

# **COSE341**

# **Operating System**

Department: 컴퓨터학과

Student Number: 2022320039

Name: 정진욱

Submission date: 4/9

Freedays: 0

## **Development environment**

- **IDE: vi**
- **Linux version: 4.20.11**
- **Ubuntu version: 18.04.2**

## **Explanation of system calls on Linux (including call routines)**

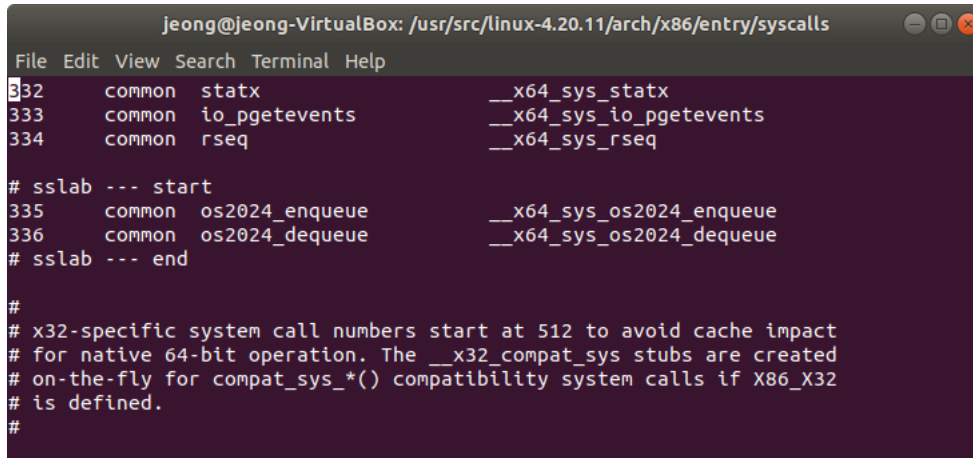
**System call:** Interface between the user space and the kernel space. They give user programs a means of requesting services and carrying out privileged actions, such as file operations, process management, network connectivity, and hardware access. By system call user program go from user mode to kernel mode. After doing the requested operation in kernel mode, the kernel returns to the caller program. System calls are essential for giving user programs access to the operating system's features while preserving security and stability.

### **Call routines on Linux:**

1. User program initiates a system call. It is possible to directly call system calls, but usually program would use high level application programming interfaces (APIs) such as standard C library.
2. Transition from user mode to kernel mode is triggered by a software interrupt or an exception.
3. Kernel determines which system call the user program is requesting by system call number or identifier.
4. Linux maintains a system call table to map the system call number to the corresponding system call handler.
5. Kernel invokes the corresponding system call handler which is a function within the kernel that implements the functionality associated with the specific system call.
6. Kernel copies the arguments from user space memory to kernel space memory for the system call handler to access. Copying parameter protect the kernel from malicious situations by separating user and kernel space.
7. Handler executes the requested operation.
8. Finally, copy the provided result and kernel transitions the execution back to user mode.

## Implementation

### 1. syscall\_64.tbl

A terminal window titled 'jeong@jeong-VirtualBox: /usr/src/linux-4.20.11/arch/x86/entry/syscalls'. The window shows the content of the 'syscall\_64.tbl' file. It lists system call numbers 332, 333, and 334, each with a 'common' type and a name. For each, there is a corresponding '\_\_x64\_sys\_' stub name. A comment block follows, starting with '# sslab --- start' and ending with '# sslab --- end', containing entries for 'os2024\_enqueue' and 'os2024\_dequeue'. Another comment block explains that x32-specific system call numbers start at 512 to avoid cache impact for native 64-bit operation, and that '\_\_\_x32\_compat\_sys' stubs are created on-the-fly for compatibility.

