جامعة خليفة
Khalifa University

ECCE 632: Advanced Operating Systems
Assignment

System Calls

Group Members:

Ghebrebrhan Weldit	100043383
Zhang Zhibo	100060990
Mohamed Tarek	100038660

Submitted to:
Dr Ernesto Damiani
Spring 2022

1 Introduction

1.1 System Calls

System calls provide an interface for user programs to access the services made available by the operating system. Application developers access system calls via an application programming interface (API). The API specifies a set of C functions that are available to the application programmer, including the parameters that are passed to each function and the return values the programmer can expect. The functions that make up the API typically invoke the actual system calls on behalf of the application programmer.

1.1.1 Example of Standard API

As an example of the standard POSIX API, consider the open() function that is available in UNIX and Linux systems. The API for this function is obtained from the man page. A description of this API appears below:

```
$ man open

#include <unistd.h>

int open(const char *path, int oflags);
int open(const char *path, int oflags, mode_t mode);
```

A program that uses the open() function must include the unistd.h header file, as this file defines int data types (among other things). The parameters passed to open() are as follows.

• const *path - The relative or absolute path to the file that is to be opened.

• int oflags - A bitwise 'or' separated list of values that determine the method in which the file is to be opened.

• mode t mode - A bitwise 'or' separated list of values that determine the permissions of the file if it is created.

The file descriptor returned is the integer that is positive or zero if the file was opened without problems. If a negative value is returned, then there was an error occur while opening the file.

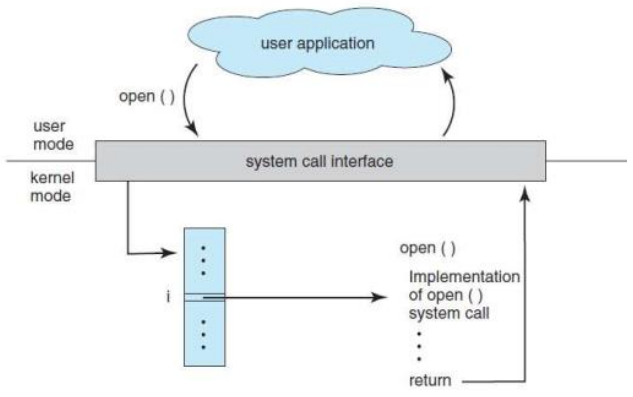


Figure 1: A user application invoking the open() system call.

The relationship between an API, the system-call interface, and the operating system is shown in Figure 1, which illustrates how the operating system handles a user application invoking the open() system call.

1.1.2 Diagram

Figure 2 shows, the diagram of doing the assignment. First, you will practice the progress of compiling Linux kernel. After that, the most important part is to implement a system call inside the kernel. The assignment is divided into multiple stages and score is marked by each stage as Figure 2.

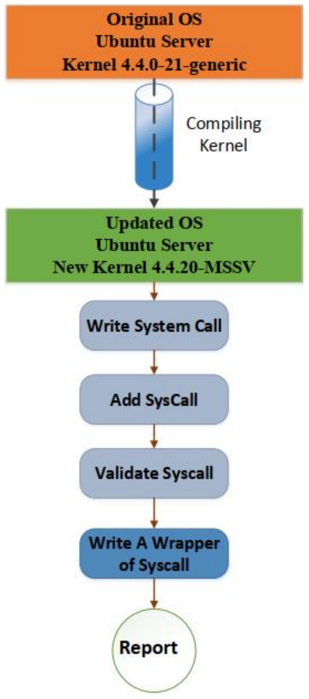


Figure 2: Diagram of implementing the assignment

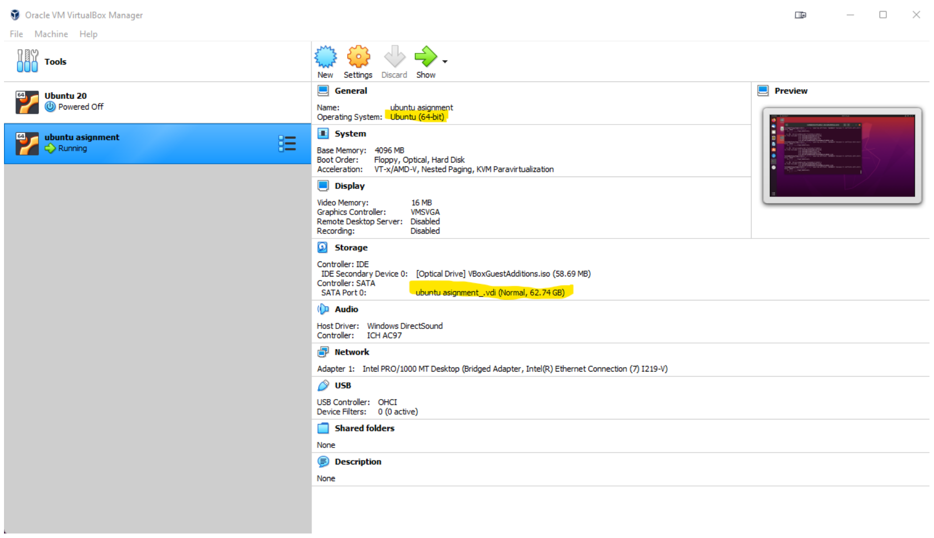
2 Prepare Linux Kernel

The main objective of this chapter is to prepare the linux kernel on a virtual machine running Ubuntu 20.04.

2.1 Preparation

2.1.1 Set up Virtual Machine

1. **Oracle VM Virtual Box Manager was used along with Ubuntu 20.4 image.**



Ubuntu 20
Powered Off

ubuntu asignment
Running

General
Name: ubuntu asignment
Operating System: Ubuntu (64-bit)

System
Base Memory: 4096 MB
Boot Order: Floppy, Optical, Hard Disk
Acceleration: VT-x/AMD-V, Nested Paging, KVM Paravirtualization

Display
Video Memory: 16 MB
Graphics Controller: VMSVGA
Remote Desktop Server: Disabled
Recording: Disabled

Storage
Controller: IDE
IDE Secondary Device 0: [Optical Drive] VBoxGuestAdditions.iso (58.69 MB)
Controller: SATA
SATA Port 0: ubuntu asignment_.vdi (Normal, 62.74 GB)

Audio
Host Driver: Windows DirectSound
Controller: ICH AC97

Network
Adapter 1: Intel PRO/1000 MT Desktop (Bridged Adapter, Intel(R) Ethernet Connection (7) I219-V)

USB
USB Controller: OHCI
Device Filters: 0 (0 active)

Shared folders
None

Description
None

Figure 3: Virtual Box Manager Settings

Some settings were modified to allow us the system to run smoothly. First, at least 50 GB of storage memory is required to be allocated for Ubuntu to allow all the required packages to be installed. Secondly, bidirectional drag and drop and clapboarding to allow the host machine to send/share data to the virtual machine. For networking, it is important to set bridge adapter to allow the virtual machine to access the internet to download the required packages.

