

Lab1 Report

All codes are open-sourced at [GitHub](#).

Part 1

Overall:

Run `make & make test` to see the results.

Run `make clean` to clean all the generated files.

Detail:

We first define both the init (loading) and exit (unloading) functions as static.

```
static int __init hello_world_init(void) {  
    printk(KERN_INFO "Hello World!\n");  
    return 0;  
}  
  
static void __exit hello_world_exit(void) {  
    printk(KERN_INFO "Goodbye World!\n");  
}
```

And at the end of the file, we call `module_init` and `module_exit` to determine the loading and unloading functions for kernels.

```
module_init(hello_world_init);  
module_exit(hello_world_exit);
```

We write an Makefile to compile the file. (Note use tab instead of space)

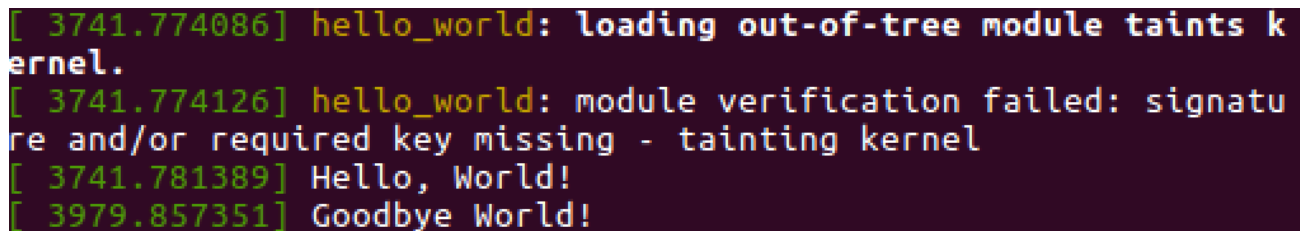
```
obj-m += hello_world.o
all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean

test:
    sudo dmesg -C
    sudo insmod hello_world.ko
    sudo rmmod hello_world.ko
    dmesg
```

However, when we first write the codes and try to run the following commands:

```
make
sudo insmod hello_world.ko
sudo rmmod hello_world.ko
dmesg
```

It does work but with the following errors:

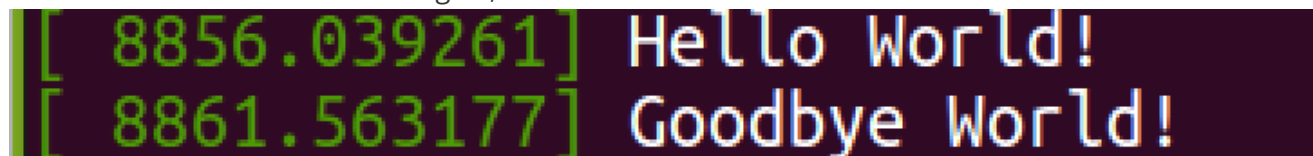


```
[ 3741.774086] hello_world: loading out-of-tree module taints kernel.
[ 3741.774126] hello_world: module verification failed: signature and/or required key missing - tainting kernel
[ 3741.781389] Hello, World!
[ 3979.857351] Goodbye World!
```

The straight-forward solution is to unenale module verification which is the solution I take.

Just add one line `CONFIG_MODULE_SIG=n` to the top of the MakeFile:

Then run the above commands again, it works now.



```
[ 8856.039261] Hello World!
[ 8861.563177] Goodbye World!
```

ref:

<https://blog.sourcerer.io/writing-a-simple-linux-kernel-module-d9dc3762c234>

Part 2

Overall:

Describe why you think the Linux kernel no longer allows loadable kernel modules to modify syscall tables or add system calls: The major advantages of the loadable modules over the built-in modules is that we can load/unload them on runtime. This is very beneficial when we are working on a module and testing it. While for the built-in module, if we make any changes, we need to compile the whole kernel and then reboot the system with the new image of the kernel. However, with the flexibility and power of the loadable module, it can be easily be misused by a malicious user, such as intercepting a system call if it is allowed to modify syscall tables. If we carefully save the addresses of the original system call function, we can even invoke it from within our own "interceptor codes", which allowing the interceptor codes perform as if nothing happened. Indeed, many rootkits have been exploiting this to maintain a presence on the system after it left.

