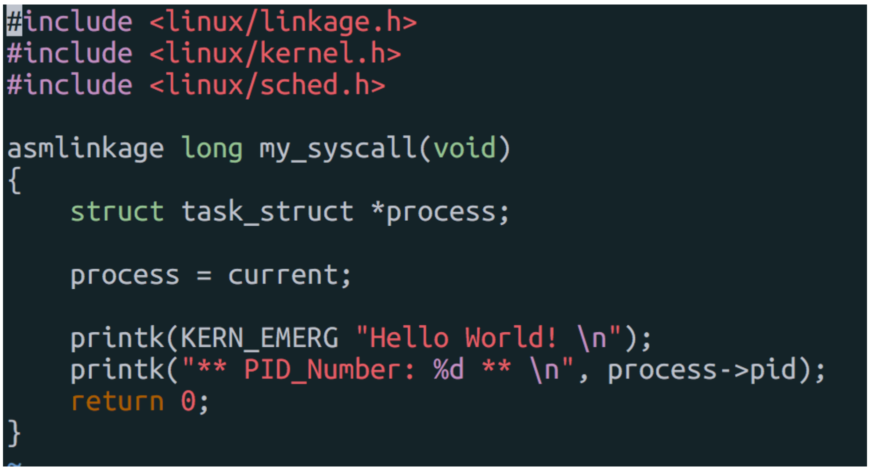
Exercise #3

The aim of this exercise is to implement system calls and invoke it from user-space.

Task 1

Write a new system call and a user-space process that invokes it. Test correctness and when invoking the system call, it should print pid (process ID) of the invoking process.

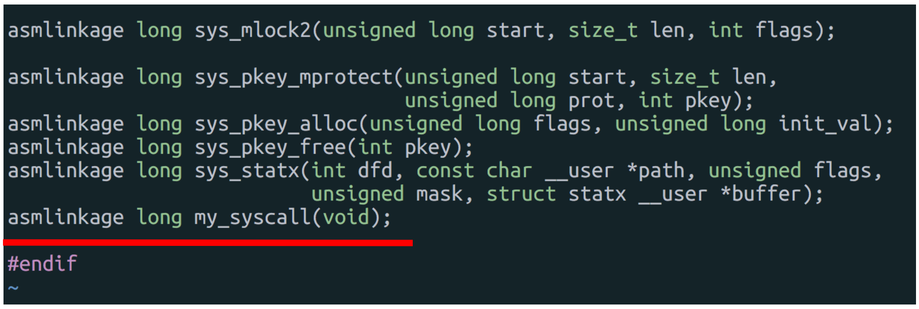
Step 1. Add the function of system call definition at /usr/src/linux-4.15.1/kernel



```
#include <linux/linkage.h>
#include <linux/kernel.h>
#include <linux/sched.h>

asmlinkage long my_syscall(void)
{
    struct task_struct *process;

    process = current;

    printk(KERN_EMERG "Hello World! \n");
    printk("** PID_Number: %d ** \n", process->pid);
    return 0;
}
```

