라미터(e.g., 시스템 콜 고유번호)를 레지스터에 move 명령어를 통해 저장한다. 그리고 리눅스에서의 소프트웨어 인트럽트(SWI: software interrupt)의 instruction인 0x80 트랩 instruction을 발생시킨다.

2) 커널 공간에서의 소프트웨어 인터럽트 처리

앞의 1)에서 프로세스에서 시스템 콜을 호출하고 라이브러리에서 해당 함수를 찾아 호출한 뒤 커널로 0x80 소프트웨어 인터럽트를 발생시켰다. 해당 트랩을 커널 공간에서 처리해야 한다. [그림 3]은 커널 공간에서 트랩을 처리하는 과정을 보여준다. 일단 사용자 공간에서 커널 공간으로 트랩을 발생하면 가장 먼저 IDT(Interrupt Descriptor Table)[2]를 확인하여 해당 트랩을 처리하기 위한 핸들러의 주소를 찾는다. 위의 경우, 0x80 트랩을 처리하기 위해 system-call()의 시작주소를 찾아 실행한다.

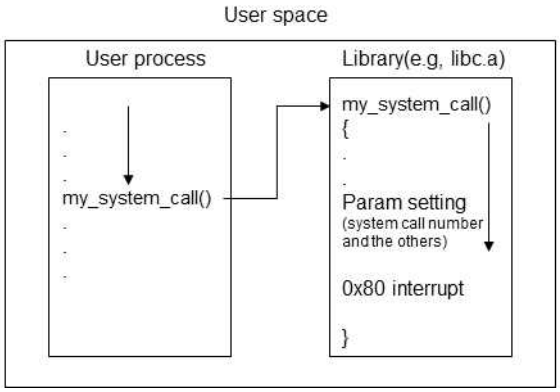


< Flow of handling trap>

[그림 1] 시스템 콜 도식

---

1) C언어에서 시스템콜을 호출했을 때 참조하는 라이브러리이다.
2) IDT(Interrupt Descriptor Table)는 인터럽트의 고유번호와 각 인터럽트를 처리하기 위한 핸들러를 대응하는 테이블을 말한다.



[그림 2] 시스템 콜 호출시 사용자 모드에서의 실행 흐름

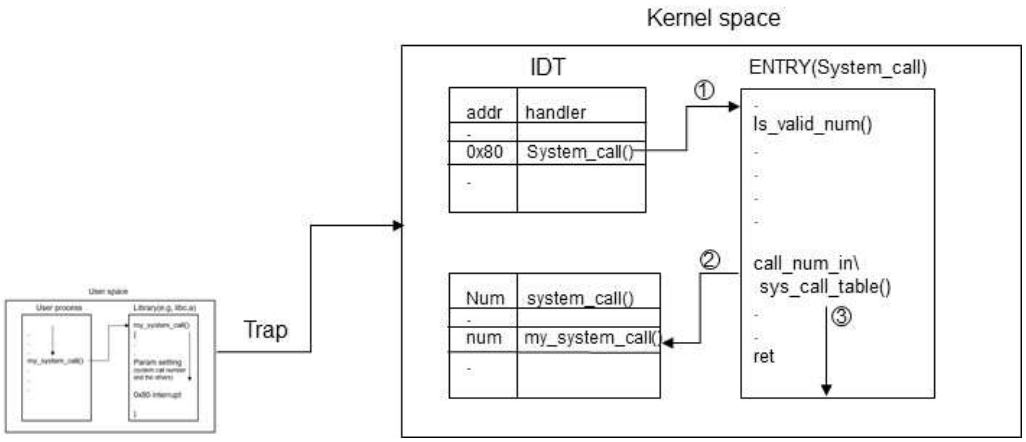
system_call()은 함수 이름 그대로, 시스템 콜 번호를 받아 그 번호에 해당하는 시스템 콜을 호출하는 함수이다. 구체적으로, 시스템 콜의 번호가 유효한지 확인하고 sys_call_Table(시스템 콜 테이블)[3]에서 해당 번호에 대응되는 시스템 콜(my_system_call)을 호출한다. 그리고 시스템 콜은 임의의 값을 반환하는데, 이는 시스템 콜 함수 실행이 종료됐음을 의미한다. 그리고 트랩 처리 루틴이 종료되며, 다시 사용자 모드로 돌아가게 된다.

2. 수정 및 작성한 부분과 설명

앞에서 설명한 시스템 콜을 우리가 sys_call_table에 추가하고 어플리케이션에서 직접 실행하는 것 또한 가능하다. 이 섹션에서는 시스템 콜을 추가하는 데 필요한 과정과 사용자 모드에서 시스템 콜을 호출하기 위한 코드를 설명한다.

