National University of Computer & Emerging Sciences Karachi Campus

System Call for Semaphore Reader- Writer Problem

Project Report
Operating Systems
Section: BS(SE) - 4B

20K-1045 Mustafa Zahid
20K-0244 Munir Abbasi
20K-1686 Syed Areeb Ali

Contents

Tools, and Technologies:

Programming Language: C language
Platform: Ubuntu 16.04

1) Introduction

This project is dedicated to creating a system call that deals with the Reader- Writer Problem by making use of semaphores. A system call is a request for a service that is made by the application programs to the operating system; these can be either user system call (without kernel intervention) or kernel system call (with kernel intervention). This is used to help solve the reader writer problem.

2) Project Specification

In the readers-writers problem, there is a critical section that both reader and writer can access. Reader only reads from the data while writer can both read and write to the data. More than one reader can read at the same time. A writer cannot access the data while a reader is reading. No other thread can access the memory while a writer is accessing the data

Important Note.

Initially, we were making the project for process scheduling algorithms, however there was a problem that came up. When we made the project for process scheduling, our lab instructor simply said run .c codes and compare the schedulers. It was easily achieved but then when we concurred from our operating system theory miss, she said this had to be implemented on kernel level. There was minute time left for this as we were ready but the sudden change in plans meant there was mass confusion as to which miss to follow. Thus the project was changed to reader writer system calls using semaphores.

3) Problem Analysis

Problem: How to manage synchronization in such a way that while there is a reader reading the data, there is no writer who can simultaneously change the data which will mean a race condition needs to be prevented.
Input: Semaphore are used.
Process: Use semaphore wait, and post to avoid multiple writes, and reads at a same instant, make the writer wait for all readers to leave.

4) Solution Design

Readers have higher priority than writer. If a writer wants to write to the resource, it must wait until there are no readers currently accessing that resource. Here priority means, no reader should wait if the share is currently opened for reading. This using semaphores this race condition the requirements can be met.

5) Project Breakdown

The work for the project was divided into parts. Firstly the code for the reader and writer problem was combined and developed by the team. Then the user mode code was completely changed into the kernel level code by replacing all of the commands by the team collectively. Then the code was divided into parts and all of the members changed it into the kernel mode. Once changed, the code was integrated together by the team. All members were provided with the code and then all the members made a system call in their laptops. After the system calls were successfully made, the code was presented to the teacher assistant in the meeting.

6) Limitations and errors:

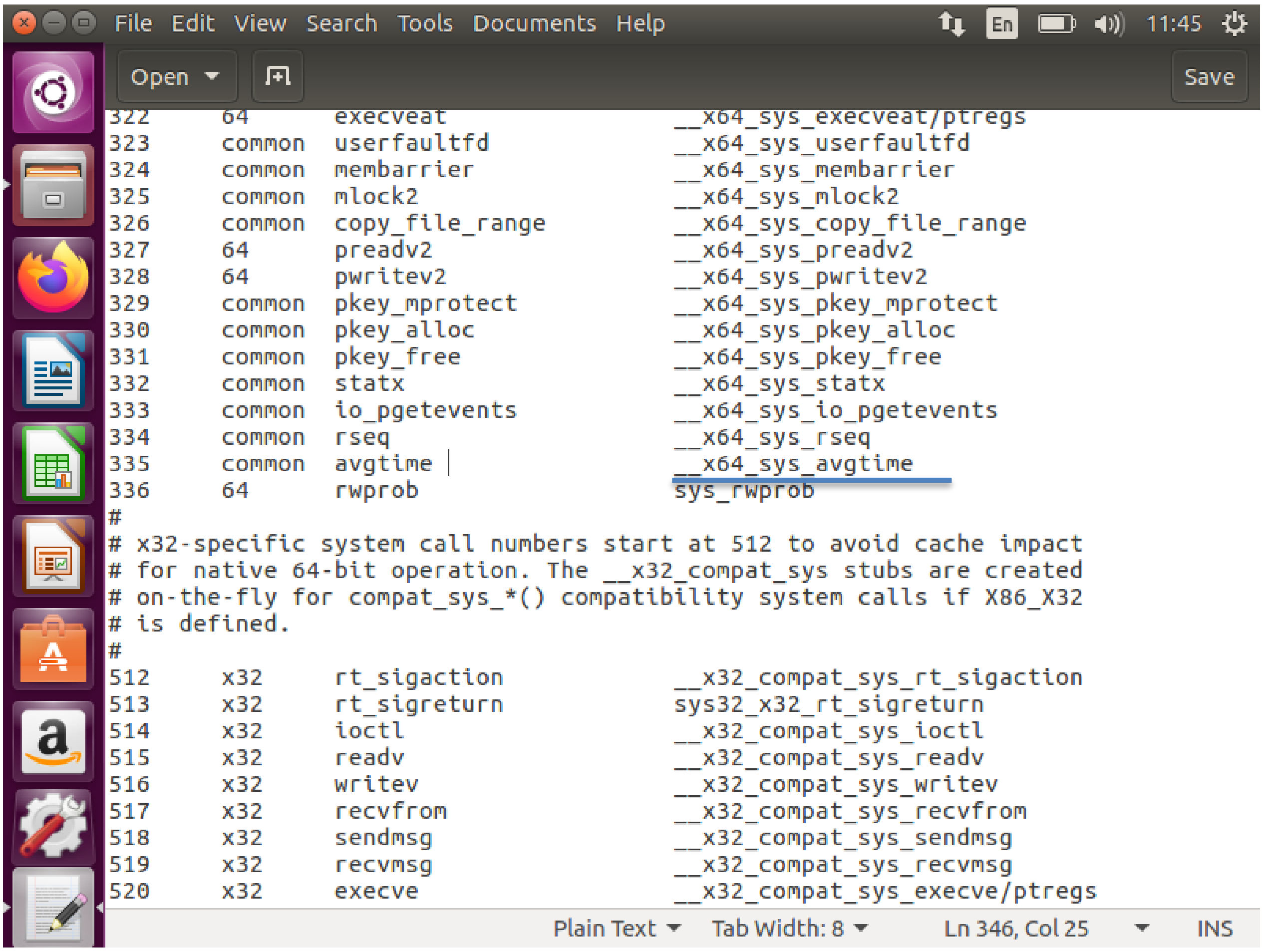
We faced several limitations and errors during this project.

VMLINUX Errors.

This error would come up and the solution for this error was solved by adding a chunk of code from stackoverflow to solve this issue present in the acknowledgement.

Undefined reference to system call.

