A

Project Report

on

'Linux-Ubuntu Installation, Compilation, and Debugging'

submitted by

Mr. Rajesh R. Surana [MS]

Department of Computer Science

Ira A. Fulton Schools of Engineering

Fall-2014

Certificate

This is to certify that, the project report entitled

'Linux-Ubuntu Installation, Compilation, and Debugging'

submitted by
Mr. Rajesh R. Surana

is a bona fide record of his own work performed
out by him in partial
fulfillment for the completion of

MASTER OF SCIENCE IN
COMPUTER SCIENCE
OF
IRA A. FULTON SCHOOLS OF ENGINEERING

Under my supervision and guidance during the
Fall-2014

Dr. Alan K. Skousen (Instructor)

Date: Place: Tempe

DEPARTMENT OF COMPUTER SCIENCE
IRA A. FULTON SCHOOLS OF ENGINEERING

Abstract

This project mainly consists of three parts – installation, compilation and debugging related to Ubuntu. At the end of each step we have significant output and experience learned. Step II is dependent on Step I and so on. First two steps viz., installation of Ubuntu and compilation of Kernel source code are fairly easy as compared to third step-debugging which require considerable diligence and understanding of Linux commands. For easy understanding, many snapshots are used for the expected output after commands or steps in the project.

Content

Project Step	Title		Page No.
	Introduction		5
	Scope		6
1	Installation of Ubuntu		7
	1	Installing VMware Player	8
	2	Installing Ubuntu	9
	Outcome and Experience		11
2	Downloading and Compiling Kernel Source Code		12
	1	Creating directory	13
	2	Downloading and installing utilities	13
	3	Downloading source code	15
	4	Compiling the code	16
	5	Upgrading to new image installation	18
	Outcome and Experience		21
3	Building and Debugging kernel module		22
	1	Creating application	23
	2	Patching the kconfig and Makefile	25
	3	Installing and using .ko	29
	4	Preparation for Debugging the kernel	35
	5	Debugging the kernel	43
	Outcome and Experience		46

Introduction

Reason behind choosing Linux for our project is very simple. It is open source as well as free. In addition, it is easily customizable because we have source code, compiling tools as well as help in the form of online forums as well as blogs. In all, in terms of cost, support and legality of use, Linux is the best choice. In terms of difficulty from 1(Novice) -100(expert), goal of our project is to reach a level of at least 20.

Before using any tool or software, the basic requirement is that we should be able to install it. That's why we have our first step as installation of Ubuntu. This is the step where we make ourselves comfortable with the GUI of the Ubuntu and its functionalities such as terminal, text editor or compiler. If you have previous experience of using any of the personal operating system then this part will be quite mundane for you!

It is important to understand how to update the software to get the latest security mechanisms as well as higher utilities. Same analogy applies to kernel also. In fact, update of kernel is not just meant to be better functionality but an environment which is more secure and bug free as well as easy to use. That's why our second step is to understand the procedure to update a kernel to its latest revision.

Perk of this project is at last where you actually experience the debugging of kernel. Yes, you read it correct! In the last part, we are going to see a very interesting way to debug the kernel. To keep you excited this secret will be opened when you actually reach to third step of this project. After this step you will have ability to debug the kernel code which is the first step towards understanding the kernel source code.

[Note: This project is based on the problem statements given by the instructor for each of the three parts, hence no references mentioned.]

Scope

Ubuntu can be installed on a personal computer in a number of ways.

For example:

1) The method that uses the least space, at least initially, is to install Ubuntu in VMware-Player as a virtual machine using a growing virtual disk. Use VMWare-Fusion for Mac.

2) Ubuntu can be installed as an OS co-resident with windows. Complete information can be found regarding installation here.

In this project, we will be using virtual machine for demonstration. Other details are as follows:-

Host OS: -Windows 7 (64-bits)

VMware-player: - 6.0.3 build-1895310

Guest OS: - Ubuntu 14.04 LTS (64-bits)

RAM for Guest OS: - 1GB

Linux revision before compilation:-3.13.0.32-generic

VMware-player can be downloaded from its official website here.

Latest Ubuntu Desktop OS can be downloaded from here.

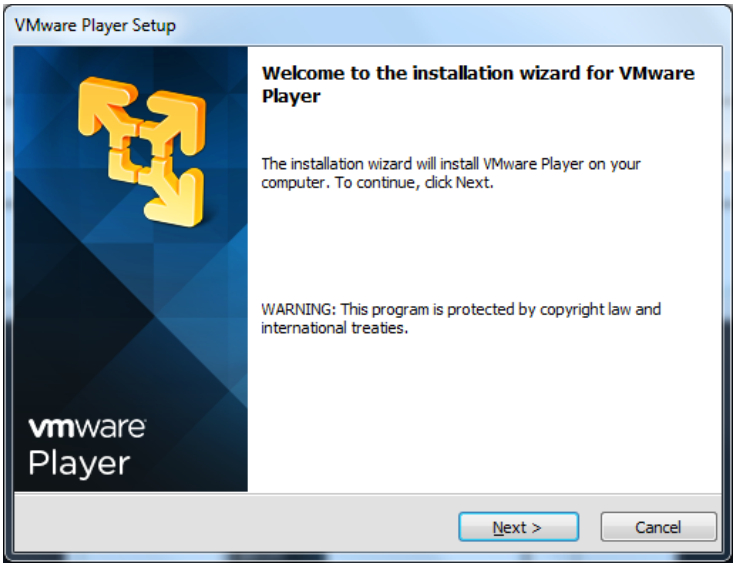
Motivation behind use of Virtual Machine (VMware-Player)-

1. Installed Guest OS can be easily recovered if corrupted.
2. Guest OS can be easily migrated to other machines where virtual machine is installed.
3. It requires least space.
4. If Guest OS got compromised or corrupted, then it could not harm the host OS. So, for new learners as well as for researchers this is the best platform to experiment.