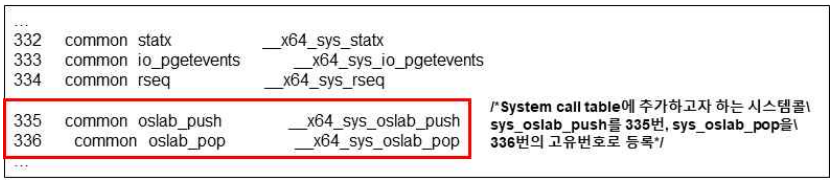
가. 시스템 콜 테이블 수정(./arch/x86/entry/syscalls/syscall_64.tbl)

앞의 시스템 콜 호출 루틴에서 보았듯이, system_call()의 루틴에서 sys_call_table을 참고하여 우리가 원하는 시스템 콜을 호출한다. 따라서 커널에 시스템 콜을 추가하기 위해서는 이 테이블을 수정해야 한다. [그림 4]는 테이블을 수정한 모습을 나타낸다. 추가하고자 하는 sys_oslab_push라는 시스템 콜에 번호 335를, sys_oslab_pop이라는 시스템 콜에 번호 336을 할당했다. 우리가 어플리케이



[그림 3] 커널 공간에서의 트랩 처리 실행 흐름

3) 리눅스의 경우 시스템 콜 테이블은(sys_call_table)은 리눅스에서 제공하는 모든 시스템 콜의 고유번호를 저장하고, 시스템 콜 함수의 주소들을 저장하는 테이블이다.



[그림 4] sys_call_table을 수정한 모습

션에서 이 번호를 통해 시스템 콜을 호출할 수 있다.

나. 시스템 콜 헤더파일 수정(./include/linux/syscalls.h)

시스템 콜을 코딩한 파일에는 정확한 함수의 정의가 들어가지 않는다. 따라서 다른 시스템 콜에서 해당 시스템 콜을 참조하게 해주기 위해 헤더 파일에 정확한 정의를 넣어줘야 할 필요가 있다. 함수의 정의는 C언어에서의 함수 정의와 같은 형태이다. 또한, asmlinkage를 붙여준 이유는 우리가 만든 시스템 콜은 어셈블리 코드에서 호출되는데, 어셈블리 코드에서 C함수 호출이 가능하게 해주기 위해서이다. sys_call_table에 추가한 것과 이름이 같은 함수를 헤더 파일에 추가하였다.

다. 시스템 콜 코드 추가 (./kernel/my_stack_syscall.c)

우리가 추가한 코드가 시스템 콜의 역할을 하게 하기 위해서는 [그림 6]과 같이 몇 개의 헤더 파일을 추가해 주어야 한다. 그리고 stack으로 활용할 배열과 stack의 top을 가리키는 instructor 변수를 추가하였다.

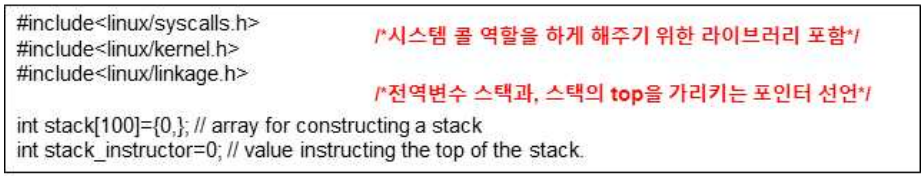
1) sys_oslab_push

[그림 7]은 sys_oslab_push 함수에 대한 코드를 나타낸다. 함수를 header 부분에서 SYSCALL_DEFINE1을 통해 함수를 정의하는 것을 확인할 수 있다. 이것은 syscalls.h에서 제공하는 시스템 콜을 구현하기 위한 매크로이다. 끝에 붙은 숫자는 파라미터의 개수를 의미하며, 함수의 이름을 적은 뒤 파라미터 형과 파라미터 이름을 ,(콤마)로 구분하여 적는 것이 특징이다. **그리고 스택을 순회하여 입력과 중복되는 값이 스택에 있다면 return하는 것으로 중복처리 하였다.** 그리고 반복문의 헤더가 아니라 oslab_push의 바디에서 반복문 변수를 선언한 것을 볼 수 있는데, 반복문의 헤더에서 선언하면 에러가 발생하기 때문이다. 이 내용은 섹션 4에 포함했다.

