

[운영체제 1차 과제]

시스템 콜 추가 및 이해

이름: 김보민

학과: 산업경영공학부

학번: 2021170810

제출일자: 2023년 4월 10일

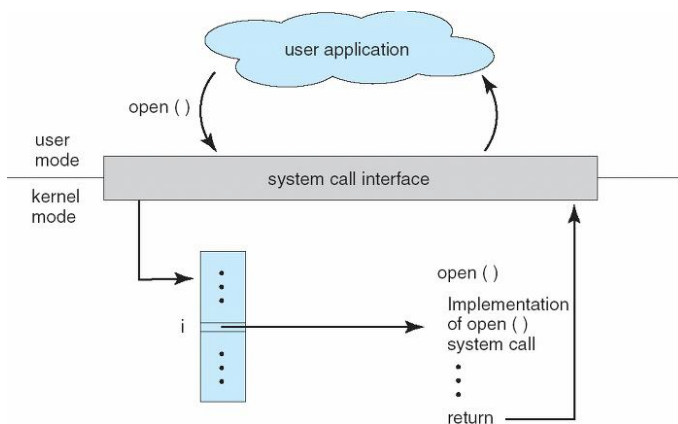
free day: 0일 사용

1. 리눅스의 System call

1) System call이란?

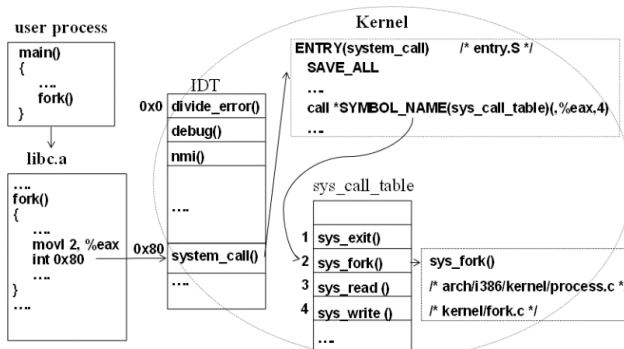
시스템콜은 운영체제가 제공하는 서비스를 user가 이용할 수 있도록 하는 방법이다. 운영체제는 kernel, GUI, library로 구성된다. 여기서 kernel은 운영체제를 구성하는 핵심 프로그램의 집합을 말한다. 운영체제는 자원 할당, 오류 탐지, 입출력 연산, 보안 기능 등 다양한 서비스를 사용자와 사용자 프로그램에게 제공한다. 그런데 사용자는 kernel에 직접적으로 접근이 불가능하기 때문에, system call의 도움을 받아야 하는 것이다.

2) System call의 구조



운영체제는 두 가지 모드, 커널 모드(Kernel Mode)와 사용자 모드(User Mode)로 구동된다. 커널 모드는 모든 시스템 메모리에 접근하고 모든 CPU 명령을 실행할 수 있다. 반면 사용자 모드는 사용자 애플리케이션을 실행하여 하드웨어에 직접 접근할 수 없다. 시스템 콜을 호출하면, 일시적으로 커널 모드로 전환되어 커널 영역의 기능을 사용자 모드에서 접근할 수 있게 해준다. 시스템 콜은 일반적으로 여러 가지 기능으로 나뉘며 각각에 번호가 할당된다. 시스템 콜 인터페이스는 이러한 번호를 인덱스로 사용하는 테이블을 유지한다.

3) System call 호출 루틴



System call은 다음과 같은 단계로 동작한다.

1. 프로그램에서 시스템 콜 함수를 호출
2. 운영 체제는 해당 함수의 인자를 확인, 시스템 콜 번호를 결정
3. 시스템 콜 번호를 기반으로, 시스템 콜 서비스 루틴의 주소를 검색
4. 운영 체제는 인터럽트를 발생시켜, 커널 모드로 전환
5. 커널 모드에서, 운영 체제는 시스템 콜 서비스 루틴을 실행함. 인자를 전달하고, 필요한 하드웨어 자원을 점유
6. 시스템 콜 서비스 루틴이 작업을 수행한 후, 결과 반환
7. CPU는 커널 모드에서 다시 사용자 모드로 전환. 운영 체제가 실행한 코드의 결과를 프로그램으로 전달, 프로그램 다시 실행