2.1.2 Install core packages

- Get Ubuntu's toolchain (gcc, make, and so forth) by installing the **build essential** metapackage:

2.1.3

```
$ sudo apt-get update
$ sudo apt-get install build-essential
```

- Install kernel-package:

```
$ sudo apt-get install kernel-package
```

2.1.4 Create a kernel compilation directory:

It is recommended to create a separate build directory for your kernel(s). In this example, the directory kernelbuild will be created in the home directory:

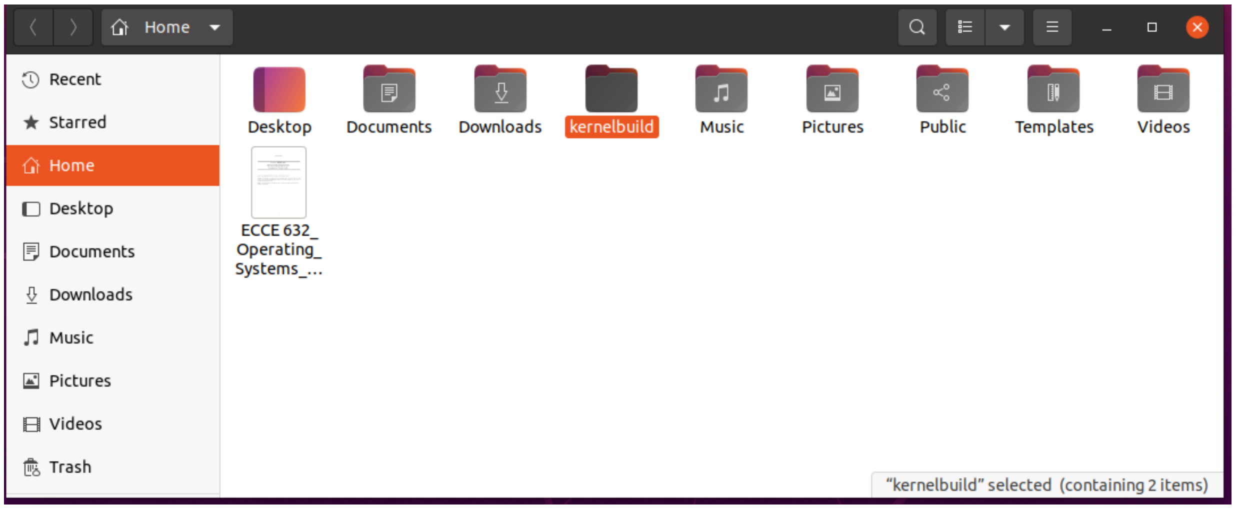


Figure 4: Kernel Directory

2.1.5 Download the kernel source

Download the kernel source from http://www.kernel.org. This should be the tarball (tar.xz) file for your chosen kernel. It can be downloaded by simply right-clicking the tar.xz link in your browser and selecting Save Link As..., or any other number of ways via alternative graphical or command-line tools that utilize HTTP, FTP, RSYNC, or Git. In the following command-line example, wget has been installed and is used inside the ˜ /kernelbuild directory to obtain kernel 4.4.56.

```
$ cd ˜/kernelbuild
$ wget https://cdn.kernel.org/pub/linux/kernel/v4.x/linux-4.4.56.tar.xz
```

QUESTION: Why we have to use another kernel source from the server such as http://www.kernel.org, can we compile the original kernel (the local kernel on the running OS) directly?

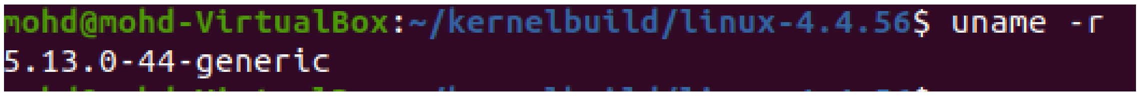
We can't compile the local operating system because its already compiled. The local operating system is in a binary format and couldn't be compiled. I need to download another copy, source format, then I need to build the binary of the source code in each directory on my disk.

2.2 Configuration

A crucial step in customizing the default kernel to reflect your computer's precise specifications. Kernel configuration is its .config file, which includes the use of Kernel modules. By setting the options in .config properly, your kernel and computer will function most efficiently. Since making our own configuration file is a complicated process, we could borrow the content of configuration file of an existing kernel currently used by the virtual machine. This file is typically located in /boot/ so our job is simply copy it to the source code directory:

```
$ cp /boot/config-x.x.x-x-generic ˜/kernelbuild/linux-4.4.56/.config
```

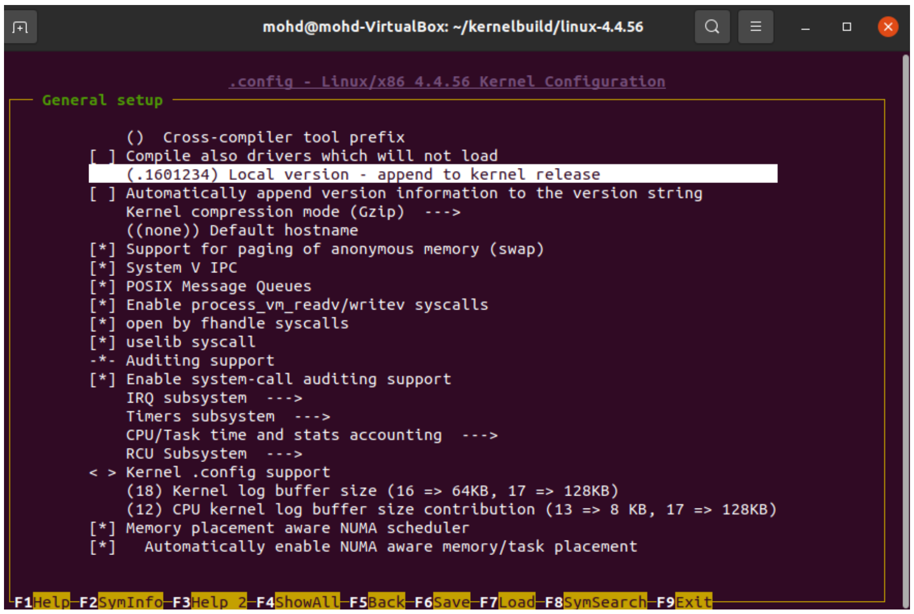
Using the command uname -r we can find the specific version of kernel installed.



```
mohd@mohd-VirtualBox:~/kernelbuild/linux-4.4.56$ uname -r
5.13.0-44-generic
```

run $ make menuconfig or $ make nconfig inside the top directory to open Kernel Configuration.