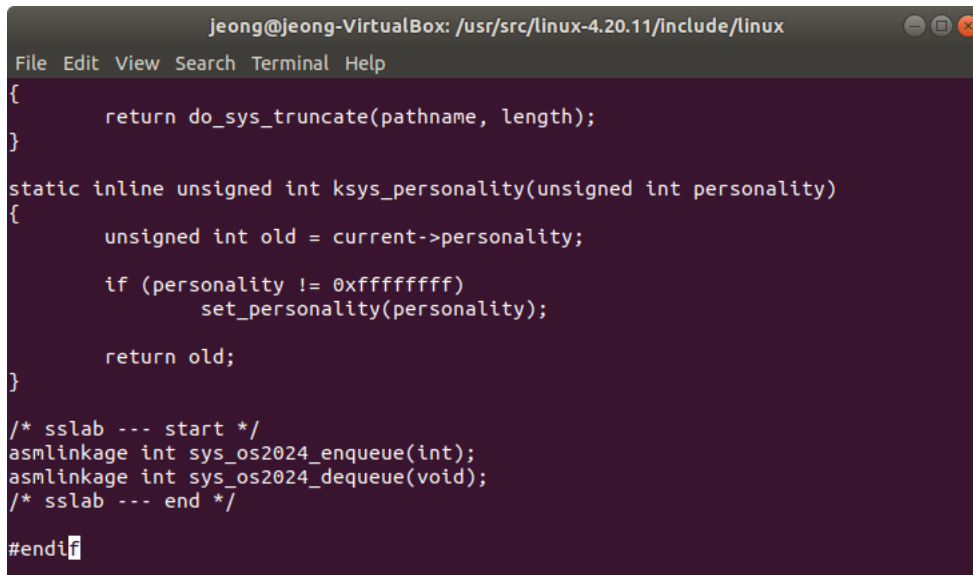
```
jeong@jeong-VirtualBox: /usr/src/linux-4.20.11/arch/x86/entry/syscalls
File Edit View Search Terminal Help
332      common  statx              __x64_sys_statx
333      common  io_pgetevents      __x64_sys_io_pgetevents
334      common  rseq                __x64_sys_rseq

# sslab --- start
335      common  os2024_enqueue      __x64_sys_os2024_enqueue
336      common  os2024_dequeue      __x64_sys_os2024_dequeue
# sslab --- end

#
# 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.
#
```

Add the implemented system call in the syscall table with new system call number. When a system call is called, the kernel finds the called system call through this system call table and executes it.

### 2. syscalls.h

A terminal window titled 'jeong@jeong-VirtualBox: /usr/src/linux-4.20.11/include/linux'. The window shows the content of the 'syscalls.h' file. It contains a function definition for 'do\_sys\_truncate', a static inline function 'ksys\_personality', and a comment block for 'os2024\_enqueue' and 'os2024\_dequeue' with 'asmlinkage' prototypes. The file ends with '#endif'.

```
jeong@jeong-VirtualBox: /usr/src/linux-4.20.11/include/linux
File Edit View Search Terminal Help
{
    return do_sys_truncate(pathname, length);
}

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

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

    return old;
}

/* sslab --- start */
asmlinkage int sys_os2024_enqueue(int);
asmlinkage int sys_os2024_dequeue(void);
/* sslab --- end */

#endif
```

Define prototypes of new system call functions using asmlinkage enables calling of C functions within assembly code.