2. Kernel source code 수정

1) System call 번호 할당

: syscall_64.tbl

먼저, 새롭게 추가할 콜과 해당하는 번호를 정해주어야 한다. 이때 수정해야 할 파일은 syscall_64.tbl로, Linux 운영 체제에서 시스템 콜(System call) 번호와 해당 콜에 대한 함수를 정의하는 파일이다.

```
bonin@bonin-VirtualBox: /usr/src/linux-4.20.11$ sudo su
[sudo] password for bonin:
root@bonin-VirtualBox: /usr/src/linux-4.20.11# vi /arch/x86/entry/syscalls/syscall_64.tbl
```

syscall_64.tbl이 위치한 /usr/src/linux/arch/x86/entry/syscalls/ 경로로 접근하여 vim을 통해 파일 텍스트를 수정해준다.

```
332 common stack          __x64_sys_stack
333 common io_pgetevents   __x64_sys_io_pgetevents
334 common rseq            __x64_sys_rseq
#
#
#
335 common os2023_push     __x64_sys_os2023_push
336 common os2023_pop      __x64_sys_os2023_pop
#
#
# x32-specific system call numbers start at 512 to avoid cache impact
# for native 64-bit operation. The __x32_compat_sys stubs are created
# on-the-fly for compat_sys_*() compatibility system calls if X86_X32
# is defined.
#
512 x32 rt_sigaction       __x32_compat_sys_rt_sigaction
```

다음과 같이 추가할 시스템콜의 고유번호, 이름, 해당 콜에 대한 함수를 새롭게 입력시킨다. 이번 과제를 통해 추가할 콜은 스택에서 push하는 기능과 pop하는 기능으로, 아무것도 할당되지 않은 번호 335, 336에 각각 os2023_push와 os2023_pop을 추가하여 작성했다.

2) System call 함수의 prototype 정의

:syscalls.h

다음으로 syscalls.h 파일을 수정해 시스템 콜 번호에 대응하는 함수들의 원형을 추가해야 한다. syscalls.h 파일은 Linux 운영 체제에서 시스템 콜(System call) 인터페이스를 제공하는 헤더 파일이다. 이 파일은 user 영역 프로그램에서 시스템 콜을 호출할 때 필요한 함수들을 정의하고 있다.

```
bomin@bomin-VirtualBox:~$ sudo su
[sudo] password for bomin:
root@bomin-VirtualBox:/home/bomin# cd /usr/src/linux-4.20.11
root@bomin-VirtualBox:/usr/src/linux-4.20.11# vim include/linux/syscalls.h
```

syscalls.h가 위치한 (linux)/include/linux/ 경로에 접근하여 vim을 통해 파일 텍스트를 수정해준다.

```
static inline unsigned int sys_personality(unsigned int personality)
{
    unsigned int old = current->personality;

    if (personality != 0xffffffff)
        set_personality(personality);

    return old;
}

/*oslab*/
asmlinkage void sys_os2023_push(int);
asmlinkage int sys_os2023_pop(void);
#endif
-- INSERT --
```

위와 같이 asmlinkage를 사용하여 추가할 시스템 콜 함수들의 prototype을 정의하였다. 여기서 자료형 앞에 asmlinkage를 붙인 이유는 assembly 코드로 작성되는 인터럽트 핸들러에서도 C함수 호출이 가능하게끔 하기 위함이다.

3) 추가할 System call 함수 구현

:oslab_my_stack.c

이제 추가할 시스템 콜 소스를 작성할 차례이다. 실제로 스택에서 push와 pop을 어떤 과정으로 할 수 있는지 C코드로 구현한다.

```
bomin@bomin-VirtualBox:/usr/src/linux-4.20.11/kernel$ sudo su
[sudo] password for bomin:
root@bomin-VirtualBox:/usr/src/linux-4.20.11/kernel# vim oslab_my_stack.c
root@bomin-VirtualBox:/usr/src/linux-4.20.11/kernel#
```