Part I – Installation of Ubuntu

1. Installing VMware Player-

1.1. Just double click on the VMware Player downloaded software and following window will appear.

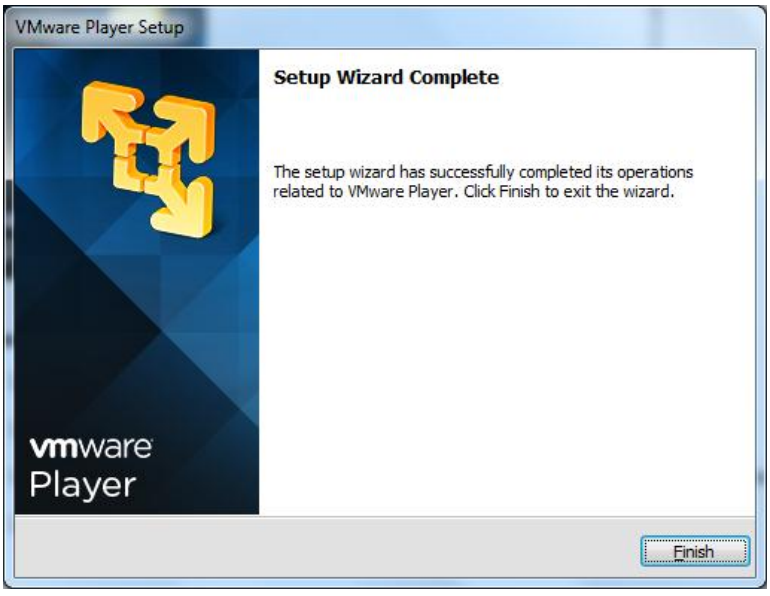


1.2. Accept the terms and conditions and click on Next.

1.3. Select the destination directory where you want to install the software files.

1.4. If you are currently connected to internet, then check the software update checkbox.

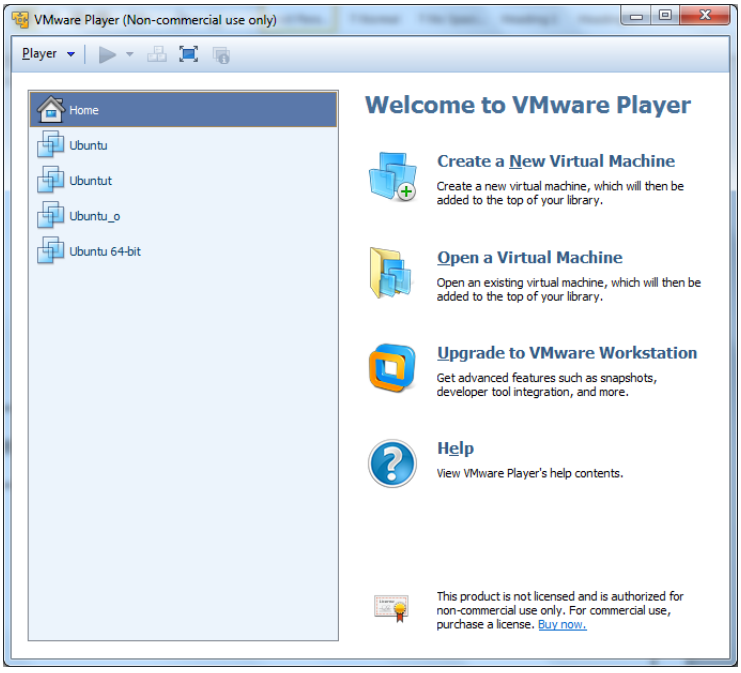
1.5. Then click on Continue/Next till actual installation starts.



1.6. Click on Finish and you are ready to use the VMware-Player.

2. Installing Ubuntu-

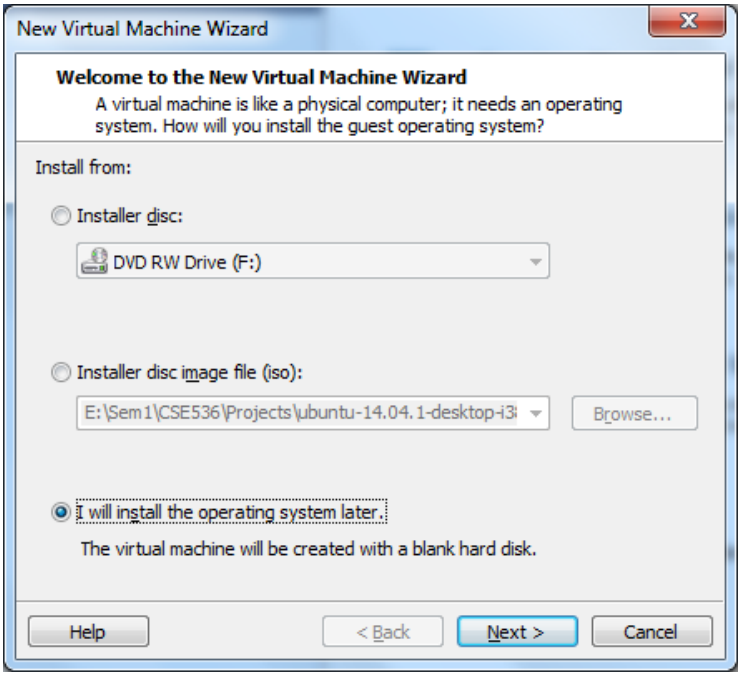
2.1. After opening the VMware-Player you will have the following window.



Here, you can see 4 virtual machines already created by me below Home tab.

2.2. Click on the 'Create a New Virtual Machine' icon.

2.3. Now on the next window select third option.

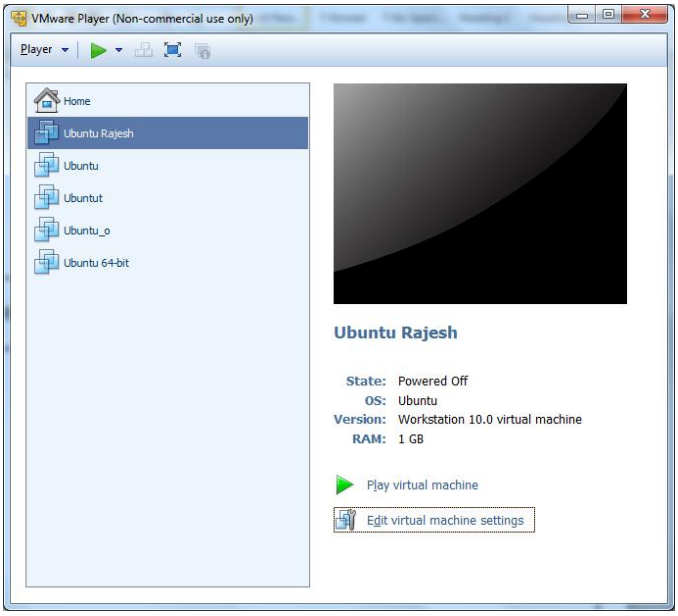


2.4. Select Guest Operating System as Linux and version as Ubuntu 64-bit. Click Next.

2.5. Select name for your Virtual Machine such as Ubuntu <yourname>. Select the location where you want your virtual machine to be created. Click Next.

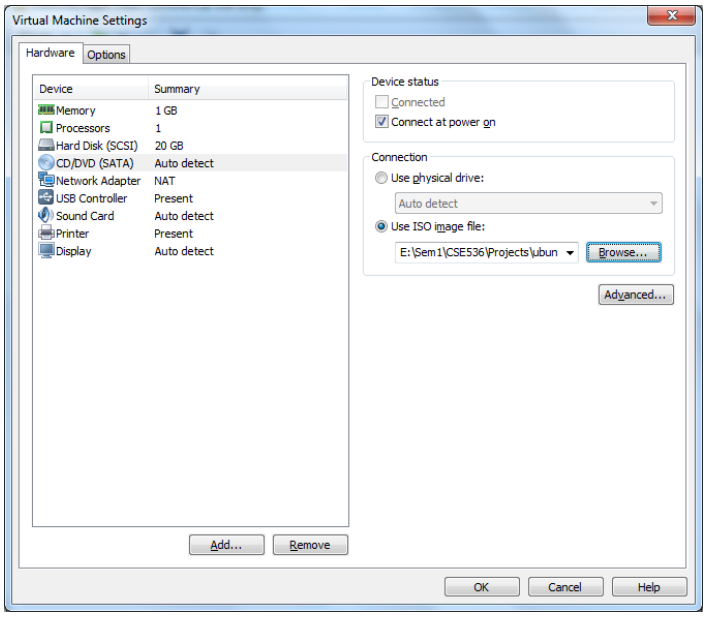
2.6. Click on ‘Split virtual disk into multiple files’. Click Next. Click Finish.