To change kernel version, go to General setup option, Access to the line "(-ARCH) Local version append to kernel release". Then enter a dot "." followed by your MSSV. For example: .1601234 Press F6 to save your change and then press F9 to exit.



```
mohd@mohd-VirtualBox: ~/kernelbuild/linux-4.4.56

.config - Linux/x86 4.4.56 Kernel Configuration
General setup
    ()  Cross-compiler tool prefix
[ ] Compile also drivers which will not load
    (.1601234) Local version - append to kernel release
[ ] Automatically append version information to the version string
    Kernel compression mode (Gzip)  --->
    ((none)) Default hostname
[*] Support for paging of anonymous memory (swap)
[*] System V IPC
[*] POSIX Message Queues
[*] Enable process_vm_readv/writev syscalls
[*] open by fhandle syscalls
[*] uselib syscall
-*- Auditing support
[*] Enable system-call auditing support
    IRQ subsystem  --->
    Timers subsystem  --->
    CPU/Task time and stats accounting  --->
    RCU Subsystem  --->
< > Kernel .config support
    (18) Kernel log buffer size (16 => 64KB, 17 => 128KB)
    (12) CPU kernel log buffer size contribution (13 => 8 KB, 17 => 128KB)
[*] Memory placement aware NUMA scheduler
[*]   Automatically enable NUMA aware memory/task placement

F1Help F2SymInfo F3Help 2 F4ShowAll F5Back F6Save F7Load F8SymSearch F9Exit
```

Install relevant compilers:

```
sudo apt-get update
sudo apt-get install libncurses5-dev libssl-dev
sudo apt-get install build-essential openssl
sudo apt-get install zlibc minizip
sudo apt-get install bison
sudo apt-get install vim
sudo apt-get install flex
sudo apt-get install libssl-dev
sudo apt install dwarves
sudo apt-get install libelf-dev
```

3 Adding a Hello World Sys call

Add the desired system call function

```
sudo vim sys.c
```

(Note: In vim, “I” enters edit, “esc” exits edit. “G” jumps to the end, “gg” goes to the beginning. “:wq” save to exit, “:q” don't save to exit)

Add the following function in this file:

```
asmlinkage long sys_helloworld(void)
{
   printk( "helloworld!");
    return 1;
}
```

Add the system call number:

```
 439   64   helloworld          sys_helloworld
```

```
cd  /usr/src/linux-5.6
sudo make mrproper
sudo make clean

sudo make menuconfig
```

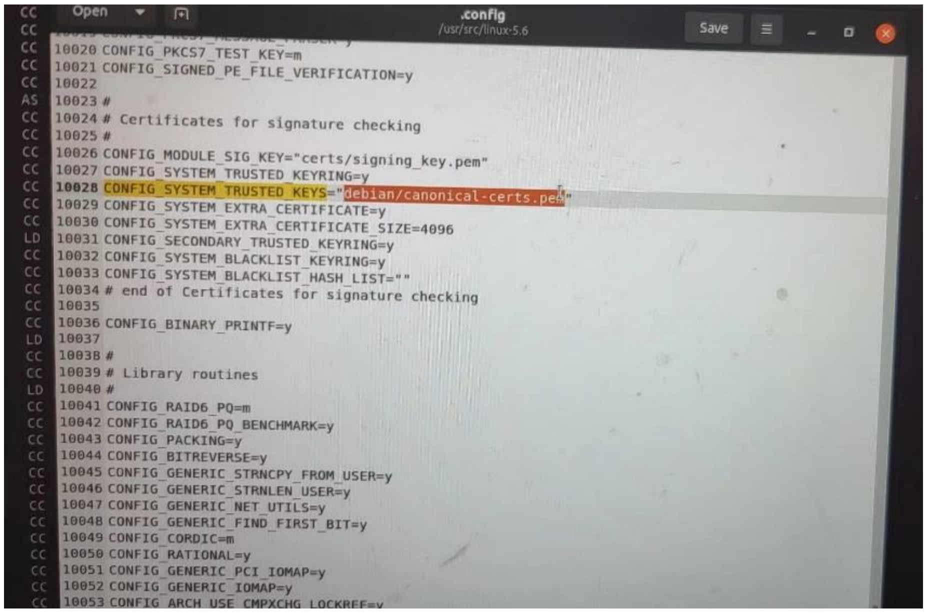
since there is a runtime error, we had to troubleshoot by using `sudo gedit .config`

Find:

```
CONFIG_SYSTEM_TRUSTED_KEYS="debian/canonical-certs.pem"
```

Delete:

```
debian/canonical-certs.pem
```



```
CONFIG_PKCS7_TEST_KEY=m
CONFIG_SIGNED_PE_FILE_VERIFICATION=y

#
# Certificates for signature checking
#
CONFIG_MODULE_SIG_KEY="certs/signing_key.pem"
CONFIG_SYSTEM_TRUSTED_KEYRING=y
CONFIG_SYSTEM_TRUSTED_KEYS="debian/canonical-certs.pem"
CONFIG_SYSTEM_EXTRA_CERTIFICATE=y
CONFIG_SYSTEM_EXTRA_CERTIFICATE_SIZE=4096
CONFIG_SECONDARY_TRUSTED_KEYRING=y
CONFIG_SYSTEM_BLACKLIST_KEYRING=y
CONFIG_SYSTEM_BLACKLIST_HASH_LIST=""
# end of Certificates for signature checking

CONFIG_BINARY_PRINTF=y

#
# Library routines
#
CONFIG_RAID6_PQ=m
CONFIG_RAID6_PQ_BENCHMARK=y
CONFIG_PACKING=y
CONFIG_BITREVERSE=y
CONFIG_GENERIC_STRNCPY_FROM_USER=y
CONFIG_GENERIC_STRNLEN_USER=y
CONFIG_GENERIC_NET_UTILS=y
CONFIG_GENERIC_FIND_FIRST_BIT=y
CONFIG_CORDIC=m
CONFIG_RATIONAL=y
CONFIG_GENERIC_PCI_IOMAP=y
CONFIG_GENERIC_IOMAP=y
CONFIG_ARCH_USE_CMPXCHG_LOCKREF=y
```