/usr/src/linux-4.20.11/kernel/ 하위에 vim을 이용해 oslab_my_stack.c 라는 파일을 생성하고 새로운 파일에 코드를 작성했다.

```

bomin@bomin-VirtualBox: /usr/src/linux-4.20.11
File Edit View Search Terminal Help
#include <linux/syccalls.h>
#include <linux/kernel.h>
#include <linux/linkage.h>

int stack[50]; //stack declaration
int top=0; //top initialization
void print_stack(void);

SYSCALL_DEFINE1(os2023_push, int, a){
    int i;

    //overflow prevention
    if (top>=50){
        printk("os2023_push: Stack overflon\n"); //use 'printk' instead of 'printf'
        return;
    }

    //stack push
    stack[top++]=input;
    print_stack();

    return;
}

SYSCALL_DEFINE0(os2023_pop){
    int output;

    // Underflow prevention
    if (top<=0){
        printk("os2023_pop(): Stack underflow\n");
    }

    //stack pop
    output=stack[--top];
    print_stack();

    return output;
}

void print_stack(void){
    int i;

    printk("Stack Top-----\n");
    for (i=top-1;i>=0;--){
        printk("%d\n",stack[i]);
    }
    printk("Stack Bottom-----\n"); //print stack from top to bottom

    return;
}

```

작성한 코드는 다음과 같으며, 주석을 첨부하였다. 먼저 전역변수로 크기가 50인 int 배열 형태의 stack을 선언하였다. 이 스택에 input값을 push하기도 하고 나중에 들어오는 값부터 pop하기도 할 것이다. top은 stack에서 새로운 값을 받을, 혹은 내보낼 위치의 인덱스로, stack[0]에서부터 값을 채워야하기에 0으로 초기화했다.

이전에 push와 pop 두가지 기능을 구현하기 위한 각각의 시스템콜 함수를 os2023_push, os2023_pop으로 정의했었다. 이 둘을 더 효율적으로 정의하기 위해 SYSCALL_DEFINEx 매크로를 사용했다. SYSCALL_DEFINEx는 리눅스 커널 소스 코드에서 사용되는 매크로로, x는 파라미터 개수를 나타낸다. 이 매크로를 사용하면 시스템 콜 함수의 인자를 처리하고, 시스템 콜 결과를 반환하는 등의 기본적인 동작을 커널 내부에서 자동으로 처리할 수 있다.

Stack은 데이터의 삽입과 삭제가 한쪽 끝에서만 이뤄지는 후입선출(LIFO)의 방식으로 동작하는 자료구조다. 따라서 push는 스택의 맨 위에 input을 받도록 구현해야 한다. 맨 위 위치인 top의 인덱스를 차례로 하나씩 늘려가며 입력값인 a를 추가하는 코드를 작성했다. 다만, stack의 크기를 넘어서 삽입을 받을 수 없고, 스택에 이미 입력값이 존재하는 경우에도 추가로 삽입되지 않도록 해야한다. 이를 위한 코드도 조건문을 이용해 작성했다. Pop은 스택의 맨 위 값을 내보내는 기능으로, 역시 top 바로 이전 인덱스에 해당하는 값

을 output으로 정의하여 return하게끔 구현했다. Pop과 push 둘다 실행된 후에는 stack을 출력해야하는데, 중복되는 기능이므로 print_stack()이라는 함수를 하나 더 정의했다. Stack의 top부터 bottom까지 (인덱스 top-1 에서 0까지) 출력하게끔 반복문을 이용해 구현했다.

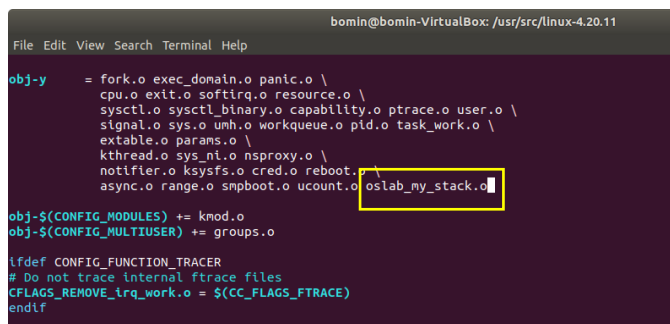
4) Makefile 수정

:Makefile

Makefile은 커널을 빌드하기 위해 필요한 규칙과 의존성을 정의하고, 컴파일러 및 링커 등의 도구들을 사용하여 커널 이미지를 생성하는 스크립트이다.

```
bomin@bomin-VirtualBox: /usr/src/linux-4.20.11/kernel$ sudo su
[sudo] password for bomin:
root@bomin-VirtualBox: /usr/src/linux-4.20.11/kernel# vim Makefile
```