This error popped up during compilation. This error occurred due to a significant failure in 3rd column of the system call table when the reference was not connected with the c file and had a typing error. This was solved by the error debugging of the whole team.



```
322     64      execveat                __x64_sys_execveat/ptregs
323     common  userfaultfd             __x64_sys_userfaultfd
324     common  membarrier              __x64_sys_membarrier
325     common  mlock2                  __x64_sys_mlock2
326     common  copy_file_range         __x64_sys_copy_file_range
327     64      preadv2                 __x64_sys_preadv2
328     64      pwritev2                __x64_sys_pwritev2
329     common  pkey_mprotect           __x64_sys_pkey_mprotect
330     common  pkey_alloc              __x64_sys_pkey_alloc
331     common  pkey_free               __x64_sys_pkey_free
332     common  statx                   __x64_sys_statx
333     common  io_pgetevents           __x64_sys_io_pgetevents
334     common  rseq                    __x64_sys_rseq
335     common  avgtime                 __x64_sys_avgtime
336     64      rwprob                  sys_rwprob
#
# 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
513     x32     rt_sigreturn            sys32_x32_rt_sigreturn
514     x32     ioctl                   __x32_compat_sys_ioctl
515     x32     readv                   __x32_compat_sys_readv
516     x32     writev                  __x32_compat_sys_writev
517     x32     recvfrom                __x32_compat_sys_recvfrom
518     x32     sendmsg                 __x32_compat_sys_sendmsg
519     x32     recvmsg                 __x32_compat_sys_recvmsg
520     x32     execve                  __x32_compat_sys_execve/ptregs
```


Segmentation faults.

When the kernel was compiled and the system call was triggered by using a test.c code, there was an error which would occur on the second try of the execution of the same code. There would be a segmentation fault occurring. The team sat together to debug this problem. The solution was achieved simply by making a new test call every time. This segmentation fault would occur due to the problem in memory blocks due to the amount of threads and memory that was being consumed by the system but not freeing it and mixing it in with other memory leading to segmentation faults.

INITRAMFS.

There was a problem as initramfs would load up on shell level and remove the linux GUI as a UUID id was missing. This was solved by following the link on stack overflow mentioned in acknowledgements.We used the blkid to detect bad memory blocks, then fcsk on that block was called.

7) Conclusions.

The end result of team efforts is a system call that deals with the Reader-Writer Problem. This system call is used essentially to manage synchronization so that there are no problems with the object data for the readers and the writers. Even though a direct output was not needed, we were able to provide the output for the average time spent by the reader and writer in critical region.

8) Acknowledgement

We made use of several websites to solve the errors that were arising during the compilation of the kernel and also when setting up the Ubuntu GUI after bad blocks were preventing access.

INITRAMFS Error (Used the blkid command, and then called fcsk function to clear the sda blocks on the system)
VMLINUX errors (Used the code provided pasting it in syscalls)

9) Code Snippets:

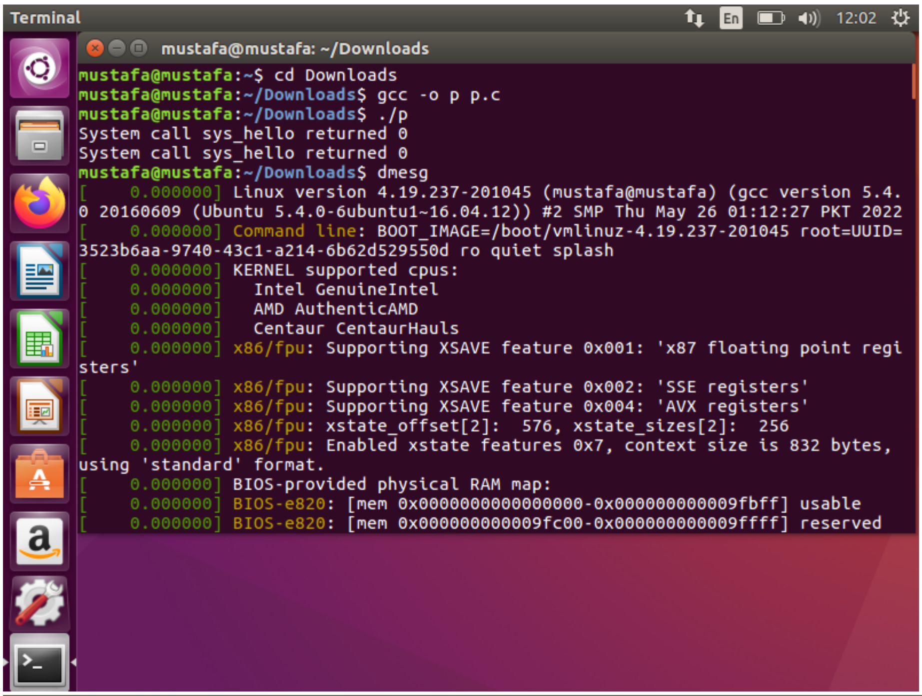
We have directly presented the code for the kernel level which was developed by the team together.