Figure 1. my_syscall.c

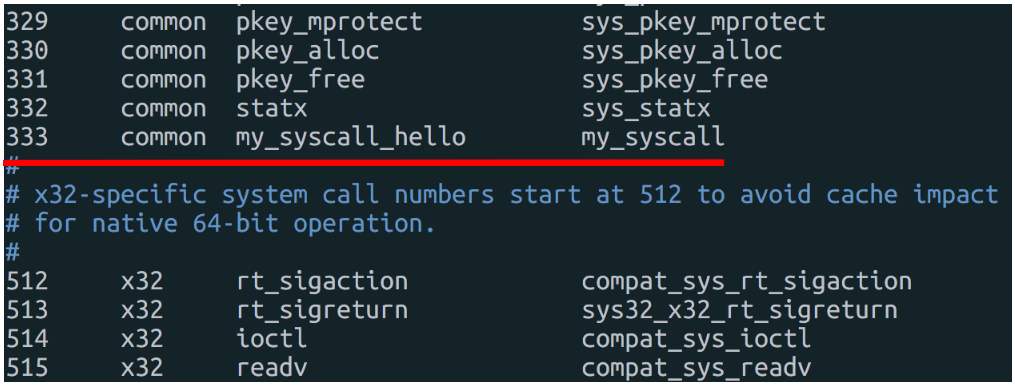
Step 2. Add the function prototype in the header file in the file usr/src/linux-4.15.1/include/linux/syscalls.h



```
asmlinkage long sys_mlock2(unsigned long start, size_t len, int flags);

asmlinkage long sys_pkey_mprotect(unsigned long start, size_t len,
                                  unsigned long prot, int pkey);
asmlinkage long sys_pkey_alloc(unsigned long flags, unsigned long init_val);
asmlinkage long sys_pkey_free(int pkey);
asmlinkage long sys_statx(int dfd, const char __user *path, unsigned flags,
                          unsigned mask, struct statx __user *buffer);
asmlinkage long my_syscall(void);

#endif
~
```

Figure 2. syscalls.h

Step 3. Create an entry in system call table in the file /usr/src/linux-4.15.1/arch/x86/entry/syscalls/syscall_64.tbl

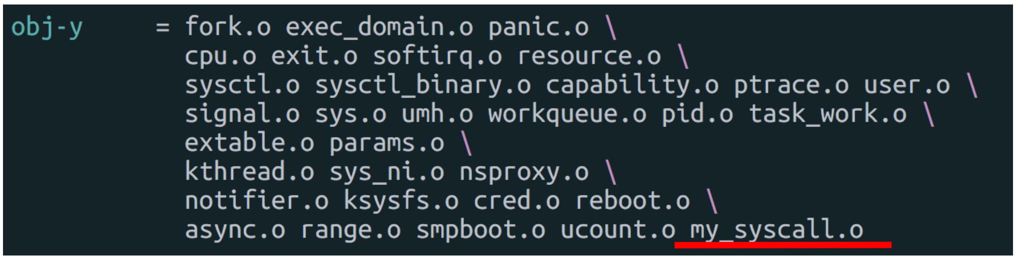


```
329     common  pkey_mprotect           sys_pkey_mprotect
330     common  pkey_alloc              sys_pkey_alloc
331     common  pkey_free               sys_pkey_free
332     common  statx                   sys_statx
333     common  my_syscall_hello        my_syscall
#
# x32-specific system call numbers start at 512 to avoid cache impact
# for native 64-bit operation.
#
512     x32     rt_sigaction            compat_sys_rt_sigaction
513     x32     rt_sigreturn            sys32_x32_rt_sigreturn
514     x32     ioctl                   compat_sys_ioctl
515     x32     readv                   compat_sys_readv
```

Figure 3. syscall_64.tbl

The number of 333 is going to be number of our own system call. 'common' means that the system call can run on both 64 and 32 bits' architecture.

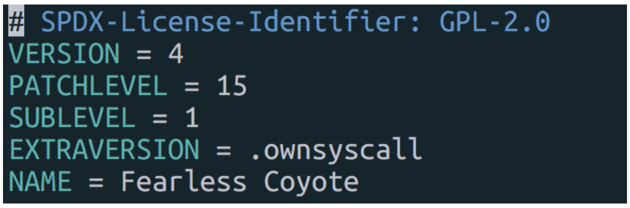
Step 4. Add the Makefile for compiling our system call function at /usr/src/linux-4.15.1/Makefile



```
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 my_syscall.o
```

Figure 4. Makefile (/kernel)

Step 5. To create our own kernel images, change the EXTRAVERSION in the main Makefile at /usr/src/linux-4.15.1



```
# SPDX-License-Identifier: GPL-2.0
VERSION = 4
PATCHLEVEL = 15
SUBLEVEL = 1
EXTRAVERSION = .ownsyscall
NAME = Fearless Coyote
```