What happens if you call `syscall()` with a bogus syscall number that doesn't exist? How could a userspace program prevent that error without relying on the kernel to catch it?

As shown in the [Linux manual page](#), calling `syscall()` with a bogus syscall number returns -1, and an error number is stored in `errno`. Nothing else will be done.

It is rather rare that we invoke system calls directly (the [Linux manual page](#) even makes note of this).

It's more common and convenient for programmers to use library wrappers such as `write`, `read` and `open`. In this case, if the programmer attempt to invoke a non-exist syscall, it's very likely the wrapper doesn't exist. And the program may not even be able to get past the runtime linker and fail to start.

Describe which files you needed to change to add a new system call and how you changed them to accommodate your new syscall. Also describe which safety conditions you check within your kernel implementation of `memuse_syscall`, and why each check is necessary. What, if any, extra arguments did you add to the syscall to help check whether the inputs coming from userspace are valid?

As shown in the `part2.patch`, we modify the following three files and create an directory with two files in it.

To ensure safety, we obtain a read lock using `rcu_read_lock();` before calling `find_vpid(pid)` and release it after get its `pid*` structure. This is because we want to read a global structure, it's important to inform the reclaimers that we are entering an RCU read-side critical section. Besides the read lock, we check if the `memory_type` is 1 or 2 or 3. If not, we return -1.

There is no need to add extra arguments. Because `find_vpid` can take all the `int` as input and our input argument `pid` type is `pid_t`, which equals to `int`.

In the `linux-5.8.1/arch/x86/entry/syscalls/syscall_64.tbl`, we add

```
@@ -360,6 +360,7 @@
 437 common openat2      sys_openat2
 438 common pidfd_getfd  sys_pidfd_getfd
 439 common faccessat2   sys_faccessat2
+440 common memuse_syscall sys_memuse_syscall
```

In the `linux-5.8.1/include/linux/syscalls.h`, we add

```
@@ -1424,4 +1424,6 @@
     unsigned int nsops,
     const struct old_timespec32 __user *timeout);

+asmlinkage long sys_memuse_syscall(pid_t pid, int memory_type);
+
#endif
```

In the `linux-5.8.1/Makefile`, we add

```
@@ -1070,7 +1070,7 @@
 export MODULES_NSDEPS := $(extmod-prefix)modules.nsdeps

 ifeq ($(KBUILD_EXTMOD),)
-core-y += kernel/ certs/ mm/ fs/ ipc/ security/ crypto/ block/
+core-y += kernel/ certs/ mm/ fs/ ipc/ security/ crypto/ block/ memuse/
```

We create a directory `linux-5.8.1/memuse/`, and create two files in this directory: `memuse.c` and `Makefile`

We write the following code in the `memuse.c`

```

#include <linux/kernel.h>
#include <linux/syscalls.h>
#include <linux/pid.h>

SYSCALL_DEFINE2(memuse_syscall, pid_t, pid, int, memory_type){
    rcu_read_lock();
    struct pid* res = find_vpid(pid);
    rcu_read_unlock();
    printk("pid:%d, memory_type:%d\n", pid, memory_type);
    if((memory_type > 3) || (memory_type < 1))
        return -1;
    return (res == NULL ? -1 : 1);
}

```

Write the following code in the `Makefile`

```

obj-y := memuse.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

```

Detail:

Run `sudo uname --m`, return `x86_64`. So we are using a **x86_64** system.

We change the files in the kernel source codes as illustrated above. Or apply the patch to the kernel source codes.

Note: Our kernel version is `linux-5.8.1`.

1. Configure the kernel.

Then we open the configuration window with the following command.

```
make menuconfig
```

2. Compile the kernel, because we only allocate 4 cores for the ubuntu VM.

```
make -j4
```

3. Install the kernel and reboot.

```
sudo make modules_install install -j4
reboot
```

4. Then we write an user test program to call the memuse syscall. Create a directory `mkdir user` with two files `user.c` and `CMakeLists.txt` in it.