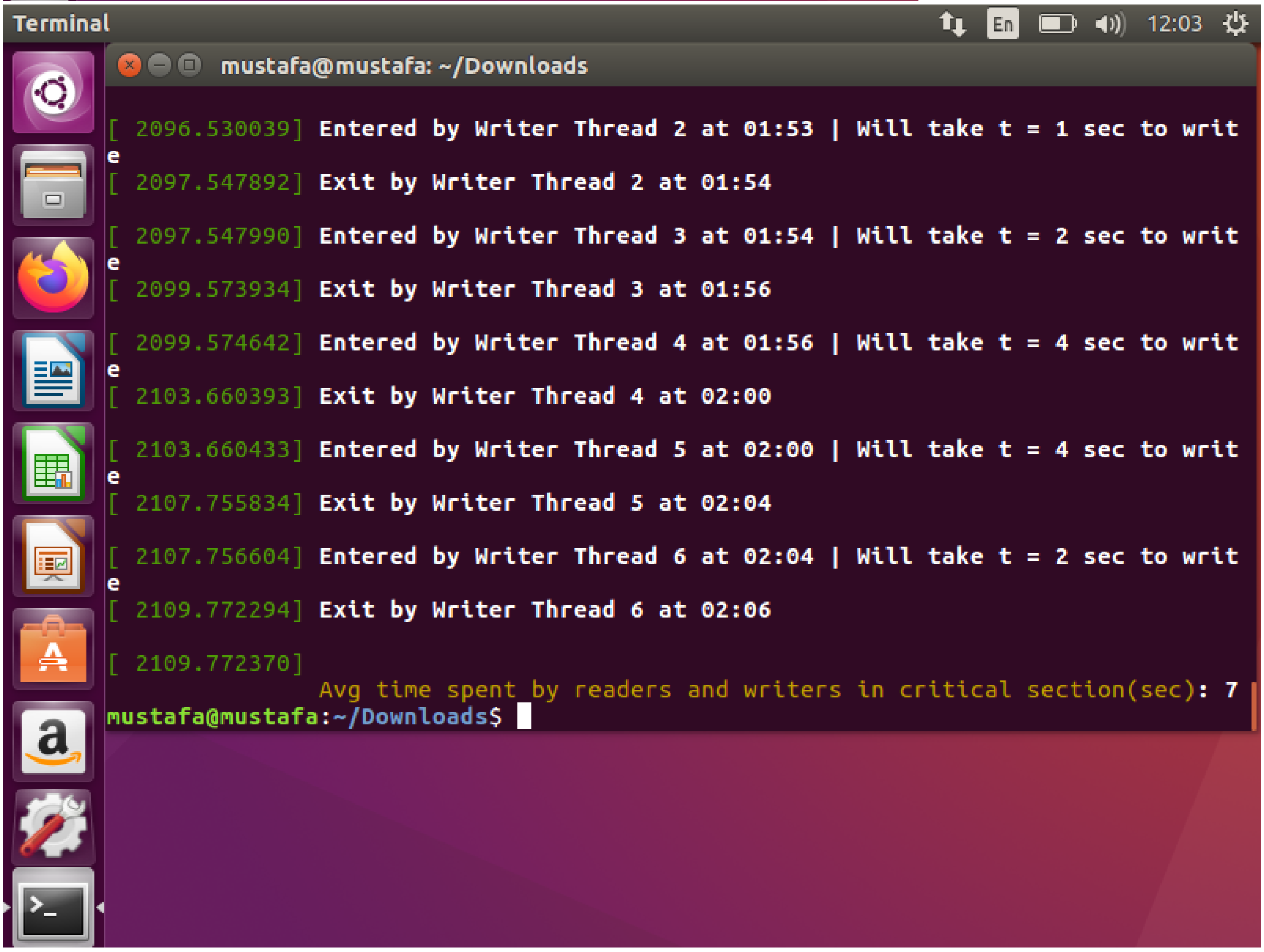
```
#include <linux/init.h>
#include <linux/semaphore.h>
#include<linux/module.h>
#include<linux/kernel.h>
#include<linux/kthread.h>
#include<linux/sched.h> #include<linux/time.h>
#include<linux/timer.h>
#include<linux/delay.h>
#include<asm/delay.h>
#include<linux/random.h>

#define parm_reader 6
#define parm_writer 6
#define rrand_int 1000
 struct read_info{
    int id;
    int time;
};  int writer(void *i);     // writer thread int reader(void *i);      // reader threads int i,j,k,l,m,n; long int avgtime,avg_time;
int wrand_int=0;

static struct semaphore mutex;            // semaphore for writer entry into cs
static struct semaphore rwmutex;          // semaphore for shared variable read_count safety
static struct semaphore avgmutex;         // semaphore for shared variable avg_time safety
int read_count=0;       // number of readers inside cs

int writer(void * param){

    int id = *(int*)param;      struct timespec rs;     struct timespec es;     struct timespec ls;


    printk("\n");
    // calculating request time      getnstimeofday(&rs);     printk("Request by Writer Thread %d at
%.2lu:%.2lu\n",id,(rs.tv_sec/60)%60,rs.tv_sec%60);
```

```c
    down(&rwmutex); // wait writer

    // calculating entry time

    get_random_bytes(&wrand_int,sizeof (wrand_int));
    if(wrand_int<0)      wrand_int = -1*wrand_int;    wrand_int = ( wrand_int % 4 ) + 1; wrand_int*=1000;

    getnstimeofday(&es);

    printk("Entered by Writer Thread %d at %.2lu:%.2lu | Will take t = %d sec to write\n",id,(es.tv_sec/60)%60,es.tv_sec%60,wrand_int/1000);

    msleep_interruptible(wrand_int);

    getnstimeofday(&ls);
    printk("Exit by Writer Thread %d at
%.2lu:%.2lu\n",id,(ls.tv_sec/60)%60,ls.tv_sec%60);

    up(&rwmutex);   // signal writer

    down(&avgmutex);
 // adding waiting time to shared variable avg_time  avg_time += es.tv_sec-rs.tv_sec;
    up(&avgmutex);

    printk("\n");
    return 0;
}
int reader(void * param)
{

    struct read_info *ptr = (struct read_info*)param;
    struct timespec rs;     struct timespec es;     struct timespec ls;

    printk("\n");   getnstimeofday(&rs);
    printk("Request by Reader Thread %d at %.2lu:%.2lu\n",ptr>id,(rs.tv_sec/60)%60,rs.tv_sec%60);


    down(&mutex);  // wait for read_count access permission     read_count++;       // increment read count as new reader is entering
    if(read_count==1)   // only if this is the first reader         down(&rwmutex); // then wait for cs permission      up(&mutex);   // signal mut

    getnstimeofday(&es);
    printk("Entered by Reader Thread %d at %.2lu:%.2lu | Will take t = 1 sec to read\n",ptr->id,(es.tv_sec/60)%60,es.tv_sec%60);

    msleep_interruptible(rrand_int);

    down(&mutex);   // wait for read count access permission     read_count--;       // decrement readcount as we are done reading    if(read_count==

    getnstimeofday(&ls);
    printk("Exit by Reader Thread %d at %.2lu:%.2lu\n",ptr>id,(ls.tv_sec/60)%60,ls.tv_sec%60);

    up(&mutex);

    down(&avgmutex);
// adding waiting time to shared variable avg_time  avg_time += es.tv_sec-rs.tv_sec;
    up(&avgmutex);

    printk("\n");
    return 0;
}
```

```
asmlinkage long sys_rwprob(void)
{

    // creating n pthreads
    static  struct task_struct *w[parm_writer],*r[parm_reader];     int w_attr[parm_writer];     struct read_info read[parm_reader];

    // initialising semaphores     sema_init(&mutex,1);        sema_init(&rwmutex,1);      sema_init(&avgmutex,1);

    for (j = 0; j < parm_reader; ++j) {

        get_random_bytes(&read[j].time,sizeof (read[j].time) );
        if(read[j].time<0)           read[j].time = -1*read[j].time;
        read[j].time = ( read[j].time % 4 ) + 1;
        read[j].time*=1000;

        read[j].id=j+1;

                r[j]=kthread_create(reader,&read[j],"reader");
        if(r[j])
        wake_up_process(r[j]);
    }
    for (i = 0; i < parm_writer; ++i) {
        w_attr[i]=i+1;
                w[i]=kthread_create(writer,&w_attr[i],"writer");
    if(w[i])
    wake_up_process(w[i]);
    }
    for(l=0;l<parm_reader;l++)                  kthread_stop(r[l]);  for(k=0;k<parm_writer;k++)
                kthread_stop(w[k]);

    avgtime=((avg_time)/((parm_reader+parm_writer)));   printk("\nAvg time spent by readers and writers in critical section(sec): %ld\n",avgtime);


    return 0;
}
```

Code output.