2.7. Below you can see the new virtual machine as ‘Ubuntu Rajesh’



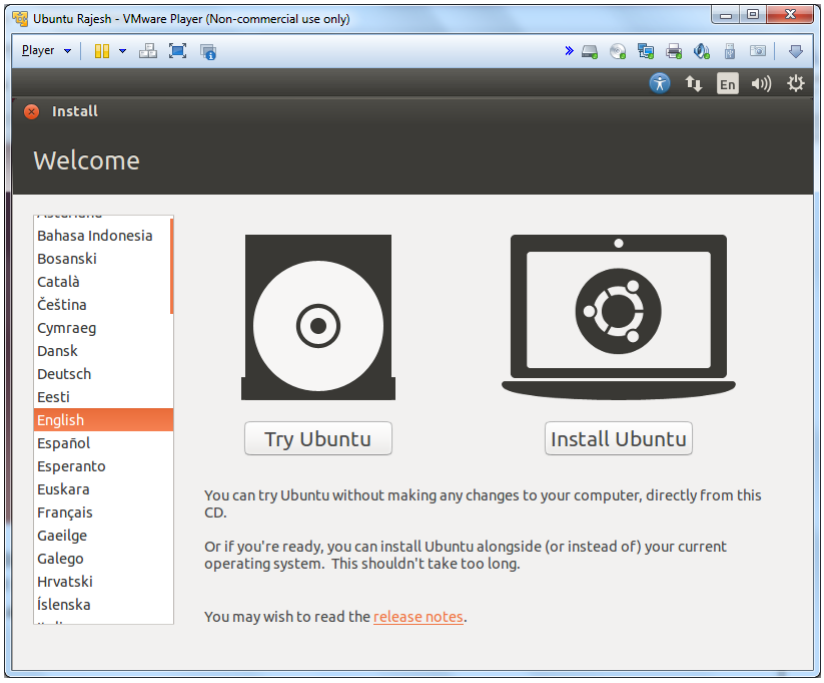
2.8. Click on ‘Edit Virtual Machine Setting’.

2.9. In the ‘Hardware’ Section select ‘CD/DVD’ device. Now, in the ‘Use ISO image file’ section select the path of your Ubuntu iso file. Click OK.



2.10. Click on the play virtual machine.

2.11. Click on the Install Ubuntu option.



2.12. After this continue following the appropriate instructions and once done restart the virtual machine.

Outcome and Experience:-

Here, we learned how to install VMware-Player which can be used to run many of the supporting OS. In addition, we learned here how to install Ubuntu on the VMware.

Part II – Download and Compile Kernel Source Code

In preparation for compiling applications and kernel objects to run in the Ubuntu development environment we need to have the kernel source code and associated utilities available on our copy of Ubuntu. The following commands executed in a terminal CLI will download the kernel source code and produce an installation image that will allow us to upgrade to the latest kernel and subsequently make modifications to it.

Linux revision before compilation:-3.13.0.32-generic

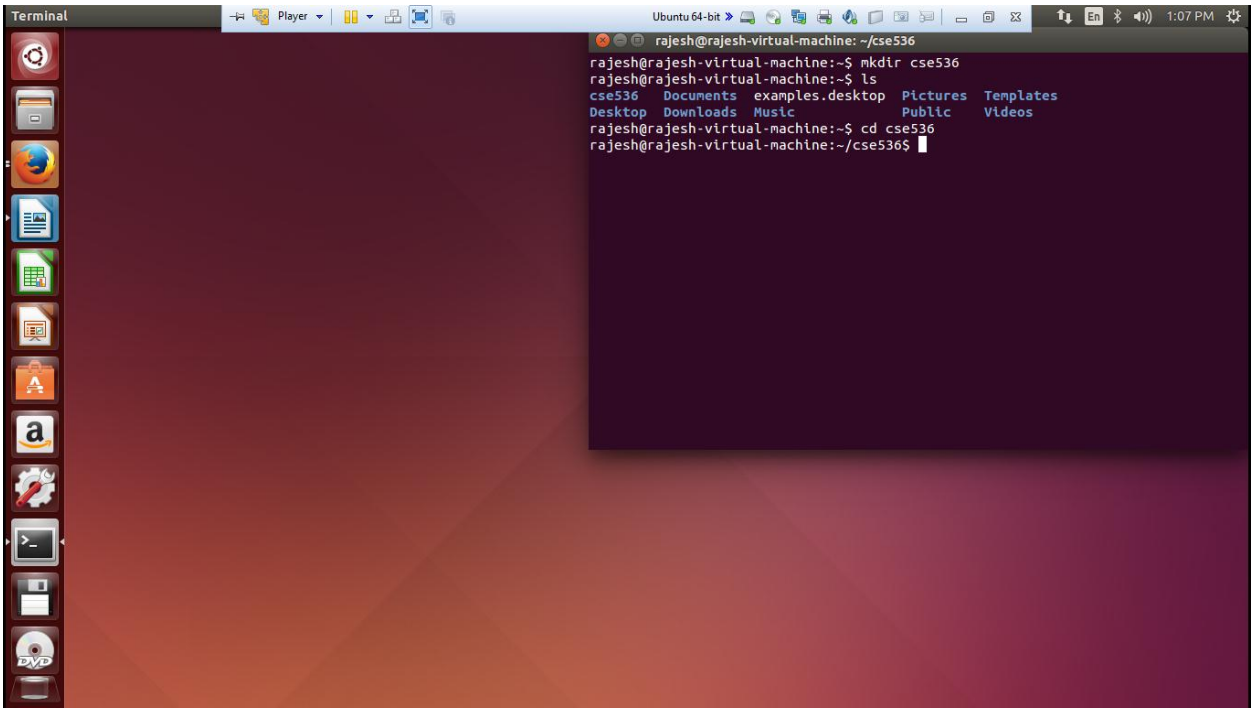
Linux revision after compilation:-3.13.11.6

This step is sub-divided into 5 steps.

Step 1 – Make a directory for kernel development. This is done in a local home directory.