(linux)/kernel/ 로 접근하여 vim을 이용해 Makefile을 수정한다.



```
bomin@bomin-VirtualBox: /usr/src/linux-4.20.11
File Edit View Search Terminal Help

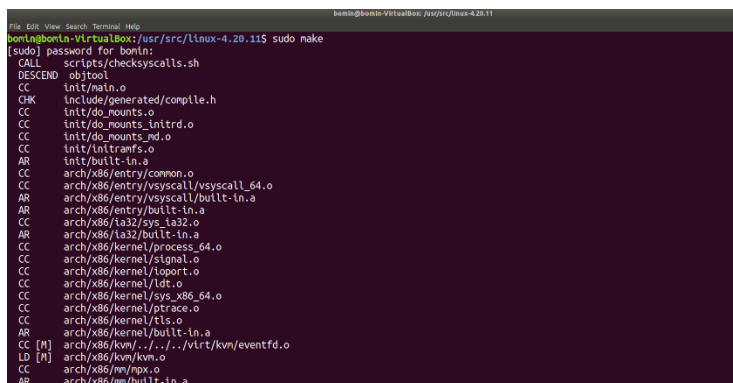
obj-y = fork.o exec_domain.o panic.o \
       cpu.o exit.o softirq.o resource.o \
       sysctl.o sysctl_binary.o capability.o ptrace.o user.o \
       signal.o sys.o umh.o workqueue.o pid.o task_work.o \
       extable.o params.o \
       kthread.o sys_ni.o nsproxy.o \
       notifier.o ksysfs.o cred.o reboot.o \
       async.o range.o smpboot.o ucount.o oslab_my_stack.o

obj-$(CONFIG_MODULES) += kmod.o
obj-$(CONFIG_MULTIUSER) += groups.o

ifdef CONFIG_FUNCTION_TRACER
# Do not trace internal ftrace files
CFLAGS_REMOVE_irq_work.o = $(CC_FLAGS_FTRACE)
endif
```

객체 파일명이 저장되어 있는 obj-y 에, 이전단계에서 구현한 시스템콜 함수가 담긴 oslab_my_stack.c를 obj파일로 변경한 oslab_my_stack.o를 추가해준다.

5) Kernel 컴파일 및 재부팅



```
bomin@bomin-VirtualBox: /usr/src/linux-4.20.11$ sudo make
[sudo] password for bomin:
CALL scripts/checksyscalls.sh
DESCEND objtool
CC init/main.o
CHK include/generated/compile.h
CC init/do_mounts.o
CC init/do_mounts_initrd.o
CC init/do_mounts_md.o
CC init/initramfs.o
AR init/built-in.a
CC arch/x86/entry/common.o
CC arch/x86/entry/vsyscall/vsyscall_64.o
AR arch/x86/entry/vsyscall/built-in.a
AR arch/x86/entry/built-in.a
CC arch/x86/ia32/sys_ia32.o
AR arch/x86/ia32/built-in.a
CC arch/x86/kernel/process_64.o
CC arch/x86/kernel/signal.o
CC arch/x86/kernel/toport.o
CC arch/x86/kernel/ldt.o
CC arch/x86/kernel/sys_x86_64.o
CC arch/x86/kernel/ptrace.o
CC arch/x86/kernel/tls.o
AR arch/x86/kernel/built-in.a
CC [M] arch/x86/kvm/../../../virt/kvm/eventfd.o
LD [M] arch/x86/kvm/kvm.o
CC arch/x86/mm/mpx.o
CC arch/x86/mm/built-in.a
AR
```

```
boem@boem-VirtualBox: /usr/src/linux-4.20.11$ sudo make install
sh ./arch/x86/boot/install.sh 4.20.11.oslab arch/x86/boot/bzImage \
System.map /boot
run-parts: executing /etc/kernel/postinst.d/apt-auto-removal 4.20.11.oslab /boot/vmlinuz-4.20.11.oslab
run-parts: executing /etc/kernel/postinst.d/initramfs-tools 4.20.11.oslab /boot/vmlinuz-4.20.11.oslab
update-initramfs: Generating /boot/initrd.img-4.20.11.oslab
run-parts: executing /etc/kernel/postinst.d/unattended-upgrades 4.20.11.oslab /boot/vmlinuz-4.20.11.oslab
run-parts: executing /etc/kernel/postinst.d/update-notifier 4.20.11.oslab /boot/vmlinuz-4.20.11.oslab
run-parts: executing /etc/kernel/postinst.d/vboxadd 4.20.11.oslab /boot/vmlinuz-4.20.11.oslab
VirtualBox Guest Additions: Building the modules for kernel 4.20.11.oslab.
VirtualBox Guest Additions: Look at /var/log/vboxadd-setup.log to find out what
went wrong
run-parts: executing /etc/kernel/postinst.d/xx-update-initrd-links 4.20.11.oslab /boot/vmlinuz-4.20.11.oslab
run-parts: executing /etc/kernel/postinst.d/zz-update-grub 4.20.11.oslab /boot/vmlinuz-4.20.11.oslab
Sourcing file /etc/default/grub
Generating grub configuration file ...
Found linux image: /boot/vmlinuz-4.20.11.oslab
Found initrd image: /boot/initrd.img-4.20.11.oslab
Found linux image: /boot/vmlinuz-4.20.11.oslab.old
Found initrd image: /boot/initrd.img-4.20.11.oslab
Found linux image: /boot/vmlinuz-4.18.0-15-generic
Found initrd image: /boot/initrd.img-4.18.0-15-generic
Found memtest86+ image: /boot/memtest86+.elf
Found memtest86+ image: /boot/memtest86+.bin
done
```