Figure 5. Makefile

Step 6. Compile the function and build & install the modules

sudo make -j 24

sudo make modules -j 24

sudo make modules_install -j 24

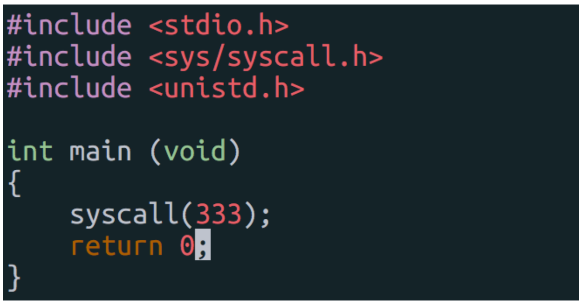
sudo make install -j 24

-j can compile and build by using several cores. -j 24 means that using 24 CPU cores. Before use this option, check the number of core using **cat /proc/cpuinfo**

Step 7. Reboot with our own imange.

sudo reboot

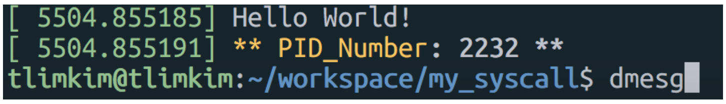
Step 8. Write a simple C application (user level) for calling our own system call.



```
#include <stdio.h>
#include <sys/syscall.h>
#include <unistd.h>

int main (void)
{
    syscall(333);
    return 0;
}
```

Figure 6. User level C source file

Step 9. Compile by this source file by using gcc and check the result of our own system call by 'dmesg'



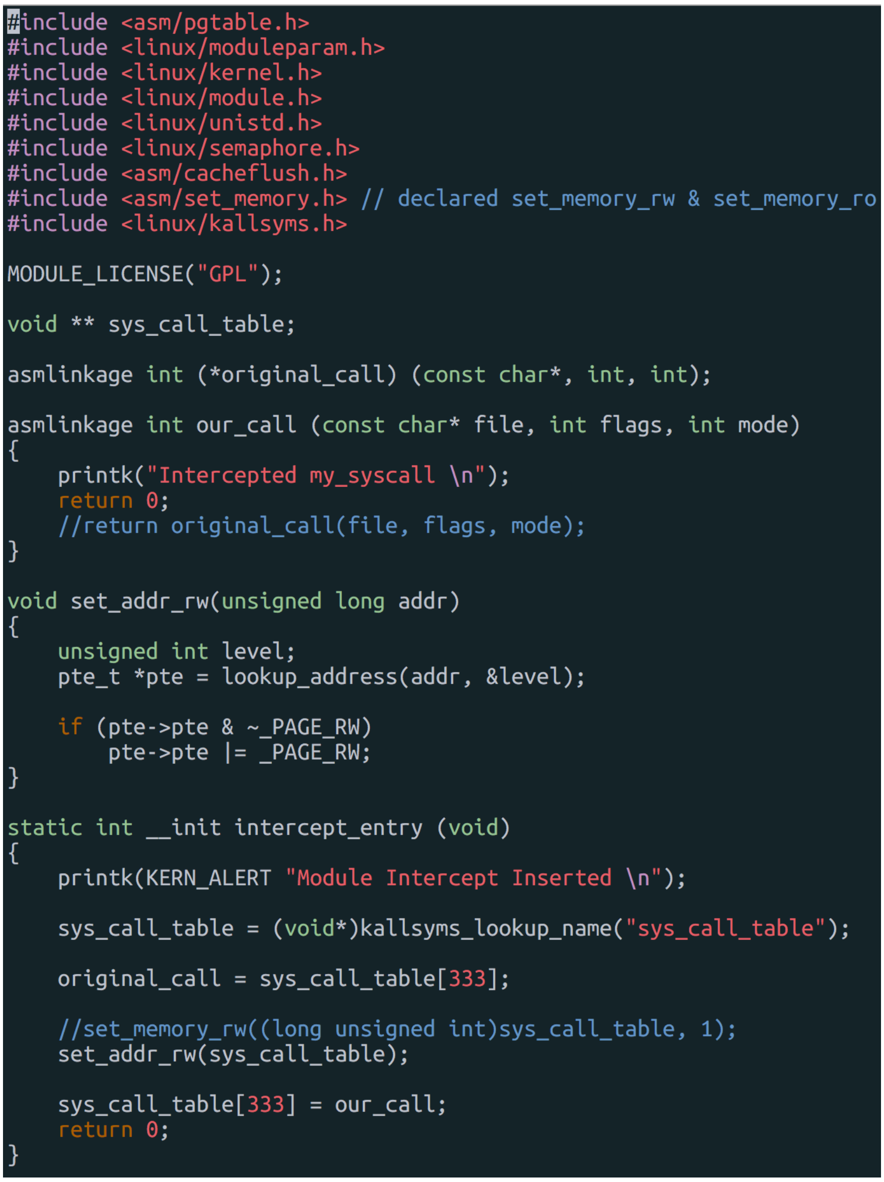
```
[ 5504.855185] Hello World!
[ 5504.855191] ** PID_Number: 2232 **
tlimkim@tlimkim:~/workspace/my_syscall$ dmesg
```