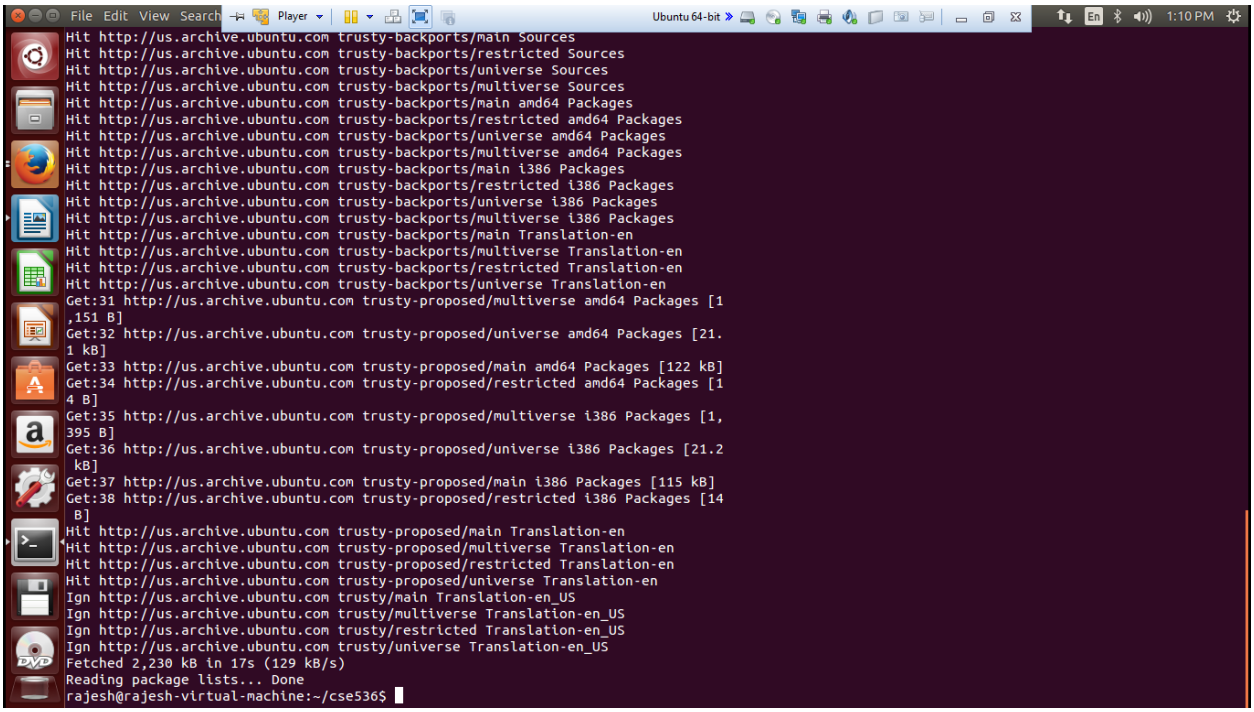
mkdir cse536

cd cse536

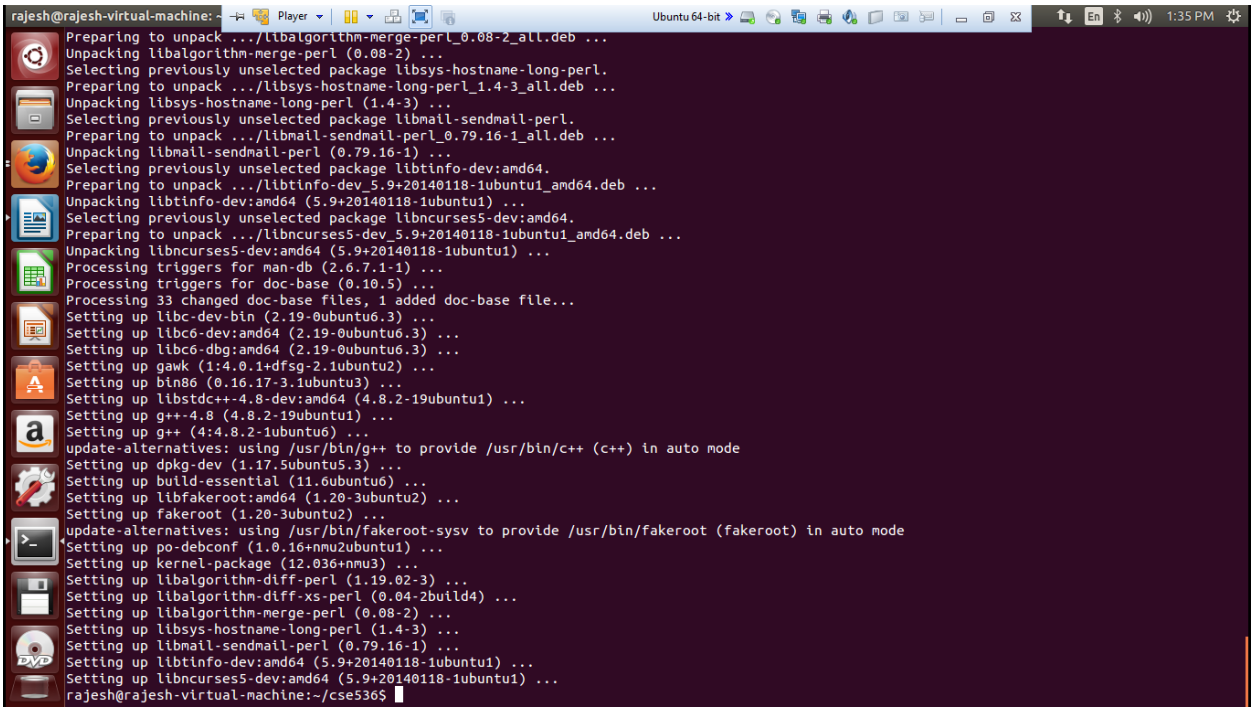


Step 2 – Download and install the utilities needed to perform the compile and packaging of the kernel. We start by first updating the package list. After that we can download all of the needed packages. Some will have already been installed so this is a shotgun approach but should capture any missing utilities.