Terminal
12:02
mustafa@mustafa: ~/Downloads
mustafa@mustafa:~$ cd Downloads
mustafa@mustafa:~/Downloads$ gcc -o p p.c
mustafa@mustafa:~/Downloads$ ./p
System call sys_hello returned 0
System call sys_hello returned 0
mustafa@mustafa:~/Downloads$ dmesg
[0.000000] Linux version 4.19.237-201045 (mustafa@mustafa) (gcc version 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1~16.04.12)) #2 SMP Thu May 26 01:12:27 PKT 2022
[0.000000] Command line: BOOT_IMAGE=/boot/vmlinuz-4.19.237-201045 root=UUID=3523b6aa-9740-43c1-a214-6b62d529550d ro quiet splash
[0.000000] KERNEL supported cpus:
[0.000000]   Intel GenuineIntel
[0.000000]   AMD AuthenticAMD
[0.000000]   Centaur CentaurHauls
[0.000000] x86/fpu: Supporting XSAVE feature 0x001: 'x87 floating point registers'
[0.000000] x86/fpu: Supporting XSAVE feature 0x002: 'SSE registers'
[0.000000] x86/fpu: Supporting XSAVE feature 0x004: 'AVX registers'
[0.000000] x86/fpu: xstate_offset[2]: 576, xstate_sizes[2]: 256
[0.000000] x86/fpu: Enabled xstate features 0x7, context size is 832 bytes, using 'standard' format.
[0.000000] BIOS-provided physical RAM map:
[0.000000] BIOS-e820: [mem 0x0000000000000000-0x000000000009fbff] usable
[0.000000] BIOS-e820: [mem 0x000000000009fc00-0x000000000009ffff] reserved


Terminal
12:03
mustafa@mustafa: ~/Downloads
[2096.530039] Entered by Writer Thread 2 at 01:53 | Will take t = 1 sec to write
[2097.547892] Exit by Writer Thread 2 at 01:54
[2097.547990] Entered by Writer Thread 3 at 01:54 | Will take t = 2 sec to write
[2099.573934] Exit by Writer Thread 3 at 01:56
[2099.574642] Entered by Writer Thread 4 at 01:56 | Will take t = 4 sec to write
[2103.660393] Exit by Writer Thread 4 at 02:00
[2103.660433] Entered by Writer Thread 5 at 02:00 | Will take t = 4 sec to write
[2107.755834] Exit by Writer Thread 5 at 02:04
[2107.756604] Entered by Writer Thread 6 at 02:04 | Will take t = 2 sec to write
[2109.772294] Exit by Writer Thread 6 at 02:06
[2109.772370]
Avg time spent by readers and writers in critical section(sec): 7
mustafa@mustafa:~/Downloads$

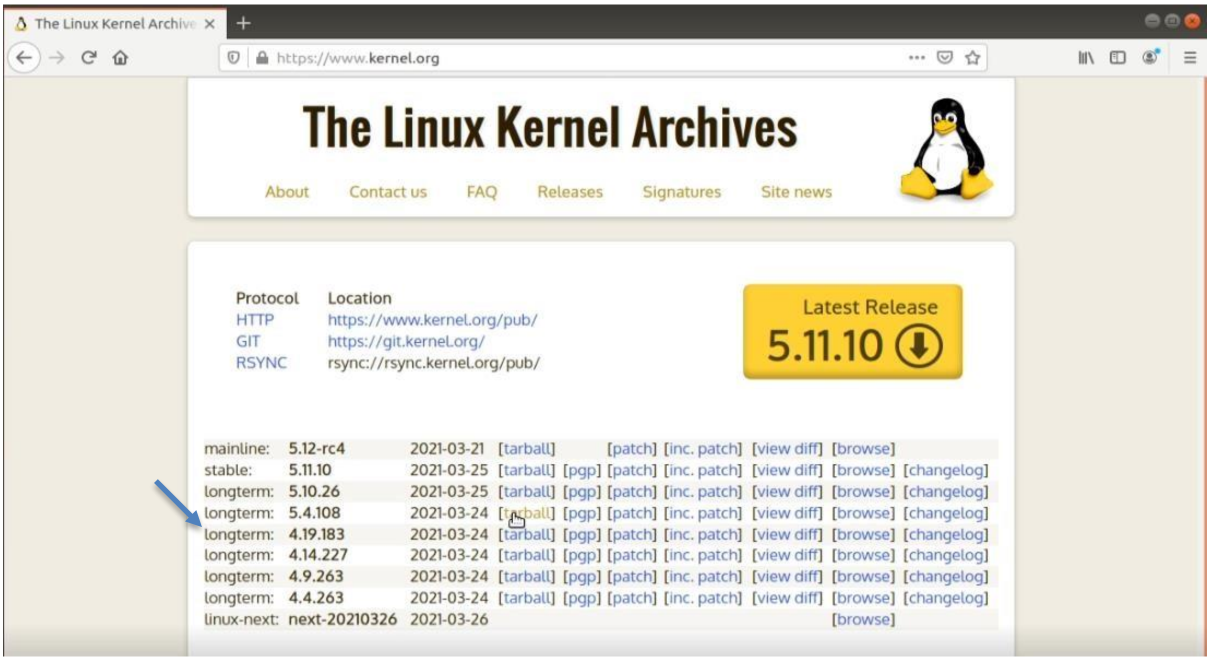
10) How the system call was developed by our team.

First these prerequisites were installed.

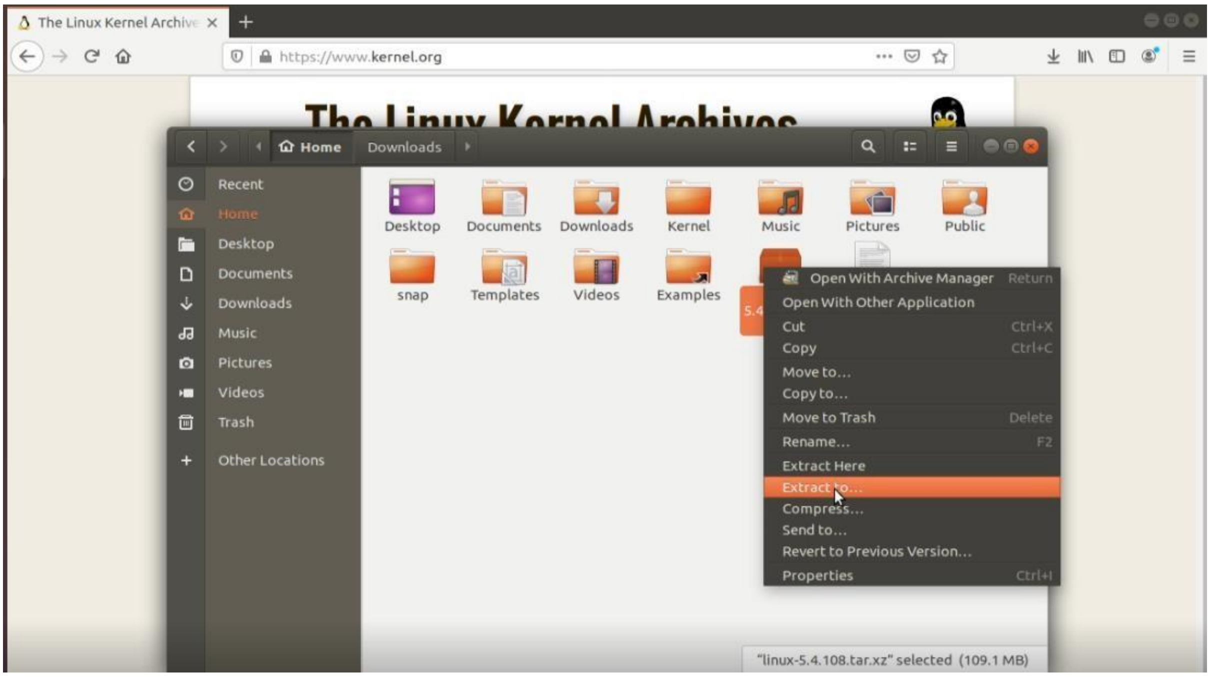
- sudo apt-get install gcc
- sudo apt-get install flex
- sudo apt-get install libncurses5-dev
- sudo apt-get install bison
- sudo apt install make
- sudo apt-get install libssl-dev
- sudo apt-get install libelf-dev
- sudo add-apt-repository "deb http://archive.ubuntu.com/ubuntu $(lsb_release -sc) main universe"
- sudo apt-get update
- sudo apt-get upgrade