Figure 7. dmesg result

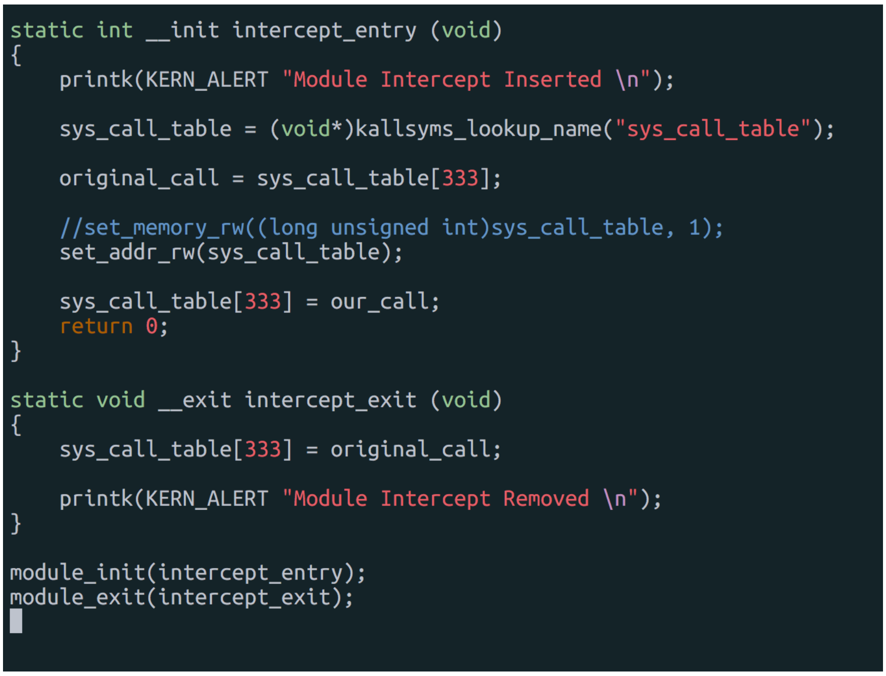
Task 2

Write a Linux kernel module to intercept this new system call and alter its functionality. When the module is removed, the original system call functionality should be restored.