Sudo make, sudo make install 명령어를 통하여 커널을 컴파일하였다. 이 과정에서 생성/수정한 소스 코드가 컴퓨터에서 실행 가능한 바이너리 코드로 변환된다.

3. User application

1) 사용자 프로그램 구현

: call_my_stack.c

추가된 시스템콜이 잘 반영되었는지 확인하기 위한 user application을 구현하고자 한다. /usr/ 디렉토리 하위에 바로 cal_my_stack.c라는 유저 프로그램 파일을 생성했다.

```
#include<unistd.h>
#include<stdio.h>

#define my_stack_push 335 //declare system number
#define my_stack_pop 336

int main(){

    int r;

    syscall(my_stack_push,1);
    printf("PUSH %d\n",1);

    r=syscall(my_stack_push,1);
    printf("PUSH %d\n",1);

    r=syscall(my_stack_push,2);
    printf("PUSH %d\n",2);

    r=syscall(my_stack_push,3);
    printf("PUSH %d\n",3);

    r=syscall(my_stack_pop);
    printf("POP %d\n",r);

    r=syscall(my_stack_pop);
    printf("POP %d\n",r);

    r=syscall(my_stack_pop);
    printf("POP %d\n",r);

    return 0;
}
```

syscall()이라는 매크로 함수를 통해 새롭게 추가한 시스템 콜을 호출하기 위해 <unistd.h>을 헤더에 추가였다. 그리고 보다 용이한 함수 사용을 위해 각 함수의 시스템 콜 고유번호를 함수이름으로 define하였다. Define을 하면 syscall(시스템 콜 번호,

parameters) 대신에 syscall(함수이름, parameters) 로 함수를 호출할 수 있다.

Stack에 1,1,2,3을 차례로 push하고 이후에 세번 pop을 하여 잘 작동하는지 결과를 살펴 보려고 한다.

```
bomin@bomin-VirtualBox:~$ gcc call_my_stack.c -o call_my_stack
bomin@bomin-VirtualBox:~$ ls
call_my_stack  Desktop  Downloads  Music  Public  Videos
call_my_stack.c  Documents  examples.desktop  Pictures  Templates
bomin@bomin-VirtualBox:~$
```

call_my_stack.c 를 모두 작성했다면 이 파일을 컴파일 하여 실행파일로 만들어야한다. gcc <filename> -o <filename> 이라는 명령어를 통해 call_my_stack 의 obj파일도 생성하였다.

2) System call 및 사용자 프로그램 작동

call_my_stack 을 실행한 결과이다.

```
bomin@bomin-VirtualBox:~$ ./call_my_stack
PUSH 1
PUSH 1
PUSH 2
PUSH 3
POP 3
POP 2
POP 1
```

유저프로그램에서 의도한대로 push와 pop이 잘 작동한 것으로 보인다.