Write the following codes in the `user.c`

```
#include <stdio.h>
#include <linux/kernel.h>
#include <sys/syscall.h>
#include <unistd.h>
int main()
{
    pid_t pid[10] = {0, 1, 10, 132, 1334, 111111};
    int memory_type[10] = {0, 1, 2, 2, 1, 2, 0};
    for (size_t i = 0; i < 6; i++)
    {
        long int res = syscall(440, pid[i], memory_type[i]);
        printf("System call memuse returned %ld\n", res);
    }
    return 0;
}
```

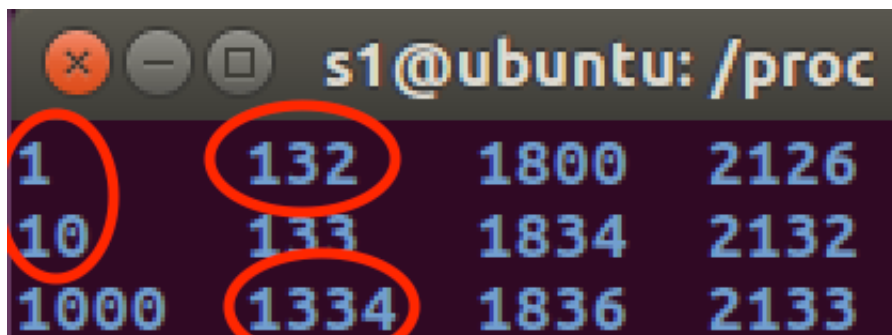
Write the following codes in the `CMakeLists.txt`

```
cmake_minimum_required(VERSION 3.5)
project(test)

set(CMAKE_CXX_STANDARD 14)

add_executable(a.out user.c)
```

Look at the current running processes `ls /proc`

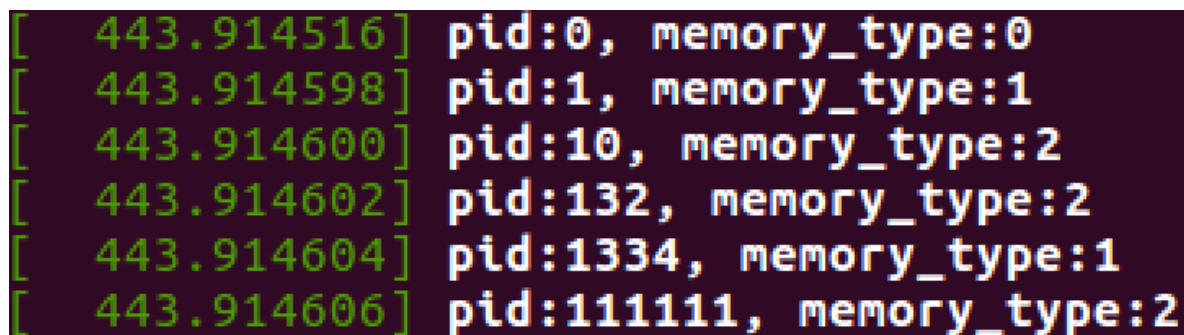


Then we run our user program to see if our syscall works. We test 6 pids: 0, 1, 10, 132, 1334, 111111. The results are as we assumed.

```
s1@ubuntu:~/Desktop/part2/user$ cmake .
s1@ubuntu:~/Desktop/part2/user$ make
s1@ubuntu:~/Desktop/part2/user$ ./a.out
System call memuse returned -1
System call memuse returned 1
System call memuse returned 1
System call memuse returned 1
System call memuse returned 1
System call memuse returned -1
```

Check the output of `dmesg`.

At the bottom, we see the following.



```
[ 443.914516] pid:0, memory_type:0
[ 443.914598] pid:1, memory_type:1
[ 443.914600] pid:10, memory_type:2
[ 443.914602] pid:132, memory_type:2
[ 443.914604] pid:1334, memory_type:1
[ 443.914606] pid:111111, memory_type:2
```

ref:

<https://dev.to/jasper/adding-a-system-call-to-the-linux-kernel-5-8-1-in-ubuntu-20-04-lts-2ga8>

<https://kernelnewbies.org/FAQ/asmlinkage>

part 3

Overall:

What is the limit on the number of parameters a system call can accept? Why is it that number, i.e., what is the limiting factor? How could you add more parameters if you needed to pass more arguments to a system call?