### 3. sslab\_my\_queue.c

```
jeong@jeong-VirtualBox: /usr/src/linux-4.20.11/kernel
#include <linux/syscalls.h>
#include <linux/kernel.h>
#include <linux/linkage.h>
#define MAXSIZE 200
int queue[MAXSIZE] = {0, };
int front = 0;
int rear = 0;
int i, r;

SYSCALL_DEFINE1(os2024_enqueue, int, a) {
// Check if the queue is full
if (rear >= MAXSIZE - 1) {
    printk(KERN_INFO "[Error] - Queue is full-----\n");
    return -1;
}

// Check if the value is already exist in the queue
for (i = 0; i <= rear; i++) {
    if (queue[i] == a) {
        printk(KERN_INFO "[Error] - Already exist in the queue\n");
        return a;
    }
}

queue[rear] = a; // enqueue

printk(KERN_INFO "[System call] os2024_enqueue(); -----\n");
printk("Queue Front ----- \n");
for (i = front; i <= rear; i++) {
    printk("%d\n", queue[i]);
}
printk("Queue Rear ----- \n");

rear++; // increase rear index
return a;
}

SYSCALL_DEFINE0(os2024_dequeue) {
// Check if the queue is empty
if (front >= rear) {
    printk(KERN_INFO "[Error] - Queue is empty-----\n");
    return -1;
}

r = queue[front]; // return value of dequeue

printk(KERN_INFO "[System call] os2024_dequeue(); -----\n");
printk("Queue Front ----- \n");
for (i = front + 1; i < rear; i++) {
    printk("%d\n", queue[i]);
    queue[i-1] = queue[i]; // move every value forward
}
printk("Queue Rear ----- \n");

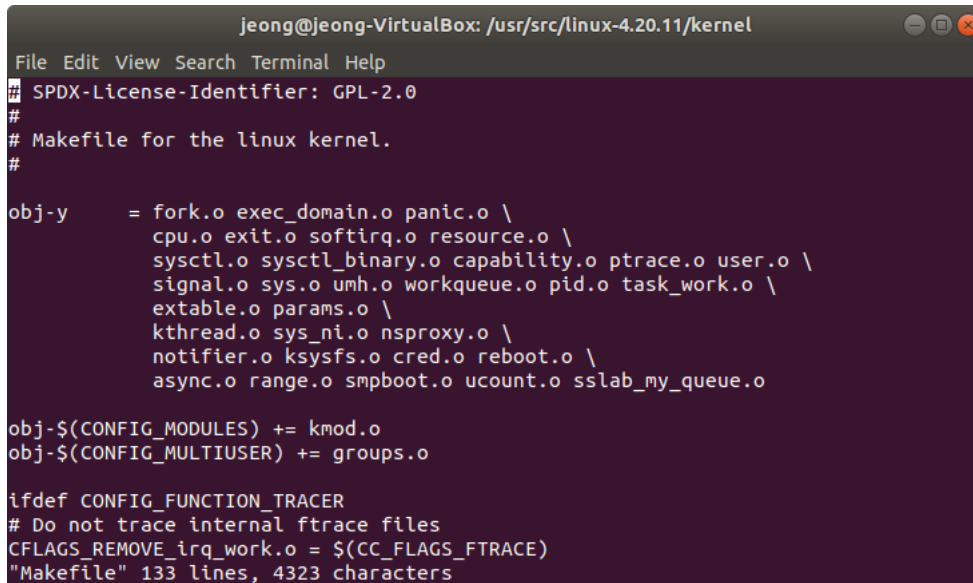
rear--; // decrease rear index
queue[rear] = 0; // init rear
return r;
}
```

In this file, I implement what system calls will task. First, I declare a queue in the form of an integer array as global variable.

Syscall os2024\_enqueue first checks if the queue is full and if there are any duplicates of the incoming parameter already existing in the queue. Then, it inserts the parameter value at the rear index of the queue and increments the value of rear by 1 to complete the enqueue operation.

Syscall os2024\_dequeue first checks if the queue is empty. Then, it keep the first value in the queue and move every value forward, which ensures that dequeue return values of queue[0]. Finally, decreas rear by 1, reset queue[rear] empty and return dequeue value.