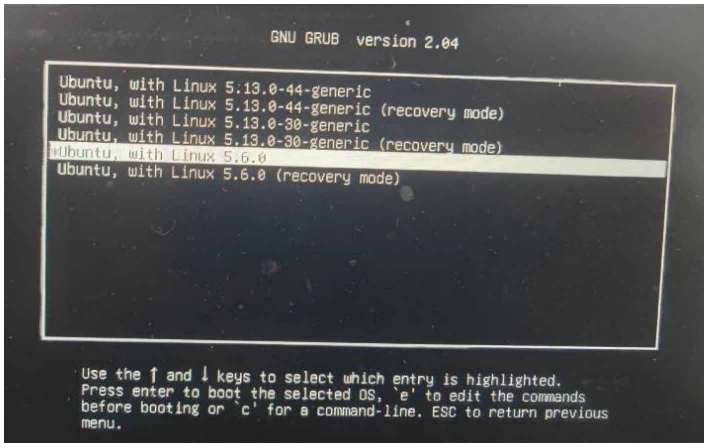
3.1 Compiling

sudo make -j4 (to use 4 cores)
(The terminal window is maximized when performing the steps in this block, otherwise an error may be reported)
It will take 2-3 hours.

sudo make modules_install

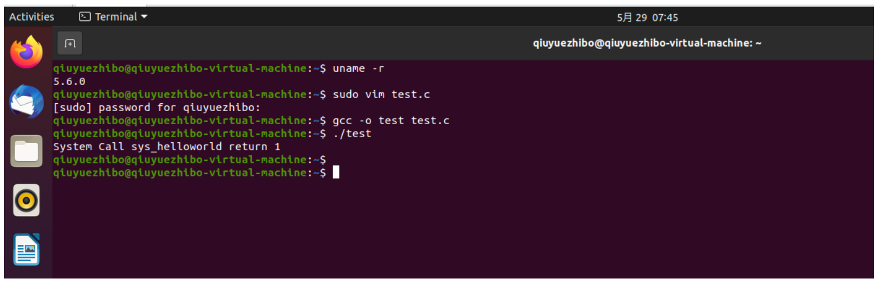
sudo make install

sudo reboot
(After rebooting, click the mouse to enter ubuntu and quickly hold down shift and long press)
Choose the linux version.

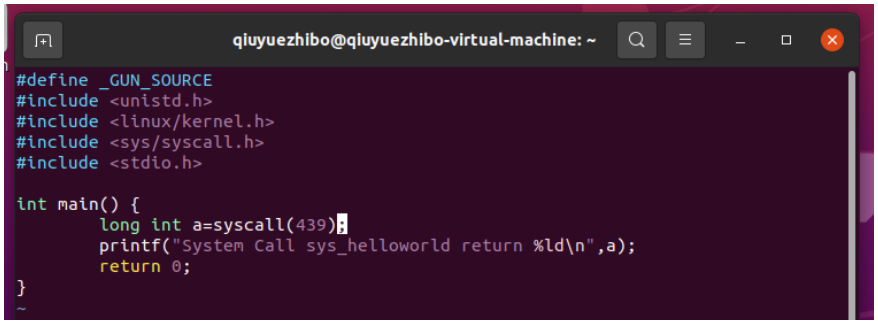


```
GNU GRUB  version 2.04

Ubuntu, with Linux 5.13.0-44-generic
Ubuntu, with Linux 5.13.0-44-generic (recovery mode)
Ubuntu, with Linux 5.13.0-30-generic
Ubuntu, with Linux 5.13.0-30-generic (recovery mode)
*Ubuntu, with Linux 5.6.0
Ubuntu, with Linux 5.6.0 (recovery mode)

Use the ↑ and ↓ keys to select which entry is highlighted.
Press enter to boot the selected OS, `e' to edit the commands
before booting or `c' for a command-line. ESC to return previous
menu.
```

Test by opening the terminal and running command uname -r

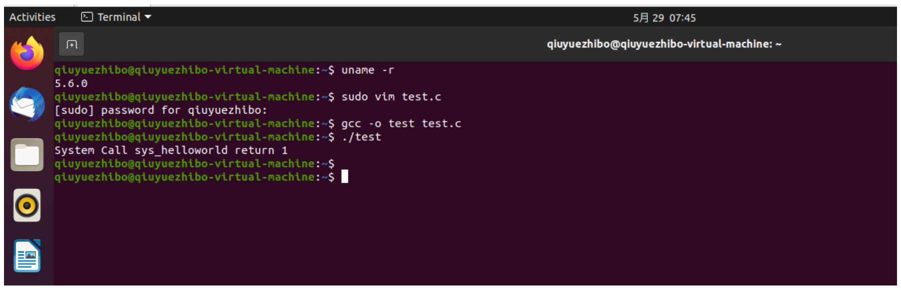


```
qiuyuezhibo@qiuyuezhibo-virtual-machine:~$ uname -r
5.6.0
qiuyuezhibo@qiuyuezhibo-virtual-machine:~$ sudo vim test.c
[sudo] password for qiuyuezhibo:
qiuyuezhibo@qiuyuezhibo-virtual-machine:~$ gcc -o test test.c
qiuyuezhibo@qiuyuezhibo-virtual-machine:~$ ./test
System Call sys_helloworld return 1
qiuyuezhibo@qiuyuezhibo-virtual-machine:~$
qiuyuezhibo@qiuyuezhibo-virtual-machine:~$
```



```c
#define _GUN_SOURCE
#include <unistd.h>
#include <linux/kernel.h>
#include <sys/syscall.h>
#include <stdio.h>

int main() {
        long int a=syscall(439);
        printf("System Call sys_helloworld return %ld\n",a);
        return 0;
}
~
```

Run the command gcc -o test test.c



```
qiuyuezhibo@qiuyuezhibo-virtual-machine:~$ uname -r
5.6.0
qiuyuezhibo@qiuyuezhibo-virtual-machine:~$ sudo vim test.c
[sudo] password for qiuyuezhibo:
qiuyuezhibo@qiuyuezhibo-virtual-machine:~$ gcc -o test test.c
qiuyuezhibo@qiuyuezhibo-virtual-machine:~$ ./test
System Call sys_helloworld return 1
qiuyuezhibo@qiuyuezhibo-virtual-machine:~$
qiuyuezhibo@qiuyuezhibo-virtual-machine:~$
```

Success.