There are only six parameters a system call can accept.

Because there are only six registers available for the parameters.

From the users' view, we can compact multiple parameters into a class and use that class to pass more arguments. From the system's view, we can use stack to pass more arguments.

Detail:

To implement our own `syscall()` invocation routine.

1. implement `mysyscall`

We create a new file `mysyscall.h` under the directory `user`, write the following codes in it.

```
#include <stdarg.h>
#include <errno.h>

long int mysyscall(int number, int arg1, int arg2){
    long resultvar=0;
    asm volatile (
        "movl %1, %%eax\n\t"
        "movl %2, %%edi\n\t"
        "movl %3, %%esi\n\t"
        "syscall\n\t"
        : "=a" (resultvar)
        : "r" (number), "r" (arg1), "r" (arg2)
    );
    return (long int) resultvar;
}
```

and update the `user.c`, replace the `syscall` with `mysyscall`

```
+ include "mysyscall.h"

- long int res = syscall(440, pid[i], memory_type[i]);
+ long int res = mysyscall(440, pid[i], memory_type[i]);
```

and update the `CMakeLists.txt`

```
- add_executable(a.out user.c)
+ add_executable(a.out user.c mysyscall.h)
```

2. Test our user program and check the results.


```
s1@ubuntu:~/Desktop/part2/user$ cmake .
s1@ubuntu:~/Desktop/part2/user$ make
s1@ubuntu:~/Desktop/part2/user$ ./a.out
System call memuse returned -1
System call memuse returned 1
System call memuse returned 1
System call memuse returned 1
System call memuse returned 1
System call memuse returned -1
```

ref:

https://en.wikibooks.org/wiki/X86_Assembly/Interfacing_with_Linux

<https://web.stanford.edu/class/archive/cs/cs107/cs107.1186/guide/x86-64.html>

Part 4

Overall:

Apply the `part4.patch` to the kernel source code, recompile and rebuild the kernel. After rebooting, we can run the user program to test our `memuse_syscall` with the following commands.

```
cd user
cmake .
make
./a.out
```

The results are as follows:

```
s1@ubuntu:~/Desktop/lab1/user$ ./a.out
user stack memory cost: 12288
kernel stack memory cost: 16384
heap memory cost: 4096

Heap tests start:
malloc 16 bytes.
heap memory cost: 4096
malloc 4096 bytes.
heap memory cost: 8192
Heap tests end.

Kernel stack tests start:
create 5 threads.
kernel stack memory cost: 98304
Kernel stack tests end.

User stack tests start:
Recursive function stops until depth == 512.
user stack memory cost: 28672
User stack tests end.
```

Detail:

To measure the physical memory used in the certain specified type of a certain process, we should first know how much virtual memory it uses and how much of them are indeed mapped to physical memory. In a Linux system, every user process has 2 stacks, a user stack and a dedicated kernel stack for the process. The user stack resides in the user address space (first 3GB in 32-bit x86 arch.) and the kernel stack resides in the kernel address space (3GB-4GB in 32bit-x86) of the process. The user stack and heap's structure are different from the kernel stack. Besides, the page table is also different. Therefore, we first focus on the user stack and heap.

User stack and heap

1. Find out the virtual memory.

```

// no need for lock cause it already has a rcu read lock inside
struct task_struct *ts = find_get_task_by_vpid(pid);
if(ts == NULL){
    printk("Process %d doesn't exist!\n", pid);
    return -1;
}

struct mm_struct *base_mm = ts->mm;
if(base_mm == NULL){ // It's a kernel process.
    printk("Process %d is a kernel process!\n", pid);
    return -1;
}

```

We first get the task_struct of the current process with `find_get_task_by_vpid(pid)`. Because `find_get_task_by_vpid` internally has a rcu read lock, we don't need to get an extra lock. All the user memory information are stored in the `mm_struct`. Also, `find_get_task_by_vpid` calls `get_task_struct` to maintain a read lock for the `task_struct`, which should be released at the end of our syscall.

To find out the virtual memory of the user stack and heap, we look into the `mm_struct`.