Step 1. Write the module source file which works for intercepting our new system call.



```c
#include <asm/pgtable.h>
#include <linux/moduleparam.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/unistd.h>
#include <linux/semaphore.h>
#include <asm/cacheflush.h>
#include <asm/set_memory.h> // declared set_memory_rw & set_memory_ro
#include <linux/kallsyms.h>

MODULE_LICENSE("GPL");

void ** sys_call_table;

asmlinkage int (*original_call) (const char*, int, int);

asmlinkage int our_call (const char* file, int flags, int mode)
{
    printk("Intercepted my_syscall \n");
    return 0;
    //return original_call(file, flags, mode);
}

void set_addr_rw(unsigned long addr)
{
    unsigned int level;
    pte_t *pte = lookup_address(addr, &level);

    if (pte->pte & ~_PAGE_RW)
        pte->pte |= _PAGE_RW;
}

static int __init intercept_entry (void)
{
    printk(KERN_ALERT "Module Intercept Inserted \n");

    sys_call_table = (void*)kallsyms_lookup_name("sys_call_table");

    original_call = sys_call_table[333];

    //set_memory_rw((long unsigned int)sys_call_table, 1);
    set_addr_rw(sys_call_table);

    sys_call_table[333] = our_call;
    return 0;
}
```



```
static int __init intercept_entry (void)
{
    printk(KERN_ALERT "Module Intercept Inserted \n");

    sys_call_table = (void*)kallsyms_lookup_name("sys_call_table");

    original_call = sys_call_table[333];

    //set_memory_rw((long unsigned int)sys_call_table, 1);
    set_addr_rw(sys_call_table);

    sys_call_table[333] = our_call;
    return 0;
}

static void __exit intercept_exit (void)
{
    sys_call_table[333] = original_call;

    printk(KERN_ALERT "Module Intercept Removed \n");
}

module_init(intercept_entry);
module_exit(intercept_exit);
```

1. **'__init intercept_entry'**

This function is for starting the module that works main feature. By printing the messages which this module is inserted, it assigned address of "sys_call_table" to the variable 'sys_call_table'. Our own system call's number is 333, so we assign original system call to variable 'original_call'. Next, to change this call sys_call_table[333] to 'our_call', we should change the authority of read and write. There are 'set_memory_rw' function that change this authority, but there are pointer bugs so create new function named 'set_addr_rw'. By using this function, we assign 'our_call' to 'sys_call_table[333]'.

2. **'our_call'**

This function works for intercepting original system call. It returns 0, so the original system call can be ignored by this function.

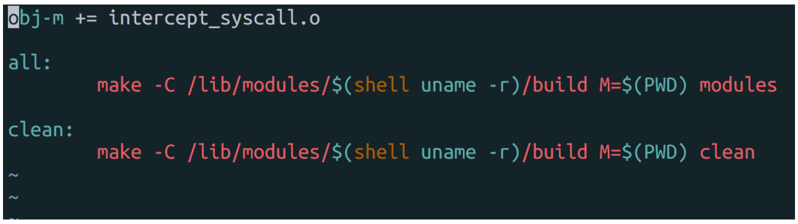
3. **'set_addr_rw'**

This function is to find the page table entry for the address and set it writable. It uses 'lookup_address' function that is find the page table entry for a virtual address. It returns the entry and the level of mapping. Then we change it to writable to assign 'our_call'.

4. '__exit intercept_exit'

This function works while we remove module from kernel. When we remove the module, the original system call should do their original function. So we assign 'original_call' to 'sys_call_table[333]'.

Step 2. Create Makefile for compiling the source code



```
obj-m += intercept_syscall.o

all:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
~
~
```

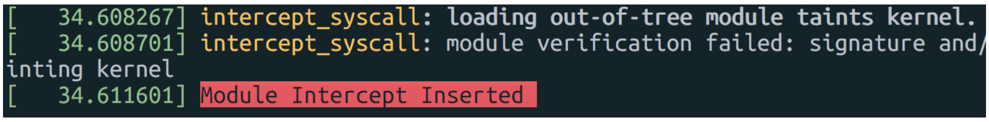
Figure 8. Makefile

Step 3. Compile the module and insert the module on kernel

sudo make

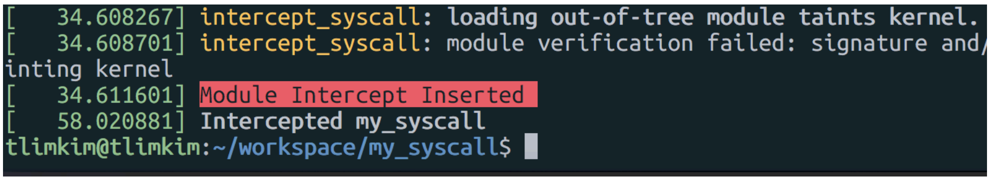
sudo insmod intercept_syscall.ko

Step 4. Check the result of intercepting system call.