sudo apt-get update

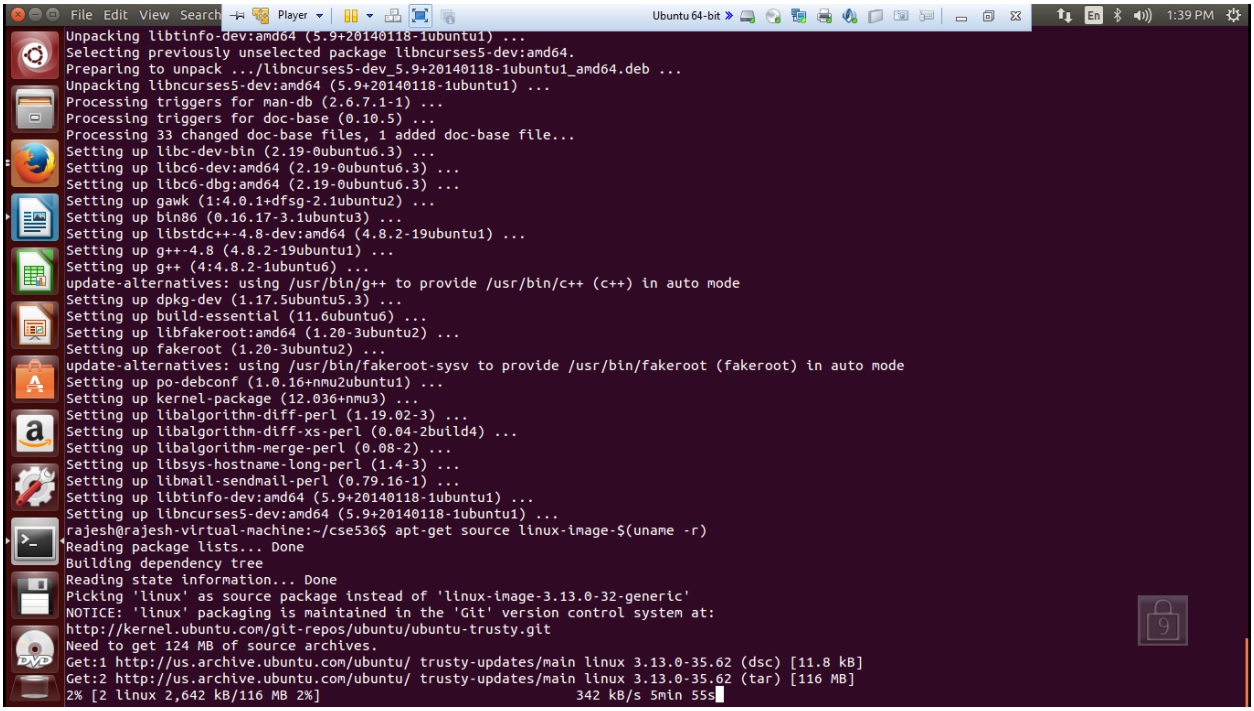


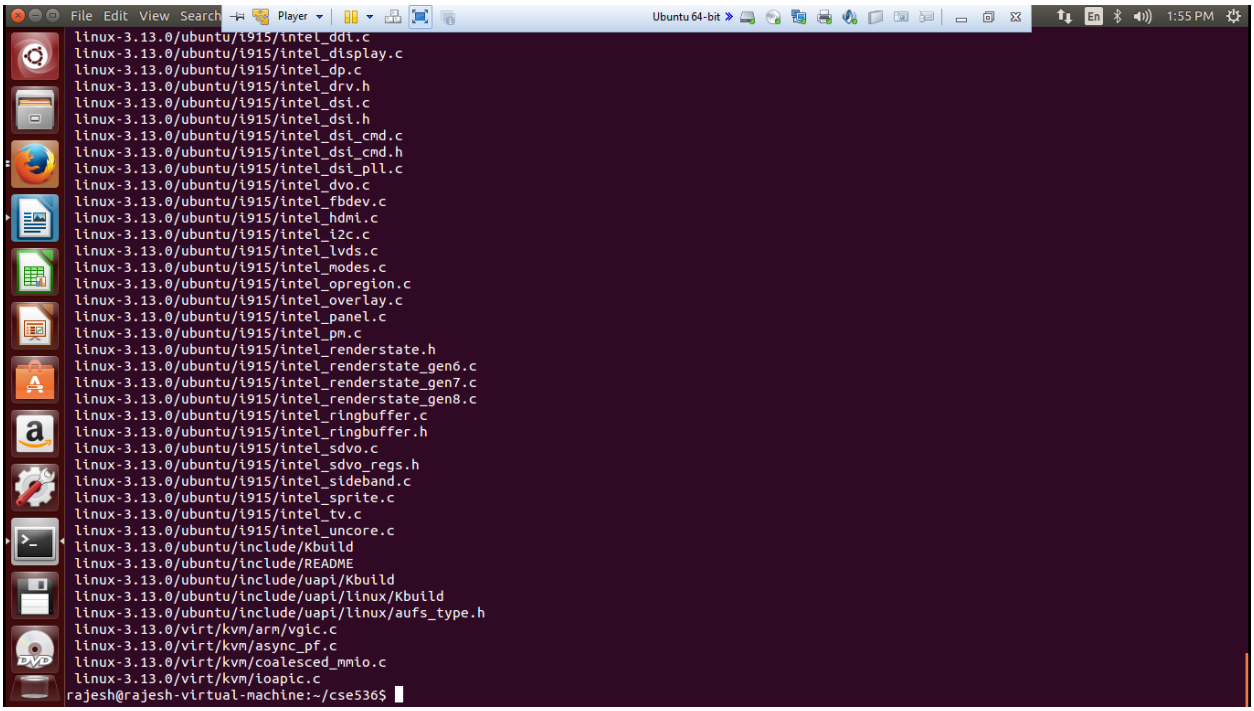
sudo apt-get install dpkg-dev kernel-package gcc libc6-dev binutils make bin86 module-init-tools gawk gzip grep libncurses5-dev



Step 3 – The next instruction downloads and decompresses the source code for the latest kernel from the Ubuntu source tree. Do not use root for this.

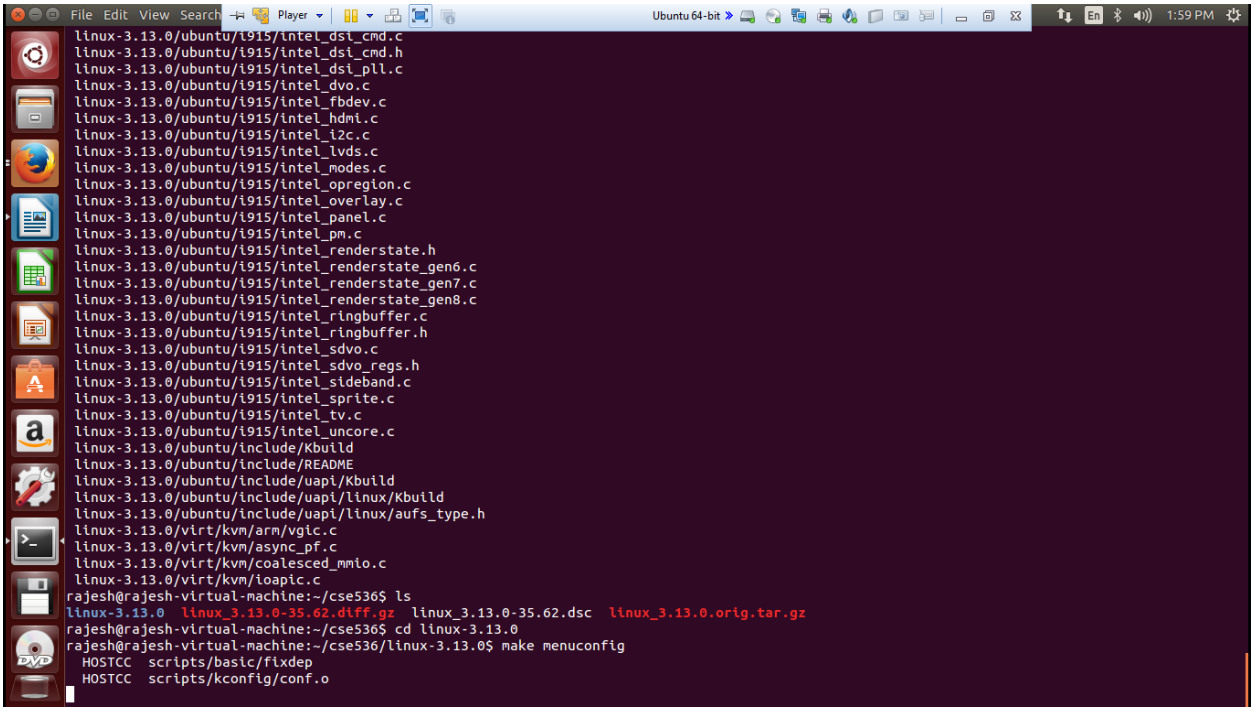
apt-get source linux-image-$(uname -r)