4 System Call – procsched

In this assignment, we have to define a system call named ‘procsched’. This syscall helps user to show the schedule information of a specific process. An example output:

```
pcount: 4567
run_delay: 829035007
last_arrival: 1818402352768
last_queued: 0
```

To implement this system call, you have to use 2 data structures defined by Linux OS, task struct and sched info. In the Linux kernel, every process has an associated struct task struct. The definition of this struct is in the header file include /linux/sched.h

```
struct task_struct {
     volatile long state;
     /* -1 unrunnable, 0 runnable, >0 stopped */
     struct thread_info *thread_info;
     atomic_t usage;
     ...
     ...
     struct sched_info sched_info;
     ...
     char comm[16];
     ...
};
```

The sched_info within the task_struct describe schedule information of the process. The sched_info is defined in include /linux/sched.h as:

```
struct sched_info {
#ifdef CONFIG_SCHED_INFO
     /* # of times we have run on this CPU: */
     unsigned long           pcount;
     /* Time spent waiting on a runqueue: */
     unsigned long long      run_delay;
     /* When did we last run on a CPU? */
     unsigned long long      last_arrival;
     /* When were we last queued to run? */
     unsigned long long      last_queued;
#endif /* CONFIG_SCHED_INFO */
};
```

Kernel modules must have at least two functions: a “start” (initialization) function called init module() which is called when the module is insmoded into the kernel, and an “end” (cleanup) function called cleanup module() which is called just before it is rmmoded.

QUESTION: What is the meaning of other parts, i.e., i386, Procsched, and sys Procsched?

I386: ABI (Application Binary Interface)

Procsched: none of the syscall

Sys_procsched: entry point.

4.1 Kernel Module

Add the desired system call function

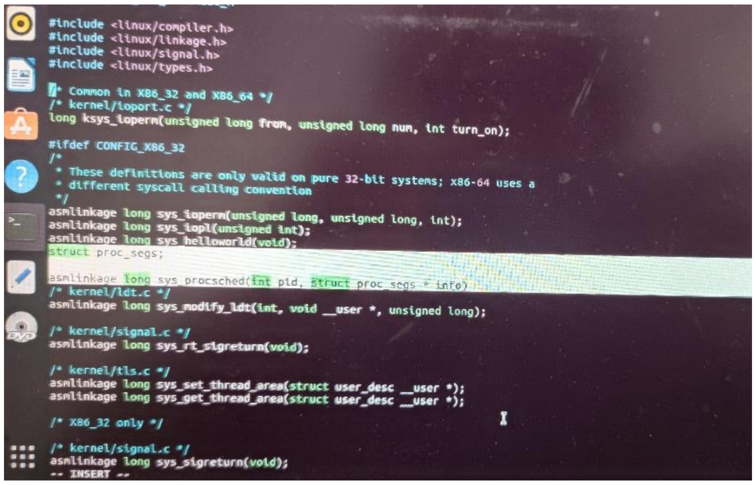
```
sudo vim sys.c
```

Add following codes: (which are basically the codes in "sys_procsched.c" file.

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

struct proc_segs {
 unsigned long mssv;
 unsigned long pcount;
    unsigned long long run_delay;
    unsigned long long last_arrival;
    unsigned long long last_queued;
};

asmlinkage long sys_procsched(int pid, struct proc_segs * info){
 struct task_struct * task;
 printk("Finding...\n");
 for_each_process(task) {
         printk("[%d] ------- [%s]\n", task->pid, task->comm);
         if(task->pid == pid) {
                 if(task->mm != NULL) {
                         struct proc_segs buff;
                         printk("inside process!\n");
                         buff.mssv = 1601234;
                         buff.pcount = task->sched_info.pcount;
                         buff.run_delay = task->mm->run_delay;
                         buff.last_arrival = task->mm->last_arrival;
                         buff.last_queued = task->mm->last_queued;
                         int res = copy_to_user(info, &buff, sizeof(buff));
                         if(res == 0) printk("copy data successful!\n");
                         else printk("copy data failed\n");
                         printk("Find out pid [%d]", pid);
                         return 0;
                 }
         }
 }
 return -1;
}
```

4.1.1 Add the statement

```
cd /usr/src/linux-5.6/arch/x86/include/asm/
sudo vim syscalls.h
```



```
#include <linux/compiler.h>
#include <linux/linkage.h>
#include <linux/signal.h>
#include <linux/types.h>

/* Common in X86_32 and X86_64 */
/* kernel/ioport.c */
long ksys_ioperm(unsigned long from, unsigned long num, int turn_on);

#ifdef CONFIG_X86_32
/*
 * These definitions are only valid on pure 32-bit systems; x86-64 uses a
 * different syscall calling convention
 */
asmlinkage long sys_ioperm(unsigned long, unsigned long, int);
asmlinkage long sys_iopl(unsigned int);
asmlinkage long sys_helloworld(void);
struct proc_segs;

asmlinkage long sys_procsched(int pid, struct proc_segs * info)
/* kernel/ldt.c */
asmlinkage long sys_modify_ldt(int, void __user *, unsigned long);

/* kernel/signal.c */
asmlinkage long sys_rt_sigreturn(void);

/* kernel/tls.c */
asmlinkage long sys_set_thread_area(struct user_desc __user *);
asmlinkage long sys_get_thread_area(struct user_desc __user *);