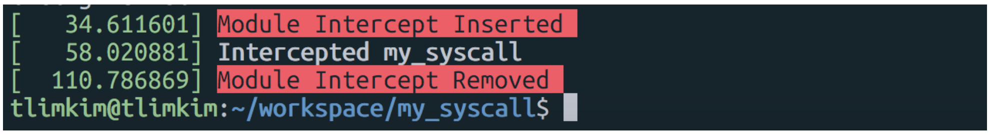
```
[   34.608267] intercept_syscall: loading out-of-tree module taints kernel.
[   34.608701] intercept_syscall: module verification failed: signature and/
inting kernel
[   34.611601] Module Intercept Inserted
```

Figure 9. Intercepting Module Inserted



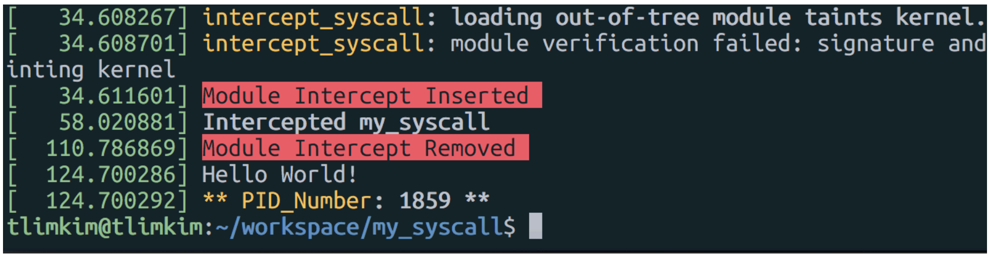
```
[   34.608267] intercept_syscall: loading out-of-tree module taints kernel.
[   34.608701] intercept_syscall: module verification failed: signature and/
inting kernel
[   34.611601] Module Intercept Inserted
[   58.020881] Intercepted my_syscall
tlimkim@tlimkim:~/workspace/my_syscall$
```

Figure 10. While calling my_syscall, Intercepted



```
[   34.611601] Module Intercept Inserted
[   58.020881] Intercepted my_syscall
[  110.786869] Module Intercept Removed
tlimkim@tlimkim:~/workspace/my_syscall$ 
```

Figure 11. Removed Module



```
[   34.608267] intercept_syscall: loading out-of-tree module taints kernel.
[   34.608701] intercept_syscall: module verification failed: signature and
inting kernel
[   34.611601] Module Intercept Inserted
[   58.020881] Intercepted my_syscall
[  110.786869] Module Intercept Removed
[  124.700286] Hello World!
[  124.700292] ** PID_Number: 1859 **
tlimkim@tlimkim:~/workspace/my_syscall$ 
```

Figure 12. Printing 'my_syscall' result ordinarily

** Additional things

- In task 2, there should be include one more header file “<linux/mm.h>”.

<<References>>

- **Adding system call**:
  http://harryp.tistory.com/69,
  http://lists.kernelnewbies.org/pipermail/kernelnewbies/2013-July/008598.html
  https://medium.com/@ssreehari/implementing-a-system-call-in-linux-kernel-4-7-1-6f98250a8c38
- **Implementing hooking system call**:
  https://stackoverflow.com/questions/14415561/intercepting-a-system-call
  https://stackoverflow.com/questions/2103315/linux-kernel-system-call-hooking-example