`base_mm->start_stack`: the start address of the user stack

`base_mm->start_brk`: the start address of the user heap

`base_mm->brk`: the end address of the user heap

Each individual segments of the user virtual memory space are maintained by Virtual Memory Area(VMA). We find the corresponding VMA with the following statements.

```

// find user stack VMA
unsigned long start_process_stack = base_mm->start_stack;
struct vm_area_struct *stack_vma = find_vma(base_mm, start_process_stack);

```

```

// find user heap VMA
unsigned long start_process_heap = base_mm->start_brk;
unsigned long end_process_heap = base_mm->brk;
struct vm_area_struct *heap_vma = find_vma(base_mm, start_process_heap);

```

Each VMA has a start address `vm_start` and a end address `vm_end`. The two addresses profile the virtual memory space of the segment.

One side question: Why `mm_struct->start_stack` and `vm_area_struct->vm_start` don't point to the same address? It looks like `mm_struct->start_stack` is the initial stack pointer address and `vm_area_struct->vm_start` is the main thread stack pointer address.

Note: we are supposed to protect the `find_vma` with the `mmap_read_lock()`.

This is because `find_vma()` makes use of, and updates, the `mmap_cache` pointer hint. The update of `mmap_cache` is racy (page stealer can race with other code that invokes `find_vma` with `mmap_sem` held), but that is okay, since it is a hint. This can be fixed, if desired, by having `find_vma` grab the `page_table_lock`.

2. Map virtual memory to physical memory

We iterate over the virtual memory space of the `stack_vma` and `heap_vma` to find out how much pages are actually mapped to physical memory.

Currently linux x86_64 has **five layers** memory map architecture: `pgd`, `p4d`, `pud`, `pmd`, `pte`

```
bool va_to_pa(struct mm_struct *mm, unsigned long va){
    pgd_t *pgd = pgd_offset(mm, va);
    if(pgd_present(*pgd)){
        p4d_t *p4d = p4d_offset(pgd, va);
        if(p4d_present(*p4d)){
            pud_t *pud = pud_offset(p4d, va);
            if (pud_present(*pud)){
                pmd_t *pmd = pmd_offset(pud, va);
                if(pmd_present(*pmd)){
                    pte_t *pte = pte_offset_kernel(pmd,
va);

                    if(pte_present(*pte)){
                        return 1;
                    }
                }
            }
        }
    }
    return 0;
}
```

We use the above function to determine if a virtual address is mapped to a physical page.

```
for (i = stack_start; i <= stack_end; i += PAGE_SIZE){
    if(va_to_pa(base_mm, i)){
        stack_size += PAGE_SIZE;
    }
}
```

```
for (i = heap_start; i <= heap_end; i += PAGE_SIZE){
    if(va_to_pa(base_mm, i)){
        stack_size += PAGE_SIZE;
    }
}
```

We iterate over the whole stack virtual memory and heap virtual memory to measure the physical memory costed by each segment.

Kernel stack

The kernel stack is different from the user stack because each thread has a kernel stack and all the kernel stacks are stored in the kernel space, while in the user space, only main thread stack are stored in the stack segment and all other stacks are stored in the heap.

We iterate all the threads with `next_thread` to get all the threads' `task_struct`.

The kernel stack start address is stored in the `task_struct->stack`.

Each kernel stack is no larger than `THREAD_SIZE (2*PAGE_SIZE)`.

To determine whether a address is mapped to physical memory, we can use `kern_addr_valid`, which returns `True` iff the address is mapped to physical memory.

Combing all the above, we iterate all the virtual pages in each kernel stack and measure the physical memory used by the kernel stacks.

```
struct task_struct *tmp = ts;
unsigned long thread_num = 0;
do{
    void *tmp_kernel_stack = tmp->stack;
    for (i = tmp_kernel_stack; i <= tmp_kernel_stack+THREAD_SIZE; i += PAGE_SIZE)
    {
        if(kern_addr_valid(i)){
            kernel_stack_size += PAGE_SIZE;
        }
    }
    rcu_read_lock();
    put_task_struct(tmp); // release the lock of the last task struct
    tmp = next_thread(tmp);
    get_task_struct(tmp); // get the lock of the current task struct
    rcu_read_unlock();
    thread_num++;
} while (tmp != ts); // traverse the rcu loop
```

We should always maintain a read lock for our current `task_struct` explicitly calling `get_task_struct`. Also when calling the `next_thread`, we should maintain a `rcu_read_lock`.