이후에는 dmesg를 통해서 oslab_call_stack.c의 printk로 원하던 출력이 나왔는지 확인하였다.

```
41.811271] [System Call] os2023_push:
41.811219] Stack Top-----
41.811219] 1
41.811220] Stack Bottom-----
41.811270] [System Call] os2023_push:
41.811270] Stack Top-----
41.811271] 1
41.811271] Stack Bottom-----
41.811274] [System Call] os2023_push:
41.811274] Stack Top-----
41.811274] 2
41.811275] 1
41.811275] Stack Bottom-----
41.811277] [System Call] os2023_push:
41.811278] Stack Top-----
41.811278] 3
41.811278] 2
41.811279] 1
41.811279] Stack Bottom-----
41.811281] [System Call] os2023_pop:
41.811281] Stack Top-----
41.811282] 2
41.811282] 1
41.811283] Stack Bottom-----
41.811284] [System Call] os2023_pop:
41.811285] Stack Top-----
41.811285] 1
41.811286] Stack Bottom-----
41.811287] [System Call] os2023_pop:
41.811288] Stack Top-----
41.811288] Stack Bottom-----
```

Stack에 데이터의 삽입과 삭제가 잘 반영된 것을 확인할 수 있다.

4. 결론

시스템콜을 추가하는 과제를 수행하면서, 운영체제와 시스템콜에 대해 깊이 이해할 수 있었다. 또한, 리눅스 커널 소스 코드를 직접 수정하여 컴파일하고 실행하는 경험을 통해, 운영체제와 커널의 동작 원리를 더욱 잘 이해할 수 있었다.

시스템콜은 운영체제와 프로그램 간의 인터페이스 역할을 하며, 시스템의 하드웨어와 소프트웨어를 관리하는 핵심적인 역할을 한다. 따라서, 시스템콜을 추가하고 이를 커널에 반영하는 과정에서는 커널의 내부 구조와 동작 방식을 자세히 이해하고 기능에 따라 수정할 수 있는 능력이 필요했다.

모든 과정이 순탄하지만은 않았는데, 맞닥뜨린 오류는 두가지가 있었다. 첫번째는 커널 컴파일 도중 알아낸 에러였다. 커널 컴파일 중에 'recipe for target 'kernel/oslab_my_stack.o' failed' 라는 에러 문구가 나왔고 컴파일이 중단됐다.

```
return;
^~~~~~
In file included from ./include/linux/compat.h:37:8,
from ./arch/x86/include/asm/ftrace.h:66,
from ./include/linux/ftrace.h:21,
from ./include/linux/perf_event.h:48,
from ./include/linux/trace_events.h:10,
from ./include/trace/syscall.h:7,
from ./include/linux/syscalls.h:85,
from kernel/oslab_my_stack.c:1:
./arch/x86/include/asm/syscall_wrapper.h:174:21: note: declared here
static inline long __do_sys#name(__MAP(x,__SC_DECL,__VA_ARGS__))
^
./include/linux/syscalls.h:225:2: note: in expansion of macro '__SYSCALL_DEFINE'
__SYSCALL_DEFINE(x, sname, __VA_ARGS__)
^
./include/linux/syscalls.h:214:36: note: in expansion of macro 'SYSCALL_DEFINE'
#define SYSCALL_DEFINE1(name, ...) SYSCALL_DEFINE(1, _#name, __VA_ARGS__)
^
kernel/oslab_my_stack.c:10:1: note: in expansion of macro 'SYSCALL_DEFINE1'
SYSCALL_DEFINE1(os2023_push, int, a){
^
kernel/oslab_my_stack.c:11:6: warning: unused variable 'i' [-Wunused-variable]
int i;
^
kernel/oslab_my_stack.c: In function 'print_stack':
kernel/oslab_my_stack.c:48:22: error: expected expression before ')' token
for (i=top-1;i>=0;i--){
^
scripts/Makefile.build:291: recipe for target 'kernel/oslab_my_stack.o' failed
make[1]: *** [kernel/oslab_my_stack.o] Error 1
Makefile:1058: recipe for target 'kernel' failed
make: *** [kernel] Error 2
bomin@bomin-VirtualBox: /usr/src/linux-4.20.11$
```

문제는 oslab_my_stack.c파일을 작성할 때 발생한 type error로부터 발생한 것이었고, for 문을 작성할 때 for(i=top-1;i>=0;i--)로 쓴다는 것을 for(i=top-1;i>=0;--i)로 쓰는 미세한 실수로 인해 컴파일 에러가 났다. 다시 oslab_my_stack.c 파일에 접근해 systax에러를 수정하였고, 커널 컴파일도 문제없이 돌아갔다.