Steps:

Downloading a kernel:

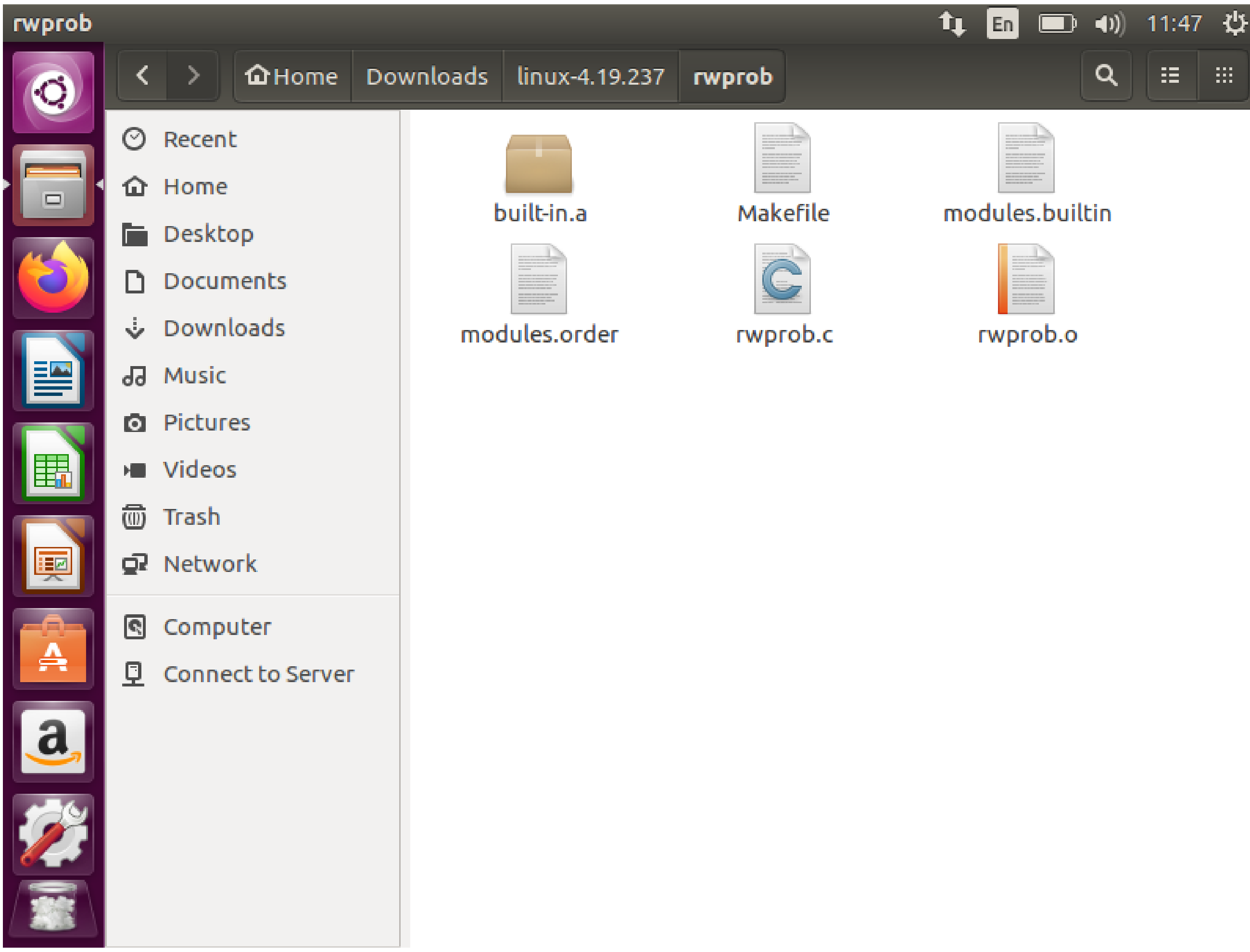


Extracting the kernel: The kernel was extracted.


The Linux Kernel Archive
https://www.kernel.org
Home
Downloads
Recent
Home
Desktop
Documents
Downloads
Music
Pictures
Videos
Trash
Other Locations
Desktop
Documents
Downloads
Kernel
Music
Pictures
Public
snap
Templates
Videos
Examples
Open With Archive Manager Return
Open With Other Application
Cut Ctrl+X
Copy Ctrl+C
Move to…
Copy to…
Move to Trash Delete
Rename… F2
Extract Here
Extract to…
Compress…
Send to…
Revert to Previous Version…
Properties Ctrl+I
"linux-5.4.108.tar.xz" selected (109.1 MB)

Making a new folder called rwprob:

We used open in terminal for the extracted kernel and then made a new folder called rwprob.



Adding a C code for the system call:

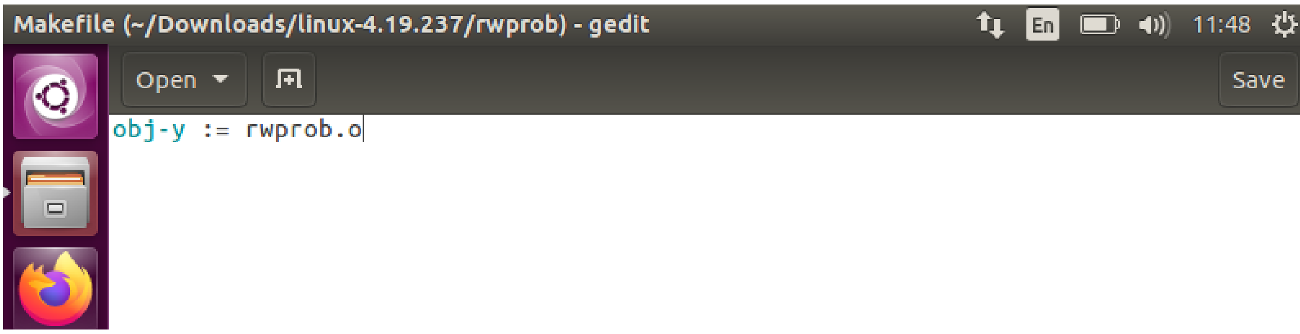
Then, inside the folder we called “gedit rwprob.c” and then pasted our kernel level code.