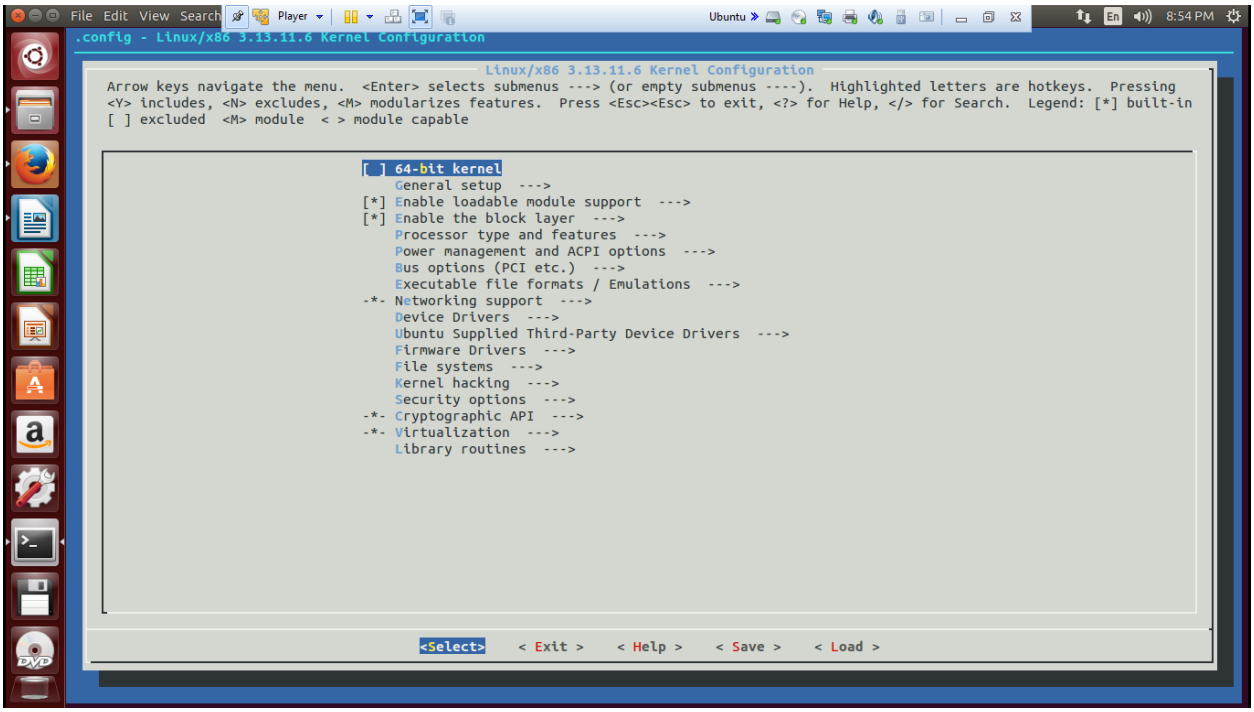


Step 4 – This step will take the longest. We must first a) change the working directory to be the top level source directory for the kernel. We can then b) configure the kernel. For now we will retain the default configuration so just exit after the configurator starts. c) The last preparatory step is to clean the build environment. This will cause everything to be rebuilt. d) The last command will take several hours to complete so be sure your computer is plugged in and is not set to sleep. For some computers it may take all night. Do not use any root privileges in this step.