#### 4. Makefile



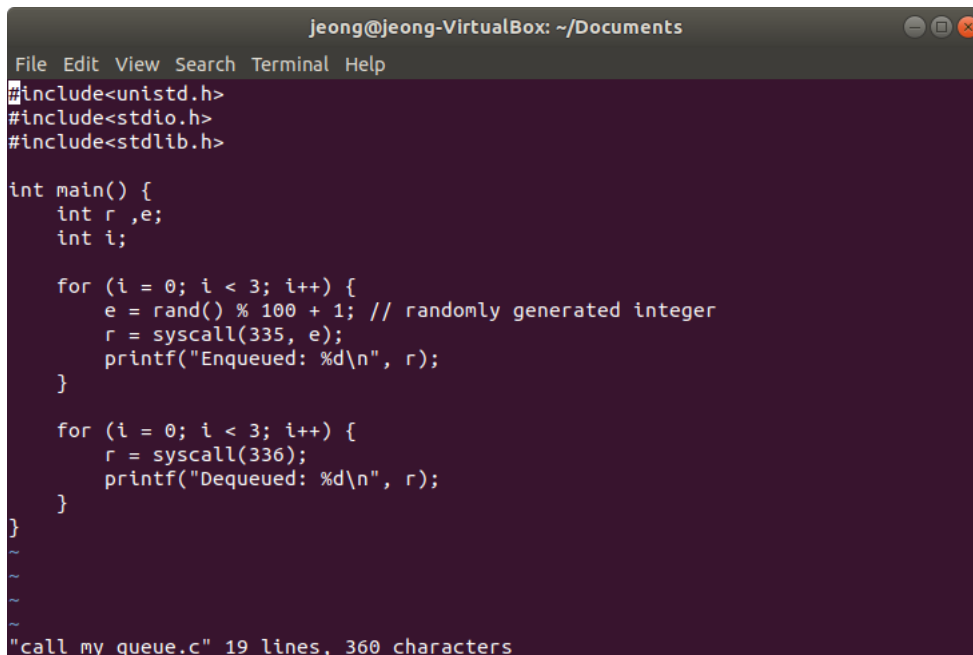
```
jeong@jeong-VirtualBox: /usr/src/linux-4.20.11/kernel
File Edit View Search Terminal Help
# SPDX-License-Identifier: GPL-2.0
#
# Makefile for the linux kernel.
#
obj-y      = fork.o exec_domain.o panic.o \
            cpu.o exit.o softirq.o resource.o \
            sysctl.o sysctl_binary.o capability.o 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 sslab_my_queue.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)
"Makefile" 133 lines, 4323 characters
```

Add new system call object file to be compiled within kernel.

#### 5. call\_my\_queue.c (User Application)



```
jeong@jeong-VirtualBox: ~/Documents
File Edit View Search Terminal Help
#include<unistd.h>
#include<stdio.h>
#include<stdlib.h>

int main() {
    int r ,e;
    int i;

    for (i = 0; i < 3; i++) {
        e = rand() % 100 + 1; // randomly generated integer
        r = syscall(335, e);
        printf("Enqueued: %d\n", r);
    }

    for (i = 0; i < 3; i++) {
        r = syscall(336);
        printf("Dequeued: %d\n", r);
    }
}

~
~
~
"call_my_queue.c" 19 lines, 360 characters
```

In user application, I pass the system call number directly using a macro function, `syscall()`. I generate three random integers from 1 to 100 to avoid 0, which represent the empty in queue. Application enqueue with these three integers and dequeue three times.

## Snapshot of project execution

### User application

```
jeong@jeong-VirtualBox:~/Documents$ ./call_my_queue
Enqueued: 83
Enqueued: 86
Enqueued: 77
Dequeued: 83
Dequeued: 86
Dequeued: 77
```

### Kernel

```
jeong@jeong-VirtualBox: ~/Documents
File Edit View Search Terminal Help
[ 343.095899] [System call] os2024_enqueue(); -----
[ 343.095900] Queue Front -----
[ 343.095900] 83
[ 343.095901] Queue Rear -----
[ 343.095971] [System call] os2024_enqueue(); -----
[ 343.095971] Queue Front -----
[ 343.095971] 83
[ 343.095972] 86
[ 343.095972] Queue Rear -----
[ 343.095974] [System call] os2024_enqueue(); -----
[ 343.095975] Queue Front -----
[ 343.095975] 83
[ 343.095975] 86
[ 343.095976] 77
[ 343.095976] Queue Rear -----
[ 343.095978] [System call] os2024_dequeue(); -----
[ 343.095978] Queue Front -----
[ 343.095979] 86
[ 343.095979] 77
[ 343.095979] Queue Rear -----
[ 343.095981] [System call] os2024_dequeue(); -----
[ 343.095981] Queue Front -----
[ 343.095982] 77
[ 343.095982] Queue Rear -----
[ 343.095983] [System call] os2024_dequeue(); -----
[ 343.095984] Queue Front -----
[ 343.095984] Queue Rear -----
jeong@jeong-VirtualBox:~/Documents$
```

## Difficulties and your efforts encountered during this project.

While working on the assignment, some commands took an unexpectedly long time to complete in certain terminals. Due to my limited experience with working in the Linux terminal, I wasn't sure if there were errors occurring, and I continued the tasks without realizing that the results were incorrect. As a result, I had to repeat the same process multiple times. Fortunately, I started the assignment early and was able to finish it on time.