User program

In our user test program, we test the correctness of the user stack, kernel stack and user heap respectively.

```
s1@ubuntu:~/Desktop/lab1/user$ ./a.out
user stack memory cost: 12288
kernel stack memory cost: 16384
heap memory cost: 4096

Heap tests start:
malloc 16 bytes.
heap memory cost: 4096
malloc 4096 bytes.
heap memory cost: 8192
Heap tests end.

Kernel stack tests start:
create 5 threads.
kernel stack memory cost: 98304
Kernel stack tests end.

User stack tests start:
Recursive function stops until depth == 512.
user stack memory cost: 28672
User stack tests end.
```

We first measure these information without any operations. The first three lines show the results.

User heap

To test the user heap, we call malloc to see the results. When mallocing 16 bytes, the heap still costs 4096 bytes. When mallocing 4096 bytes further, the heap costs 8192 bytes.

```
printf("Heap tests start:\n");
printf("malloc 16 bytes.\n");
char *tmp0 = (char *)malloc(16);
test_syscall(pid, 3);
printf("malloc 4096 bytes.\n");
char *tmp1 = (char *)malloc(4096);
test_syscall(pid, 3);
printf("Heap tests end.\n\n");
```

Kernel stack

To test the kernel stack, we create five threads and measure the kernel stack size. As shown in the results, the kernel stack size increases from 16384 bytes to 98304 bytes.

```

printf("create %d threads.\n", TEST_THREAD_NUM);
pthread_t pt[TEST_THREAD_NUM];
for (size_t i = 0; i < TEST_THREAD_NUM; i++){
    pthread_create(&pt[i], NULL, kernel_stack_test_func, NULL);
}
test_syscall(pid, 2);
for (size_t i = 0; i < TEST_THREAD_NUM; i++){
    pthread_join(pt[i], NULL);
}

```

Here the `kernel_stack_test_func` is very simple, just sleep 1 second and return.

```

void *kernel_stack_test_func(void* args) {
    int i = 1;
    sleep(1);
}

```

User stack

To test user stack, we call a recursive function, which stops until `depth == 512`.

```

void user_stack_test_func(int depth){
    if (depth == 512){
        test_syscall(pid, 1);
        return;
    }
    user_stack_test_func(depth + 1);
}

```

The results show that the stack memory cost increases from `12288` to `28672`.

Correctness double check: To check the correctness of our `memuse_syscall`, we can compare the output of `dmesg` with the output of `cat /proc/$pid/maps`, which shows the same results.

```

[ 5540.136928] pid:4070, memory_type:1
[ 5540.136931] stack start addr: 140727235129344, stack end addr: 140727235264512
[ 5540.136931] stack size: 12288
[ 5540.137012] pid:4070, memory_type:2
[ 5540.137013] kernel stack size: 16384, thread number: 1
[ 5540.137017] pid:4070, memory_type:3
[ 5540.137018] heap start addr: 39391232, heap end addr: 39526400
[ 5540.137019] heap size: 4096
[ 5540.137026] pid:4070, memory_type:3
[ 5540.137027] heap start addr: 39391232, heap end addr: 39526400
[ 5540.137027] heap size: 4096
[ 5540.137033] pid:4070, memory_type:3
[ 5540.137034] heap start addr: 39391232, heap end addr: 39526400
[ 5540.137034] heap size: 8192
[ 5540.137186] pid:4070, memory_type:2
[ 5540.137188] kernel stack size: 98304, thread number: 6
[ 5541.138007] pid:4070, memory_type:1
[ 5541.138012] stack start addr: 140727235129344, stack end addr: 140727235264512
[ 5541.138013] stack size: 28672