두번째 오류는 user application 실행 후, 작성한 push와 pop 함수가 동작하지 않아 결과가 다르게 나오는 것이었다. oslab_my_stack이 전혀 반영되지 않은 듯했다.

이러한 문제는 가상머신을 재부팅 시켰을 때 ubuntu의 버전이 수정을 반영한 버전과 다르게 설정되어 있어 발생하였다. 따라서 재부팅 시 왼쪽시프트키를 누르고 Advanced options for Ubuntu 에서 Linux 4.20.11 커널로 재설정하였다.

```
bomin@bomin-VirtualBox: /usr/src/linux-4.20.11$ cd
bomin@bomin-VirtualBox: ~$ uname -r
4.20.11.oslab
bomin@bomin-VirtualBox: ~$
```

Unamae 확인하니 비로소 linux-4.20.11 버전으로 돌아온 것이 보였고, 변경내용이 적용된 커널을 불러오니 user application이 잘 동작했다.

```
File Edit View Search Terminal Tabs Help
#linux/Makefile (linux-4.20.11)
bonin@bonin-VirtualBox: /usr/src/linux-4.20.11$ sudo su
Found, did you mean:
command 'sudo' from deb sudo
Try: sudo apt install <deb name>
bonin@bonin-VirtualBox: /usr/src/linux-4.20.11$ sudo su
[sudo] password for bonin:
root@bonin-VirtualBox: /usr/src/linux-4.20.11# cd arch/x86/entry/syscalls
root@bonin-VirtualBox: /usr/src/linux-4.20.11/arch/x86/entry/syscalls# vln syscall_64.tbl
root@bonin-VirtualBox: /usr/src/linux-4.20.11/arch/x86/entry/syscalls# su - bonin
bonin@bonin-VirtualBox:~$ cd /usr/src/linux-4.20.11
bonin@bonin-VirtualBox: /usr/src/linux-4.20.11$ sudo make
SYSHDR arch/x86/include/generated/asm/unistd_64_x32.h
SYSTBL arch/x86/include/generated/asm/syscalls_64.h
SYSHDR arch/x86/include/generated/uapi/asm/unistd_64.h
SYSHDR arch/x86/include/generated/uapi/asm/unistd_x32.h
CC arch/x86/kernel/asm-offsets.s
CALL scripts/checksyscalls.sh
DESCEND objtool
CC init/main.o
CHK include/generated/compile.h
CC init/version.o
CC init/do_mounts.o
CC init/do_mounts_initrd.o
CC init/do_mounts_md.o
CC init/initramfs.o
CC init/init_task.o
AR init/built-in.a
CC arch/x86/crypto/crc32c-intel_glue.o
AR arch/x86/crypto/built-in.a
CC [M] arch/x86/crypto/glue_helper.o
CC [M] arch/x86/crypto/aes_glue.o
LD [M] arch/x86/crypto/aes-x86_64.o
CC [M] arch/x86/crypto/des3_edg_glue.o
LD [M] arch/x86/crypto/des3_edg-x86_64.o
CC [M] arch/x86/crypto/camellia_glue.o
LD [M] arch/x86/crypto/camellia-x86_64.o
CC [M] arch/x86/crypto/blowfish_glue.o
LD [M] arch/x86/crypto/blowfish-x86_64.o
CC [M] arch/x86/crypto/twofish_glue.o
LD [M] arch/x86/crypto/twofish-x86_64.o
CC [M] arch/x86/crypto/twofish_3way.o
LD [M] arch/x86/crypto/twofish_64_3way.o
CC [M] arch/x86/crypto/chacha20_glue.o
LD [M] arch/x86/crypto/chacha20-x86_64.o
CC [M] arch/x86/crypto/serpent_sse2_glue.o
LD [M] arch/x86/crypto/serpent-sse2-x86_64.o
```