2) sys_oslab_pop

[그림 8]은 sys_oslab_pop 함수에 대한 코드를 나타낸다. sys_oslab_push에서 설명한 부분은 생략하였다. 일반적인 스택 pop 연산과 크게 다른 것은 없으며, 스택의 top에 저장된 요소를 반환하고 stack_instructor을 감소하여 스택의 논리적인 크기를 줄인다. 그리고 관습대로 함수가 pop할 요소가 없다면(실패 상황) 음수를 반환하고 시스템 콜 로그에 에러를 출력한다.

[그림 5] 함수의 프로토타입을 syscalls.h에 정의한 모습



[그림 6] 시스템 콜 소스 파일이 참고하는 라이브러리와 전역변수

```
SYSCALL_DEFINE1(oslab_push, int, a){ /*파라미터가 1개인 시스템 콜 구현을 위한 매크로 (함수이름, 파라미터 자료형, 파라미터 이름) */
    /* push param a in the stack(global variable)
       param a: a numeric value which will be pushed
    */
    int i = 0; // variable 1 for iterating
    int j = 0; // variable 2 for iterating        /*반복문을 위한 변수 선언*/

    // check a input is already in a stack. If it is true, return.
    for(i=0; i<stack_instructor; i++){                    /*스택에 파라미터 a와 같은 값을 가진 요소가 있는지 확인하기 위한 반복문*/
        if(stack[i]==a){
            printk("[System call] oslab_push(): %d is already in stack\n",a);
            return;
        }
    }

    if(i==stack_instructor){          /* if, 같은 값이 있으면 밖으로 분기*/
        stack[stack_instructor++]=a;
        printk("[System call] oslab_push(): Push %d\n", a);
        printk("Stack Top ----------\n");
        for(j=0; j<stack_instructor; j++){                /* printk를 통해 시스템 콜 로그에 stack내용을 출력하는 반복문*/
            printk("%d\n", stack[stack_instructor-1-j]); // print the element in stack to system-call log.
        }
        printk("Stack Bottom -------\n");
    }

}
```

[그림 7] 커널에 구현된 스택 연산 push를 실행하는 코드

```
SYSCALL_DEFINE0(oslab_pop){
    /* pop the element which is located in the top of stack.        /*stack을 순회하며 printk를 통해 시스템 콜 로그에 출력*/
       return : the top element of the stack if it exists.
    */

    int poped_value; // return value(the top element of the stack)
    int j=0; // variable 1 for iterating
                                                                    /*스택의 탑에 있는 반환할 정수를 poped_value에 대입*/
    if(stack_instructor>0){   // if the stack has at least one elements, then
        poped_value=stack[(stack_instructor--)-1];   // a variable storing the top element of a stack
        printk("[System call] oslab_pop(): Pop %d\n", poped_value);
        printk("Stack Top ----------\n");                        /*stack을 순회하며 printk를 통해 시스템 콜 로그에 출력*/
        for(j=0; j<stack_instructor; j++){
            printk("%d\n", stack[stack_instructor-1-j]);  // print reversely all elements to system-call log
        }
        printk("Stack Bottom -------\n");

        return poped_value;
    }
    else{  // if there is no element in the stack , then return the negative number.
        printk("[System_call] oslab_pop(): no element in stack\n");   /*stack에 아무요소가 없다면 음수 값을 반환하고 에러를 \
        return -2;                                                    시스템 콜 로그에 출력한다.*/
    }
}
```

[그림 8] 커널에 구현된 스택 연산 pop을 실행하는 코드

라. 커널 컴파일 시 추가한 시스템 콜을 컴파일하도록 Makefile 수정(./kernel/Makefile)

우리는 시스템 콜에 대한 정보를 테이블에 넣어주고, 직접 시스템 콜을 코딩하였다. 이제 남은 일은 커널을 컴파일할 때 시스템 콜을 같이 컴파일해주도록 Makefile을 수정하는 것이다. [그림 9]와 같이 컴파일 시 컴파일할 오브젝트 파일의 이름을 넣어주는 것으로 쉽게 추가할 수 있다. 이제 컴파일을 한다면 my_stack_syscall.c 파일도 함께 컴파일될 것이다.