Code explanation for kernel implementation:

a. We used #include <linux/kernel> because we are building a system call for our linux kernel.
b. The arguments for this function will be on the stack instead of the CPU registers.
c. Printk is used instead of printf because we are going to print in the kernel’s log file.
d. If the code is run and it returns 0, then it will mean that our program ran successfully and output is written to out kernel’s log file.

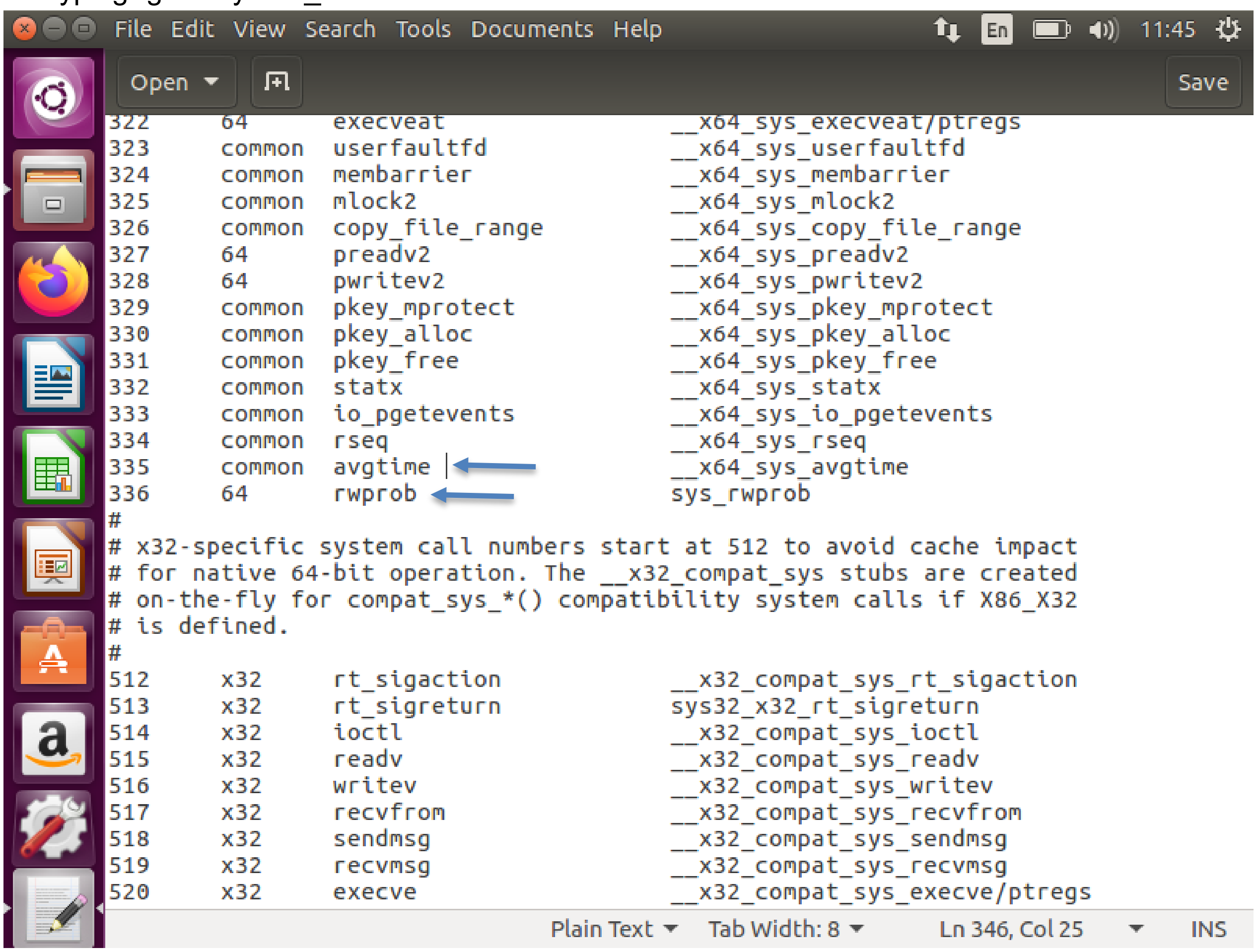
Creating a Makefile for the C code:

Then, we created a Makefile for our new folder to ensure that the code in the folder is always compiled whenever the kernel is compiled. In order to do this, we type in our terminal “gedit Makefile” and put “obj-y := rwprob.o”



Adding the new code into the system table file:

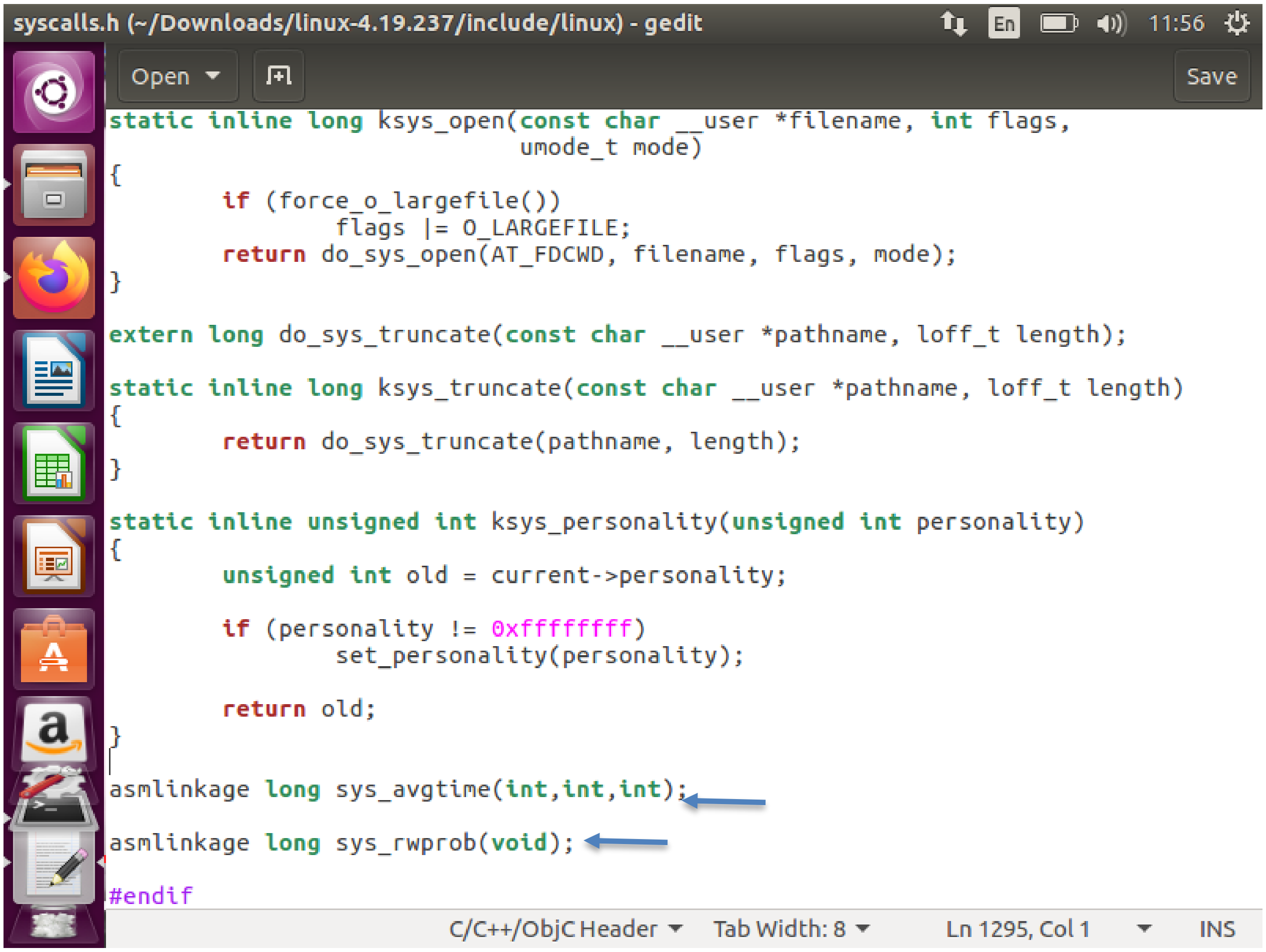
Since we are creating a 64-bit system call according to our system we have to add the system call entry into the syscall_64.tbl file which keeps the name of all the system calls in our system. This tbl file is located inside the kernel folder in /arch/x86/entry/syscalls/syscall_64.tbl. We can go into this directory by using cd and then edit the file by typing “gedit syscall_64.tbl”




```
322     64      execveat                __x64_sys_execveat/ptregs
323     common  userfaultfd             __x64_sys_userfaultfd
324     common  membarrier              __x64_sys_membarrier
325     common  mlock2                  __x64_sys_mlock2
326     common  copy_file_range         __x64_sys_copy_file_range
327     64      preadv2                 __x64_sys_preadv2
328     64      pwritev2                __x64_sys_pwritev2
329     common  pkey_mprotect           __x64_sys_pkey_mprotect
330     common  pkey_alloc              __x64_sys_pkey_alloc
331     common  pkey_free               __x64_sys_pkey_free
332     common  statx                   __x64_sys_statx
333     common  io_pgetevents           __x64_sys_io_pgetevents
334     common  rseq                    __x64_sys_rseq
335     common  avgtime                 __x64_sys_avgtime
336     64      rwprob                  sys_rwprob
#
# 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
513     x32     rt_sigreturn            sys32_x32_rt_sigreturn
514     x32     ioctl                   __x32_compat_sys_ioctl
515     x32     readv                   __x32_compat_sys_readv
516     x32     writev                  __x32_compat_sys_writev
517     x32     recvfrom                __x32_compat_sys_recvfrom
518     x32     sendmsg                 __x32_compat_sys_sendmsg
519     x32     recvmsg                 __x32_compat_sys_recvmsg
520     x32     execve                  __x32_compat_sys_execve/ptregs
```

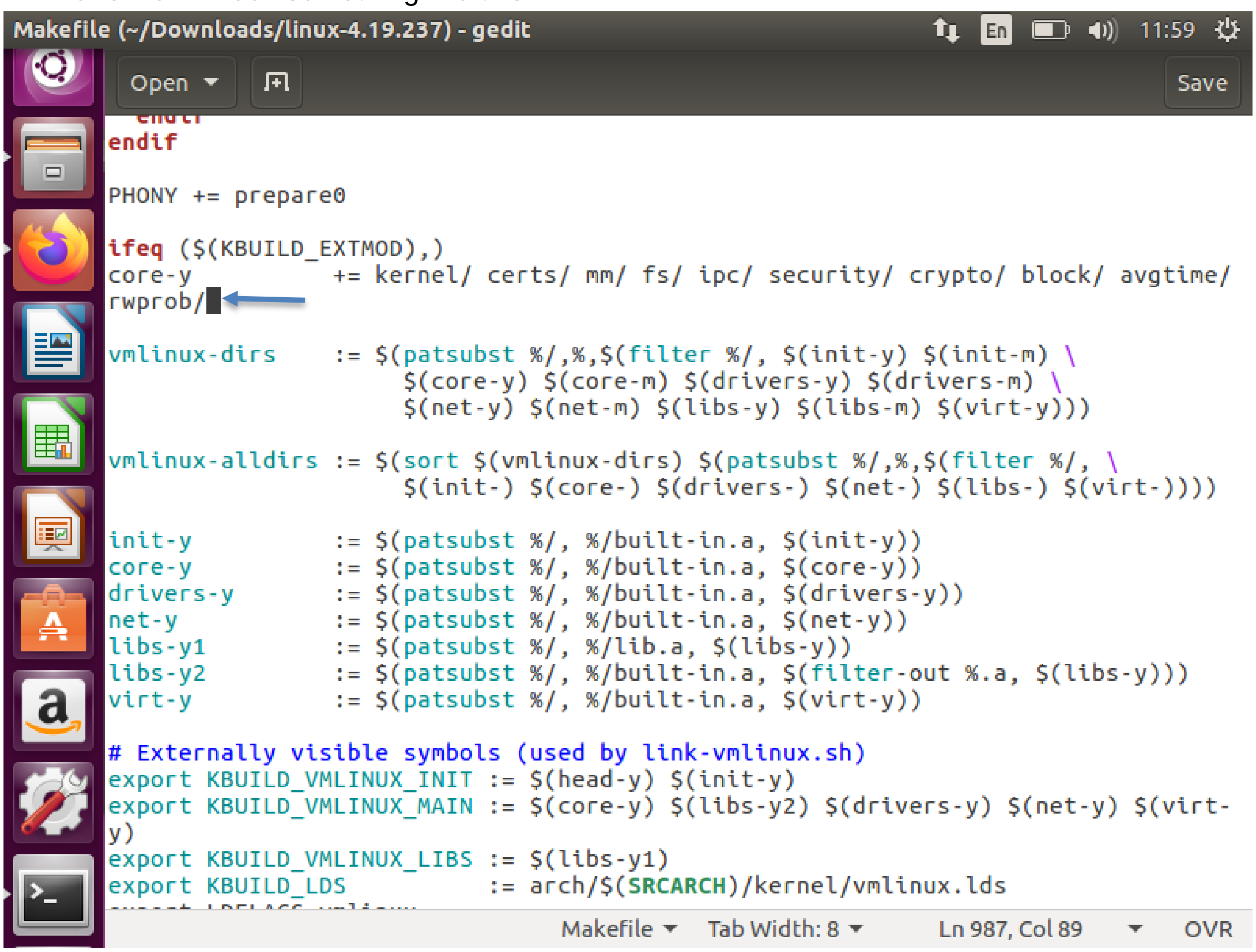
Adding the prototype of the new system call into the system calls header file:

Now we have to add the prototype of our system call in the system's header file which is located in the kernel folder then "/include/linux/syscalls.h". We have to add the prototype of our system call function in this file.


syscalls.h (~/Downloads/linux-4.19.237/include/linux) - gedit
En
11:56
Open
Save
static inline long ksys_open(const char __user *filename, int flags,
umode_t mode)
{
if (force_o_largefile())
flags |= O_LARGEFILE;
return do_sys_open(AT_FDCWD, filename, flags, mode);
}
extern long do_sys_truncate(const char __user *pathname, loff_t length);
static inline long ksys_truncate(const char __user *pathname, loff_t length)
{
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;
}
asmlinkage long sys_avgtime(int,int,int);
asmlinkage long sys_rwprob(void);
#endif
C/C++/ObjC Header
Tab Width: 8
Ln 1295, Col 1
INS

Changing version and adding the avgtime and rwprob folder in the kernel’s Makefile:

we open the Makefile of the kernel and search for “core-y” and go to it’s second instance which is under “KBUILD_EXTMOD” and add our new module which is “avgtime followed by rwprob” at the end of it. At the end, our make file will look something like this:



Creating a config file:

Now we have to create a configuration file for our kernel. We will be copying the oldconfig and using that config for new kernel. First of all, we search for the config that we currently have by typing “ls /boot | grep config” and the we copy the config that is shown to us which is our linux kernel directory*”. Then we create the old config by typing “yes "" | make oldconfig –j2”, the system will automatically create the new config for us and select the default option for everything.

Cleaning and Compiling the kernel:

We have to clean all of our existing object and executable file because compiler sometimes link or compile files incorrectly and to avoid this, we delete all of our old object and executable files by typing “make clean –j2” and when this all is done, we type “make –j2” to start building our kernel (-j2 allocates the multiple cores that our system have for compiling. If we don’t do this, the system will only use a single core for compiling the

Commands: yes “” | make oldconfig –j4

make clean –j4

make –j4

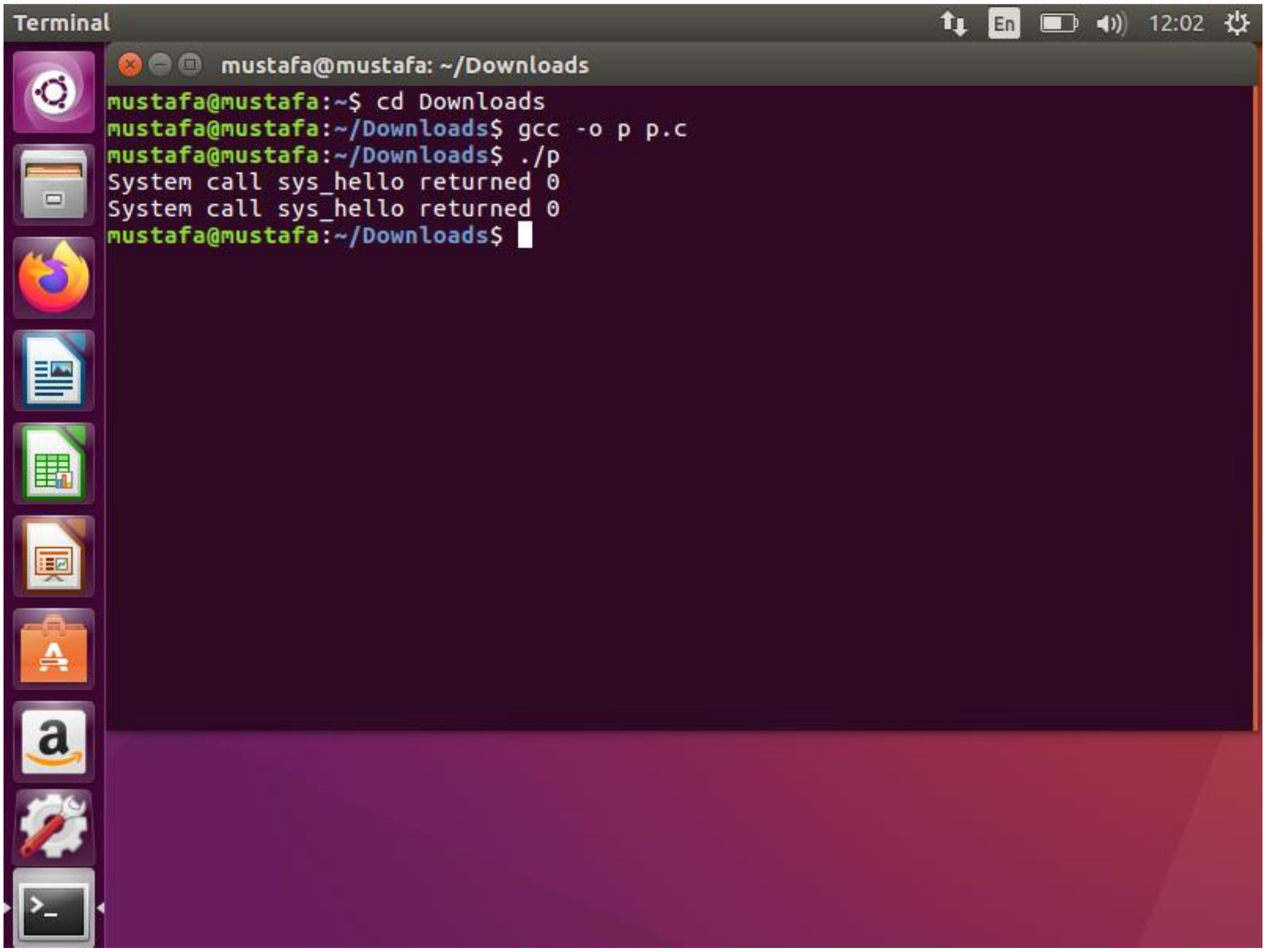
Installing modules:

Now we have to install the kernel that we built by typing “make modules_install install” which will install the kernel and update our grub as well. When this all is done and the terminal says “done”, then we can restart our laptop either manually or by typing “shutdown -r now” and hold the “Shift” key while it is restarting to open up the grub menu and switch to the new kernel which we just installed.

Checking if the System call is Working Properly:

After logging into the newly compiled kernel, we check the system call by making a C code named “p.c” and putting the following code in it:

Now we compile the code by typing “gcc p.c” and executing it by typing “./p”. If it returns 0, this means that our code has compiled successfully and the system call is working fine (Note that in calling syscall(335), 335 is the number where we added our system call in the table) and we also called syscall(336) and finally, we run “dmesg” to see the kernel messages and we will find that our syshello returned 0 as a new call was successfully made and executed at the end of it. All output is pasted before.


Terminal
En
12:02
mustafa@mustafa: ~/Downloads
mustafa@mustafa:~$ cd Downloads
mustafa@mustafa:~/Downloads$ gcc -o p p.c
mustafa@mustafa:~/Downloads$ ./p
System call sys_hello returned 0
System call sys_hello returned 0
mustafa@mustafa:~/Downloads$