```

```

root@ubuntu:~/Desktop/lab1/user# cat /proc/4070/maps
00400000-00401000 r-xp 00000000 08:01 3542370 /home/s1/Desktop/lab1/user/a.out
00601000-00602000 r--p 00001000 08:01 3542370 /home/s1/Desktop/lab1/user/a.out
00602000-00603000 rw-p 00002000 08:01 3542370 /home/s1/Desktop/lab1/user/a.out
02591000-025b2000 rw-p 00000000 00:00 0 [heap]
7f47c8703000-7f47c8704000 ---p 00000000 00:00 0
7f47c8704000-7f47c8f04000 rw-p 00000000 00:00 0
7f47c8f04000-7f47c8f05000 ---p 00000000 00:00 0
7f47c8f05000-7f47c9705000 rw-p 00000000 00:00 0
7f47c9705000-7f47c9706000 ---p 00000000 00:00 0
7f47c9706000-7f47c9f06000 rw-p 00000000 00:00 0
7f47c9f06000-7f47c9f07000 ---p 00000000 00:00 0
7f47c9f07000-7f47ca707000 rw-p 00000000 00:00 0
7f47caf08000-7f47cb0c8000 r-xp 00000000 08:01 1323197 /lib/x86_64-linux-gnu/libc-2.23.so
7f47cb0c8000-7f47cb2c8000 ---p 001c0000 08:01 1323197 /lib/x86_64-linux-gnu/libc-2.23.so
7f47cb2c8000-7f47cb2cc000 r--p 001c0000 08:01 1323197 /lib/x86_64-linux-gnu/libc-2.23.so
7f47cb2cc000-7f47cb2ce000 rw-p 001c4000 08:01 1323197 /lib/x86_64-linux-gnu/libc-2.23.so
7f47cb2ce000-7f47cb2d2000 rw-p 00000000 00:00 0
7f47cb2d2000-7f47cb2ea000 r-xp 00000000 08:01 1323179 /lib/x86_64-linux-gnu/libpthread-2.23.so
7f47cb2ea000-7f47cb4e9000 ---p 00018000 08:01 1323179 /lib/x86_64-linux-gnu/libpthread-2.23.so
7f47cb4e9000-7f47cb4ea000 r--p 00017000 08:01 1323179 /lib/x86_64-linux-gnu/libpthread-2.23.so
7f47cb4ea000-7f47cb4eb000 rw-p 00018000 08:01 1323179 /lib/x86_64-linux-gnu/libpthread-2.23.so
7f47cb4eb000-7f47cb4ef000 rw-p 00000000 00:00 0
7f47cb4ef000-7f47cb515000 r-xp 00000000 08:01 1323189 /lib/x86_64-linux-gnu/ld-2.23.so
7f47cb6fa000-7f47cb6fe000 rw-p 00000000 00:00 0
7f47cb714000-7f47cb715000 r--p 00025000 08:01 1323189 /lib/x86_64-linux-gnu/ld-2.23.so
7f47cb715000-7f47cb716000 rw-p 00026000 08:01 1323189 /lib/x86_64-linux-gnu/ld-2.23.so
7f47cb716000-7f47cb717000 rw-p 00000000 00:00 0
7ffd9cdc3000-7ffd9cde4000 rw-p 00000000 00:00 0 [stack]
7ffd9cdec000-7ffd9cdf0000 r--p 00000000 00:00 0 [vvar]
7ffd9cdf0000-7ffd9cdf2000 r-xp 00000000 00:00 0 [vdso]
fffffffff600000-fffffffff601000 r-xp 00000000 00:00 0 [vsyscall]

```

Experience:

For part1-3:

Personally, the most difficult part of this project is debugging. Every time we do slightly modification to the codes, we need to reinstall the kernel and reboot to check if my codes run correctly. Though with incremental make and cache, every reinstall takes at least five to ten minutes and sometimes even half an hour. Therefore, it takes me more than one day to finally debug a typo in the Makefile.

The second difficult part is to make clear what we need to do. Though with massive tutorials on Google, it takes me some time to find the suitable one.

For part 4:

Actually, part4 seems easier than the first three ones when I gradually get used to the kernel programming and how to get most out of [bootlin](#). The most difficult part is to make it clear which variable should be protected by a lock and which lock I should use.

Though I knew theoretically how the virtual memory works before the lab, I have never looked into the code and this lab guides me to know how linux maintain all the information in the `mm_struct` and how to perform MMU in software, which I always assume it to be performed by hardware.

Overall, this lab does enhance my understanding linux system and correct some of my misunderstanding.