마. 추가한 시스템 콜 실행(/my_app.c)

[그림 10]은 사용자 모드에서 앞에서 구현한 시스템 콜을 호출하는 프로그램의 코드이다. unistd.h의 syscall 함수를 통해 시스템 콜을 호출할 수 있는데, 입력으로 앞에서 정의한 시스템 콜의 고유번호를 넣어주어야 한다. 고유번호보다 시스템 콜 식별을 하기 쉽도록 #define을 통해 시스템 콜의 고유번호를 시스템 콜 함수의 이름으로 매핑하는 매크로를 정의한다. 그리고 syscall(function_name, parameters) 형태로 우리가 만든 시스템 콜을 호출할 수 있다. 여기서는 syscall(my_stack_push, input)을 통해 os_push 시스템 콜을 호출하고, syscall(my_stack_pop)을

```
obj-y    = fork.o exec_domain.o panic.o \
                ...
                async.o range.o smpboot.o ucount.o my_stack_syscall.o
```

[그림 9] Makefile 파일을 수정하여 컴파일할 파일 추가

```c
#include<unistd.h>
#define my_stack_push 335
#define my_stack_pop 336 /*시스템 콜의 이름을 고유번호로 매핑하는 매크로 선언*/

int main(){
        int i; // a variable for iterating
        int r; // a variable storing popped value

        // push three values by calling the added system_call oslab_push
        for(i=0 ; i<3; i++){
                syscall(my_stack_push, i+1);   /*oslab_push 시스템 콜 호출*/
                printf("Push: %d\n", i+1);
        }
        // pop three values by calling the added system_call oslab_pop.
        for(i=0; i<3; i++){
                r = syscall(my_stack_pop);  /*oslab_pop 시스템 콜 호출*/
                if(r>0){
                        printf("Pop: %d\n", r);
                }
                else{
                        printf("Pop: Error");
                }
        }
        return 0;
}
```

[그림 10] 시스템 콜을 호출하는 사용자 모드의 어플리케이션

통해 os_pop 시스템 콜을 호출한다. 이 프로그램은 1, 2, 3 스택에 push하고 pop을 세 번 수행하는 코드이다.

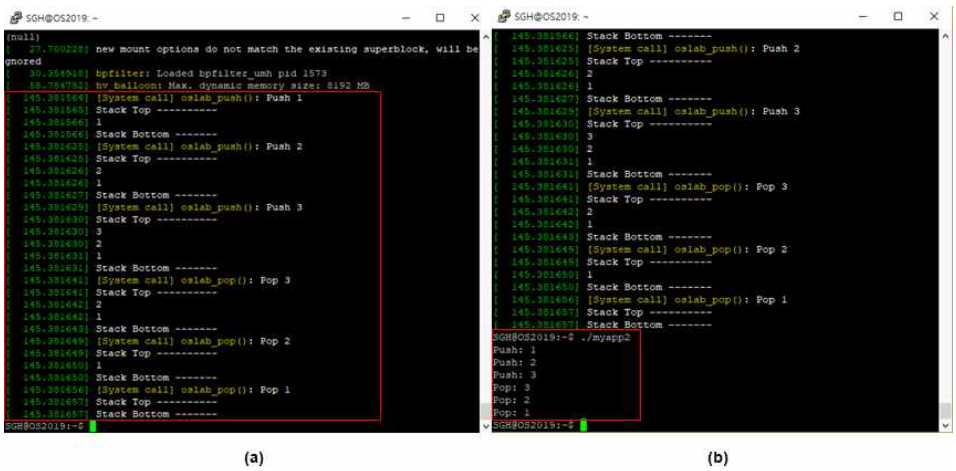
3. 실행 결과

[그림 11]은 사용자 어플리케이션을 실행시켰을 때 (a) 시스템 콜 로그, (b) 어플리케이션 출력을 나타낸다. 출력형식은 과제 개요와 같다.

4. 과제 수행 과정 중 발생한 문제점과 해결방법

과제를 수행하는 데 큰 어려움은 없었지만, 기존의 C언어 코딩 컨벤션을 이용했을 때 제대로 컴파일이 되지 않는 것을 확인할 수 있었다. [그림 12]는 for문의 헤더에서 변수선언을 했을 때 발생하는 에러를 보여준다. 문제점은 우리가 평소에 쓰는 C언어와 다른 버전의 컴파일러를 쓴다는 것이다. 그래서 for문 밖에서 변수를 선언하면 정상적으로 작동한다.

그리고 oslab_push 함수의 프로토타입을 선언할 때 반환형을 void로 하고 정수를 반환했더니 오류가 발생하지 않았다. 이것은 후에 에러를 발생시킬 수 있으므로 잠재적인 문제라고 판단하였고, return value를 제외하고 return을 실행하였다.



[그림 11] (a) 시스템 로그, (b) 사용자 어플리케이션 출력을 나타낸다.

```
kernel/my_stack_syscall.c:13:2: error: 'for' loop initial declarations are only allowed in C99 or C11 mode
  for(int i=0; i<stack_instructor; i++){
  ^~~
kernel/my_stack_syscall.c:13:2: note: use option -std=c99, -std=gnu99, -std=c11 or -std=gnu11 to compile your code
```

[그림 12] 반복문에서 변수선언 관련 에러