/* X86_32 only */

/* kernel/signal.c */
asmlinkage long sys_sigreturn(void);
-- INSERT --
```

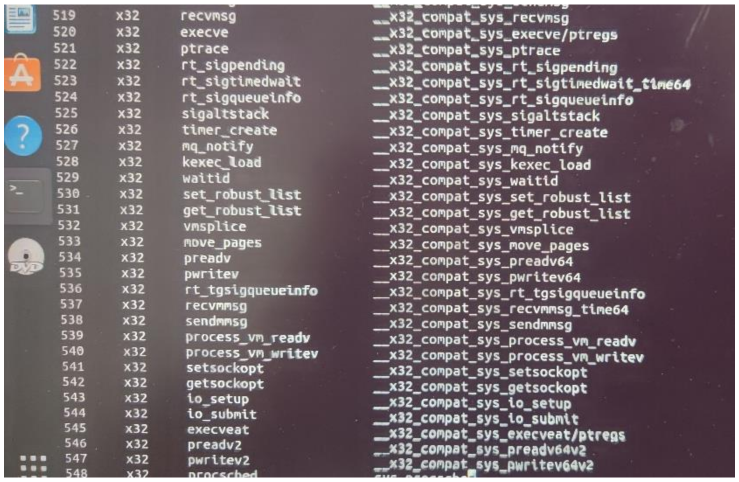
Add:

struct proc_segs;

asmlinkage long sys_procsched(int pid, struct proc_segs * info)

4.1.2 Add system call number

```
cd /usr/src/linux-5.6/arch/x86/entry/syscalls
sudo vim syscall_64.tbl
```



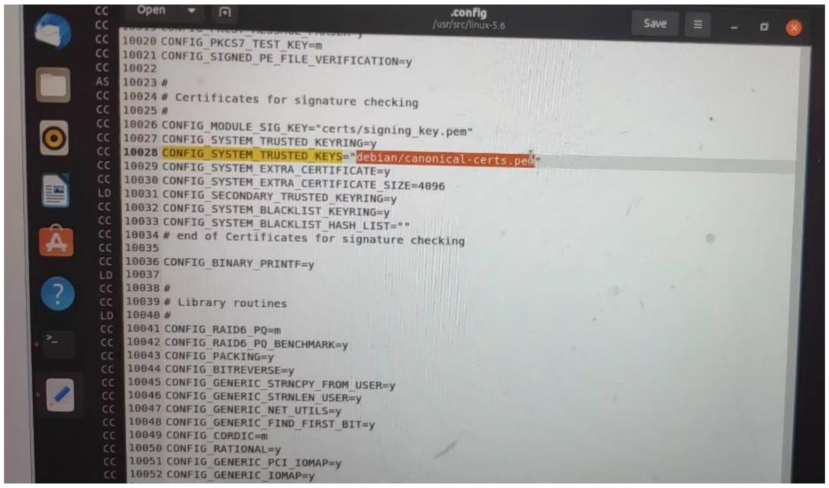
```
519     x32     recvmsg                 __x32_compat_sys_recvmsg
520     x32     execve                  __x32_compat_sys_execve/ptregs
521     x32     ptrace                  __x32_compat_sys_ptrace
522     x32     rt_sigpending           __x32_compat_sys_rt_sigpending
523     x32     rt_sigtimedwait         __x32_compat_sys_rt_sigtimedwait_time64
524     x32     rt_sigqueueinfo         __x32_compat_sys_rt_sigqueueinfo
525     x32     sigaltstack             __x32_compat_sys_sigaltstack
526     x32     timer_create            __x32_compat_sys_timer_create
527     x32     mq_notify               __x32_compat_sys_mq_notify
528     x32     kexec_load              __x32_compat_sys_kexec_load
529     x32     waitid                  __x32_compat_sys_waitid
530     x32     set_robust_list         __x32_compat_sys_set_robust_list
531     x32     get_robust_list         __x32_compat_sys_get_robust_list
532     x32     vmsplice                __x32_compat_sys_vmsplice
533     x32     move_pages              __x32_compat_sys_move_pages
534     x32     preadv                  __x32_compat_sys_preadv64
535     x32     pwritev                 __x32_compat_sys_pwritev64
536     x32     rt_tgsigqueueinfo       __x32_compat_sys_rt_tgsigqueueinfo
537     x32     recvmmsg                __x32_compat_sys_recvmmsg_time64
538     x32     sendmmsg                __x32_compat_sys_sendmmsg
539     x32     process_vm_readv        __x32_compat_sys_process_vm_readv
540     x32     process_vm_writev       __x32_compat_sys_process_vm_writev
541     x32     setsockopt              __x32_compat_sys_setsockopt
542     x32     getsockopt              __x32_compat_sys_getsockopt
543     x32     io_setup                __x32_compat_sys_io_setup
544     x32     io_submit               __x32_compat_sys_io_submit
545     x32     execveat                __x32_compat_sys_execveat/ptregs
546     x32     preadv2                 __x32_compat_sys_preadv64v2
547     x32     pwritev2                __x32_compat_sys_pwritev64v2
548     x32     procsched
```

```
548     x32     procsched               sys_procsched
```

4.2 Compiling (The same as in adding a helloworld)

```
cd  /usr/src/linux-5.6
sudo make mrproper
sudo make clean
sudo make menuconfig
```

When the window appears use the left and right keyboard keys to control directly save->ok->exit->exit sudo gedit .config



```
CONFIG_PKCS7_TEST_KEY=m
CONFIG_SIGNED_PE_FILE_VERIFICATION=y

#
# Certificates for signature checking
#
CONFIG_MODULE_SIG_KEY="certs/signing_key.pem"
CONFIG_SYSTEM_TRUSTED_KEYRING=y
CONFIG_SYSTEM_TRUSTED_KEYS="debian/canonical-certs.pem"
CONFIG_SYSTEM_EXTRA_CERTIFICATE=y
CONFIG_SYSTEM_EXTRA_CERTIFICATE_SIZE=4096
CONFIG_SECONDARY_TRUSTED_KEYRING=y
CONFIG_SYSTEM_BLACKLIST_KEYRING=y
CONFIG_SYSTEM_BLACKLIST_HASH_LIST=""
# end of Certificates for signature checking