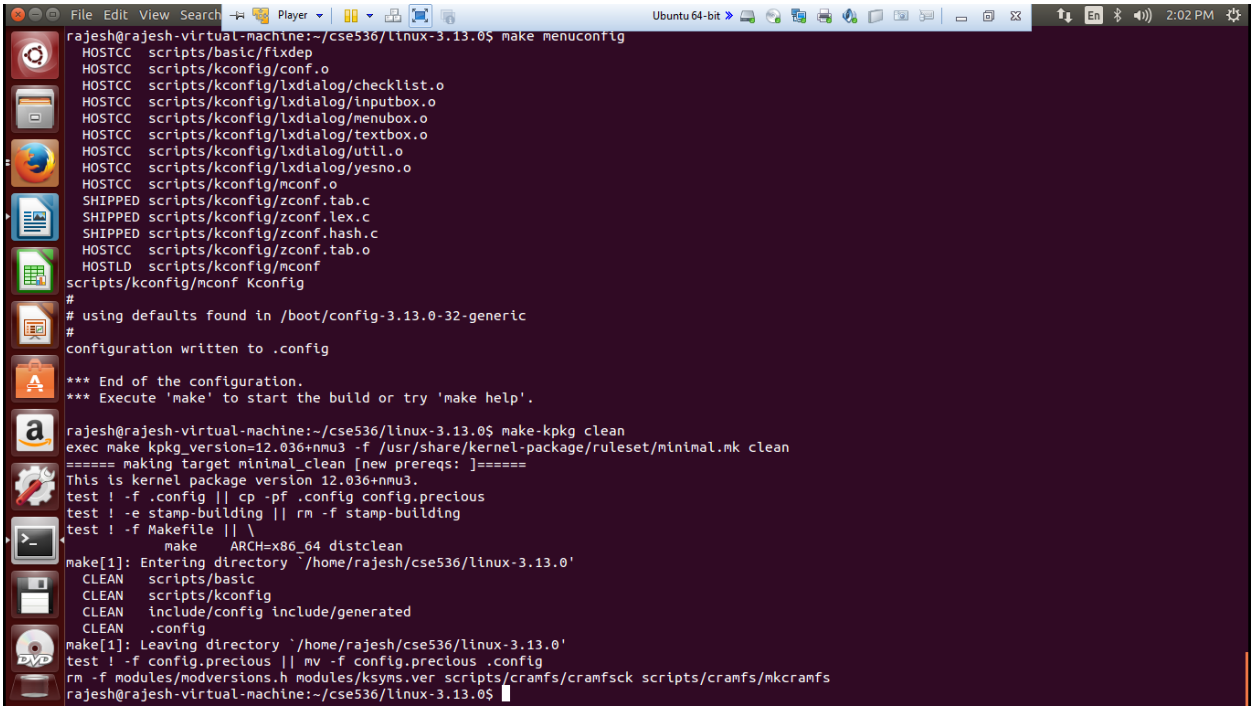
cd linux-3.13.0



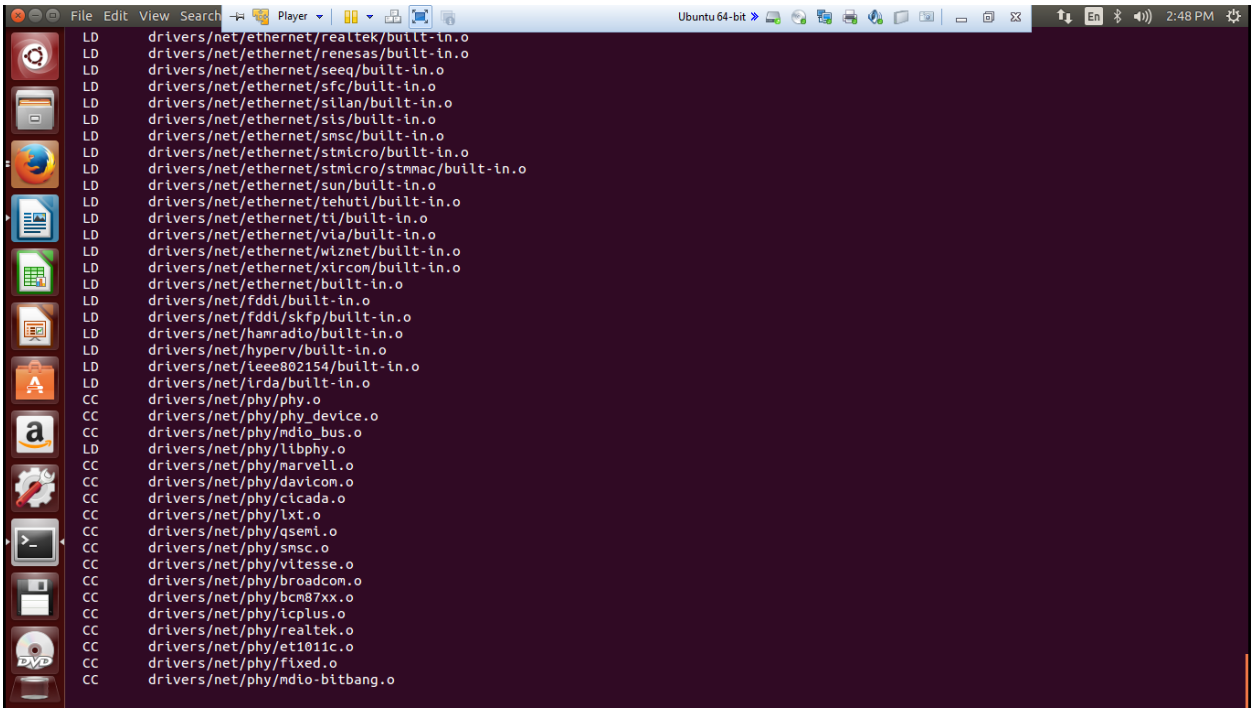
make menuconfig

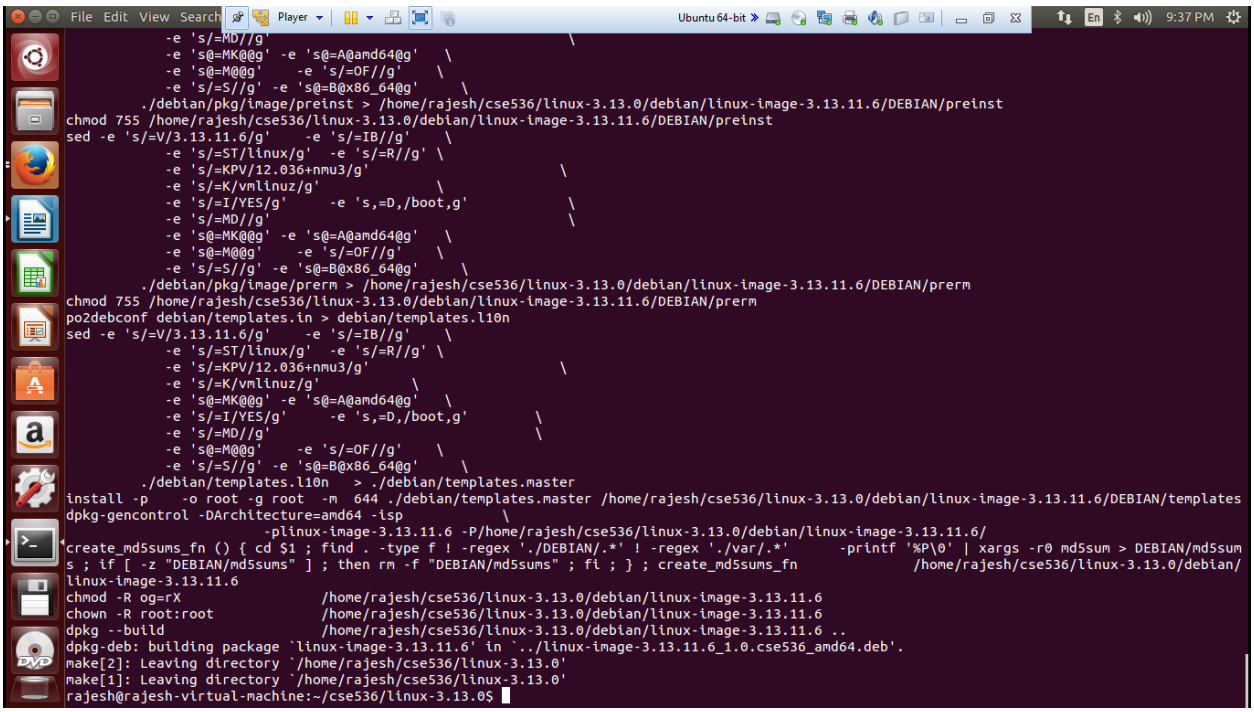


make-kpkg clean



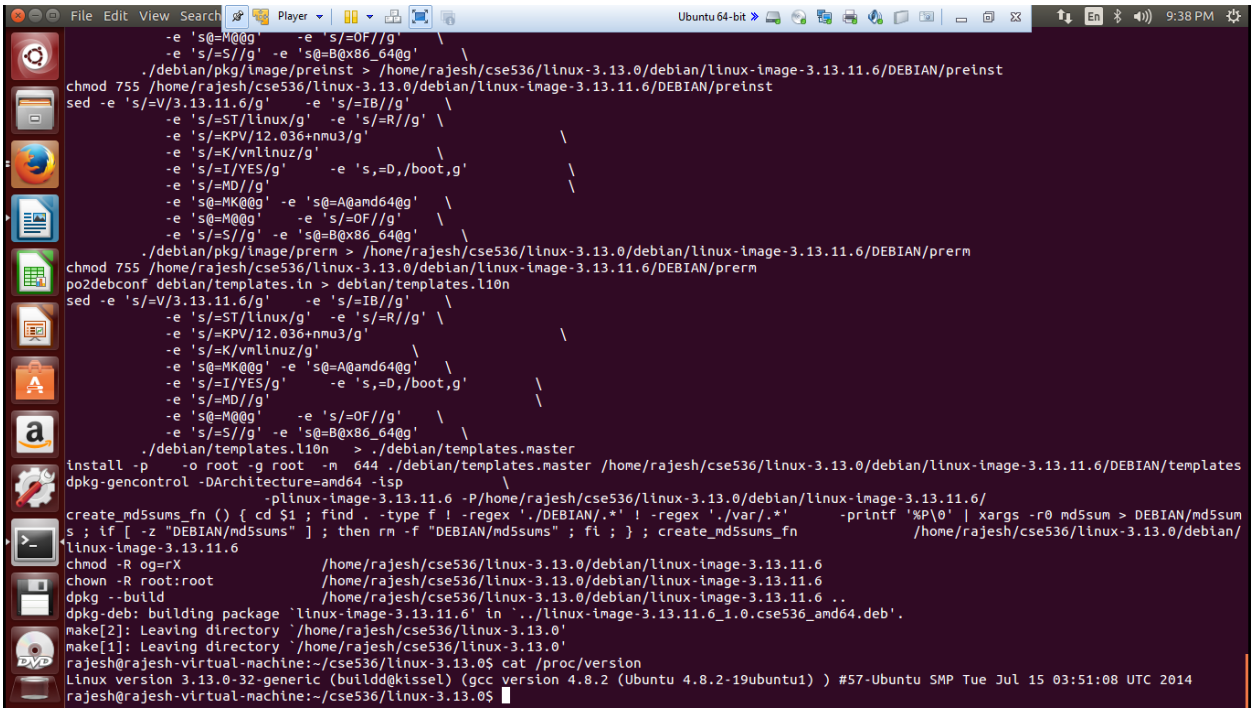
fakeroot make-kpkg --initrd --revision=1.0.cse536 kernel_image