CONFIG_BINARY_PRINTF=y

#
# Library routines
#
CONFIG_RAID6_PQ=m
CONFIG_RAID6_PQ_BENCHMARK=y
CONFIG_PACKING=y
CONFIG_BITREVERSE=y
CONFIG_GENERIC_STRNCPY_FROM_USER=y
CONFIG_GENERIC_STRNLEN_USER=y
CONFIG_GENERIC_NET_UTILS=y
CONFIG_GENERIC_FIND_FIRST_BIT=y
CONFIG_CORDIC=m
CONFIG_RATIONAL=y
CONFIG_GENERIC_PCI_IOMAP=y
CONFIG_GENERIC_IOMAP=y
```

Find:
CONFIG_SYSTEM_TRUSTED_KEYS="debian/canonical-certs.pem"

Delete:
debian/canonical-certs.pem

sudo make -j4

(The terminal window is maximised when performing the steps in this block, otherwise an error may be reported)
It will take 2-3 hours.

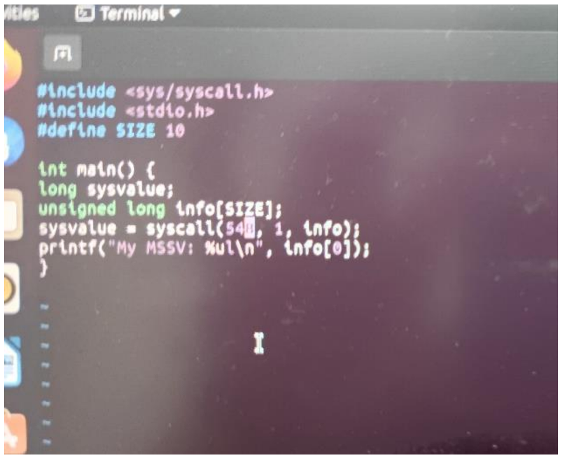
sudo make modules_install
sudo make install
sudo reboot
(After rebooting, click the mouse to enter ubuntu and quickly hold down shift and long press)

4.3 Test MSSV

sudo vim test2.c

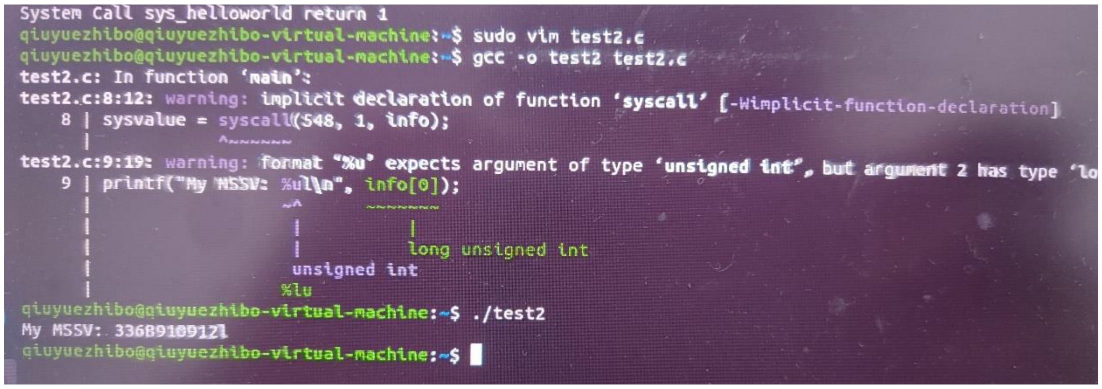
Add following codes:

```
#include <sys/syscall.h>
#include <stdio.h>
#define SIZE 10
int main() {
long sysvalue;
unsigned long info[SIZE];
sysvalue = syscall(548, 1, info);
printf("My MSSV: %ul\n", info[0]);
}
```



```
#include <sys/syscall.h>
#include <stdio.h>
#define SIZE 10

int main() {
long sysvalue;
unsigned long info[SIZE];
sysvalue = syscall(548, 1, info);
printf("My MSSV: %ul\n", info[0]);
}
~
~
~
~
```

gcc -o test2 test2.c
./test2



```
System Call sys_helloworld return 1
qiuyuezhibo@qiuyuezhibo-virtual-machine:~$ sudo vim test2.c
qiuyuezhibo@qiuyuezhibo-virtual-machine:~$ gcc -o test2 test2.c
test2.c: In function ‘main’:
test2.c:8:12: warning: implicit declaration of function ‘syscall’ [-Wimplicit-function-declaration]
    8 | sysvalue = syscall(548, 1, info);
      |            ^~~~~~~
test2.c:9:19: warning: format ‘%u’ expects argument of type ‘unsigned int’, but argument 2 has type ‘lo
    9 | printf("My MSSV: %ul\n", info[0]);
      |                  ~^      ~~~~~~~
      |                   |          |
      |                   |          long unsigned int
      |                   unsigned int
      |                  %lu
qiuyuezhibo@qiuyuezhibo-virtual-machine:~$ ./test2
My MSSV: 3368910912l
qiuyuezhibo@qiuyuezhibo-virtual-machine:~$ █
```

<u>Succeed!</u>
<u>MSSV shown on the screen.</u>

Open the file `include/linux/syscalls.h` and add the following line to the end of this file:

```
struct proc_segs;

asmlinkage long sys_procsched(int pid, struct proc_segs * info);
```

QUESTION: What is the meaning of each line above?

The 1st Line: struct proc_segs:
Proc_segs argument of the type struct
The 2nd line:
Long: function handling syscall return
Asmlinkage is a #define with some gcc magic tells the complier that the function is not expected to optimally find all the arguments in the registers (a common optimization), but only on the CPU's stack
System_call consume its first argument, the system call number and allows up to 4 more arguments passed to real system, on the stack, all system calls are marked with asmlinkage tag, so they all look to the stack for arguments.
Its also used to allow calling a function from assembly files.

First run "make" to compile the kernel and create vmlinuz. It takes a long time to "$ make", we can run this stage in parallel by using tag "-j np", where np is the number of processes you run this command.

```
$ make
or
$ make -j 4
```

vmlinuz is "the kernel". Specifically, it is the kernel image that will be uncompressed and loaded into memory by GRUB or whatever other boot loader you use.
Then build the loadable kernel modules. Similarly, you can run this command in parallel.

```
$ make modules
or
$ make -j 4 modules
```

QUESTION: What is the meaning of these two stages, namely "make" and "make modules"?

"Make": to compile and link the kernel image, there is a single file name "vmlinuz"

"Make modules"" compiles individual files for each question you answered "M" during kernel config. The object code is linked against your freshly built kernel. For each question answered "Y" there are already part of "vmlinuz" and questions answered "N" are skipped.

After compiling and executing this program, your MSSV should be shown on the screen.
QUESTION: Why this program could indicate whether our system works or not?

The program is to check if system call has been integrated into kernel by calling syscall ([number 32],1. Info], info[0] should be the MSSV. Because MSSV is defined in the first line of the struct proc_segs

5 Wrapper

5.1 Create "procsched.h" and "procsched.c" files.

procsched.c:

```
#include "procsched.h"
#include <linux/kernel.h>
#include <sys/syscall.h>
#define _SYS_PROCMEM 548

long procsched(pid_t pid, struct proc_segs * info) {
long sysvalue;
sysvalue = syscall(_SYS_PROCMEM, pid, info);
return sysvalue;
}
```

Note: You must define fields in `proc_segs` struct in the same order as you did in the kernel.
QUESTION: Why we have to re-define `proc_segs` struct while we have already defined it inside the kernel?

Because we have to leave out kernel source code directory and create to store the source for our wrapper. The redefine of proc_segs is for the prototype of the wrapper. They are in different directory with the one defined inside the kernel.

procsched.h:

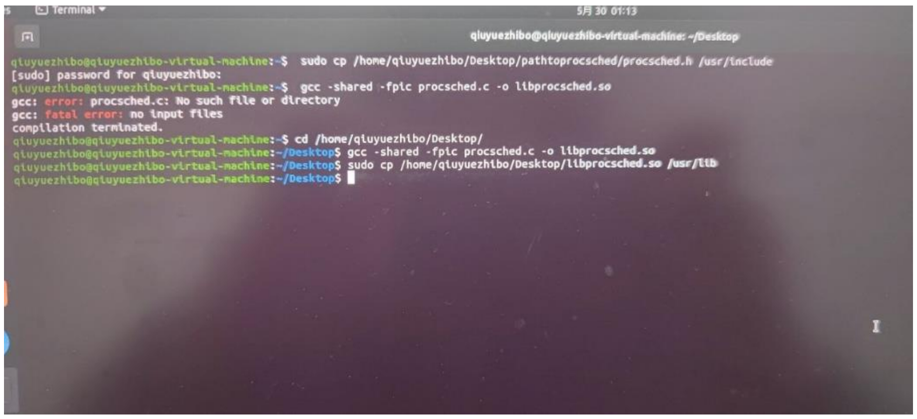
```
#ifndef _PROC_MEM_H_
#define _PROC_MEM_H_
#include <unistd.h>

struct proc_segs {
unsigned long mssv;
unsigned long pcount;
   unsigned long long run_delay;
   unsigned long long last_arrival;
unsigned long long last_queued;
};

long procsched(pid_t pid, struct proc_segs * info);

#endif //  _PROC_SCHED_H_
```

Validation You could check your work by write an additional test module to call this functions but do not include the test part to your source file (`procsched.c`). After making sure that the wrapper work properly, we then install it to our virtual machine. First, we must ensure everyone could access this function by making the header file visible to GCC. Run following command to copy our header file to header directory of our system:

```
$ sudo cp <path to procsched.h> /usr/include
```

QUESTION: Why root privilege (e.g. adding sudo before the cp command) is required to copy the header file to `/usr/include`?

The command "CD" needs to be run with root privilege. We are copying our header file "procsched.h" top header directory of our system so that everyone could access this function. This changed the privilege of the file, therefore, needs the root privilege "sudo"

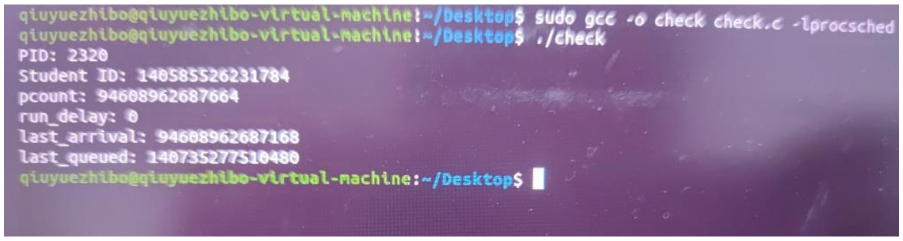


```
qiuyuezhibo@qiuyuezhibo-virtual-machine:~$  sudo cp /home/qiuyuezhibo/Desktop/pathtoprocsched/procsched.h /usr/include
[sudo] password for qiuyuezhibo:
qiuyuezhibo@qiuyuezhibo-virtual-machine:~$  gcc -shared -fpic procsched.c -o libprocsched.so
gcc: error: procsched.c: No such file or directory
gcc: fatal error: no input files
compilation terminated.
qiuyuezhibo@qiuyuezhibo-virtual-machine:~$ cd /home/qiuyuezhibo/Desktop/
qiuyuezhibo@qiuyuezhibo-virtual-machine:~/Desktop$ gcc -shared -fpic procsched.c -o libprocsched.so
qiuyuezhibo@qiuyuezhibo-virtual-machine:~/Desktop$ sudo cp /home/qiuyuezhibo/Desktop/libprocsched.so /usr/lib
qiuyuezhibo@qiuyuezhibo-virtual-machine:~/Desktop$ 
```

Last Step!
Create check.c file to check:

```c
#include <procsched.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdint.h>

int main() {
     pid_t mypid = getpid();
     printf("PID: %d\n", mypid);
     struct proc_segs info;

     if (procsched(mypid, &info) == 0) {
          printf("Student ID: %lu \n", info.mssv);
          printf("pcount: %lu \n", info.pcount);
          printf("run_delay: %llu \n", info.run_delay);
          printf("last_arrival: %llu\n", info.last_arrival);
          printf("last_queued: %llu\n", info.last_queued);
     }else{
          printf("Cannot get information from the process %d\n", mypid);
     }
     // If necessary, uncomment the following line to make this program run
     // long enough so that we could check out its maps file
     // sleep(100);
}
```

sudo gcc -o check check.c -lprocsched
. /check



```
qiuyuezhibo@qiuyuezhibo-virtual-machine:~/Desktop$ sudo gcc -o check check.c -lprocsched
qiuyuezhibo@qiuyuezhibo-virtual-machine:~/Desktop$ ./check
PID: 2320
Student ID: 140585526231784
pcount: 94608962687664
run_delay: 0
last_arrival: 94608962687168
last_queued: 140735277510480
qiuyuezhibo@qiuyuezhibo-virtual-machine:~/Desktop$ 
```

Final output:
PID: 2320
Student ID: 140585526231784
pcount: 94608962687664
run_delay: 0
last_arrival: 94608962687168
last_queued: 140735277510480

Succeed and finish!

If the compilation ends successfully, copy the output file to `/usr/lib`. (Remember to add `sudo` before `cp` command).

QUESTION: Why we must put `-share` and `-fpic` option into `gcc` command?.

-Share: indicates that a shared library is generated.
-FPIC: indicates that using position independent cache.
The shared object can be loaded to different positions by different processes.
The purpose is to fulfill the request that compile source code as a shared to allow user to integrate our system call to their applications