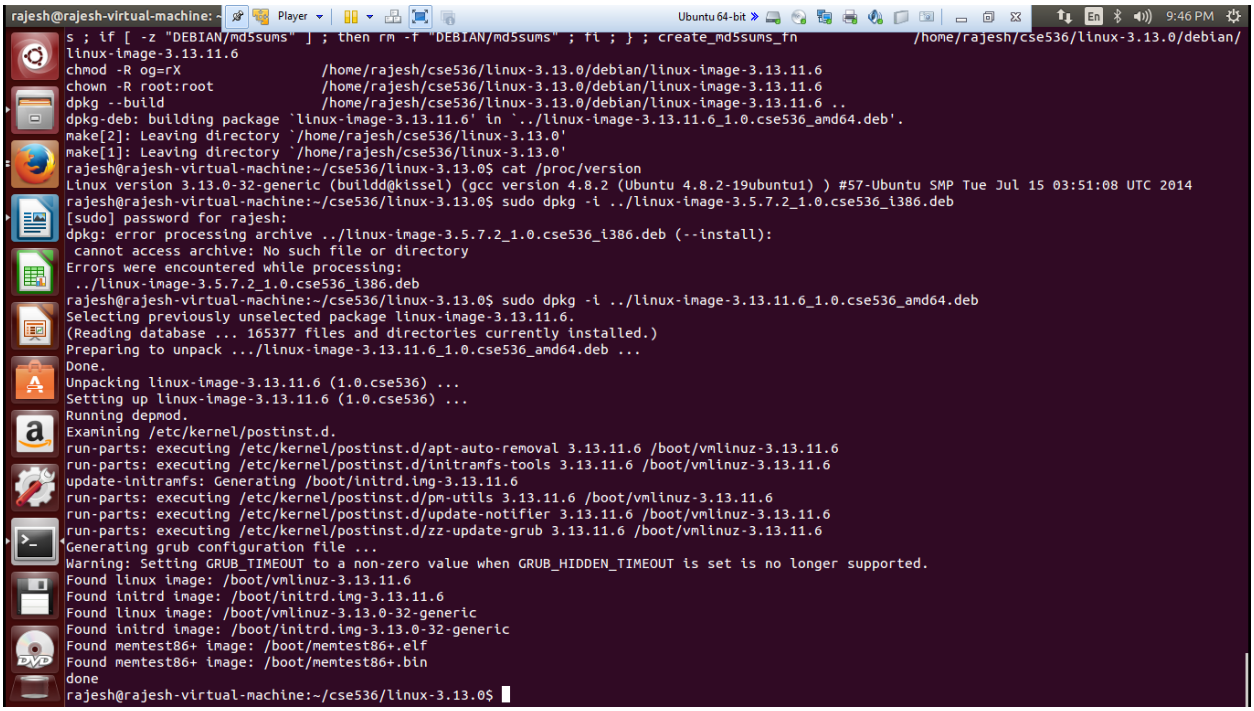


Step 5 – This step will show us your current revision of the OS (Note: *Software Updater* also updated the kernel) and then update the kernel to match your source code. We must reboot (shutdown command) to use the new kernel. We can verify the update by displaying the version again.

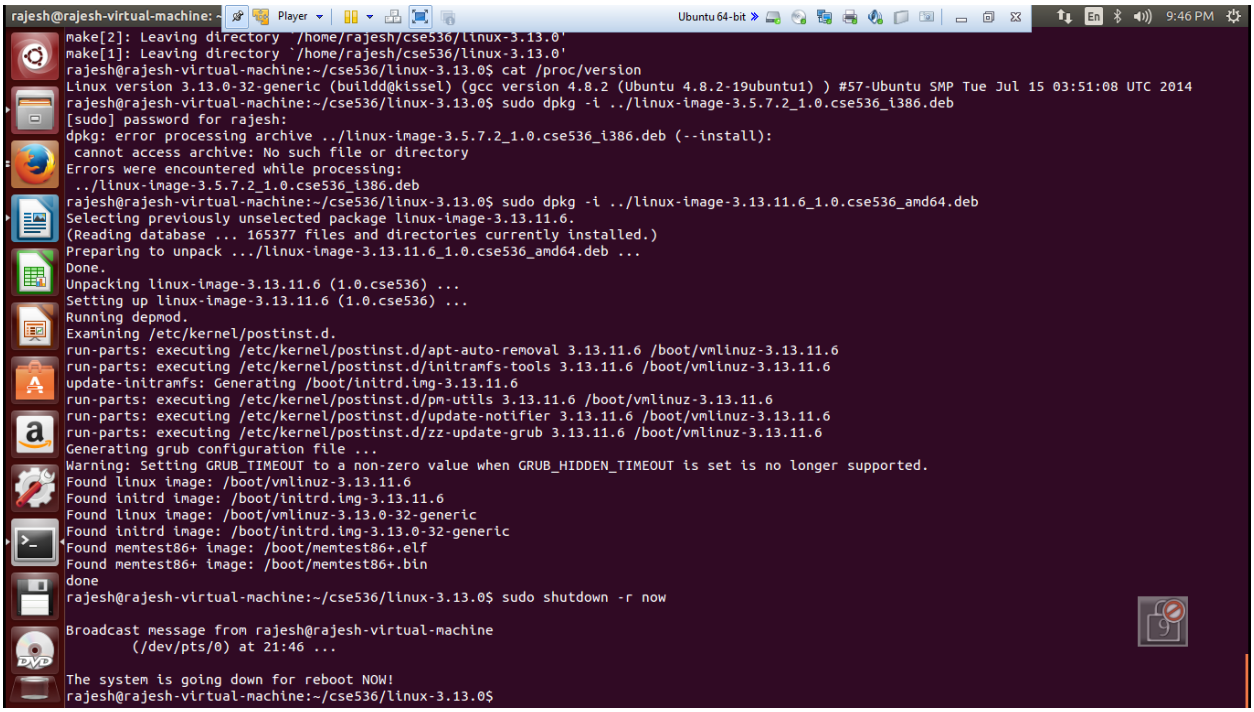
cat /proc/version



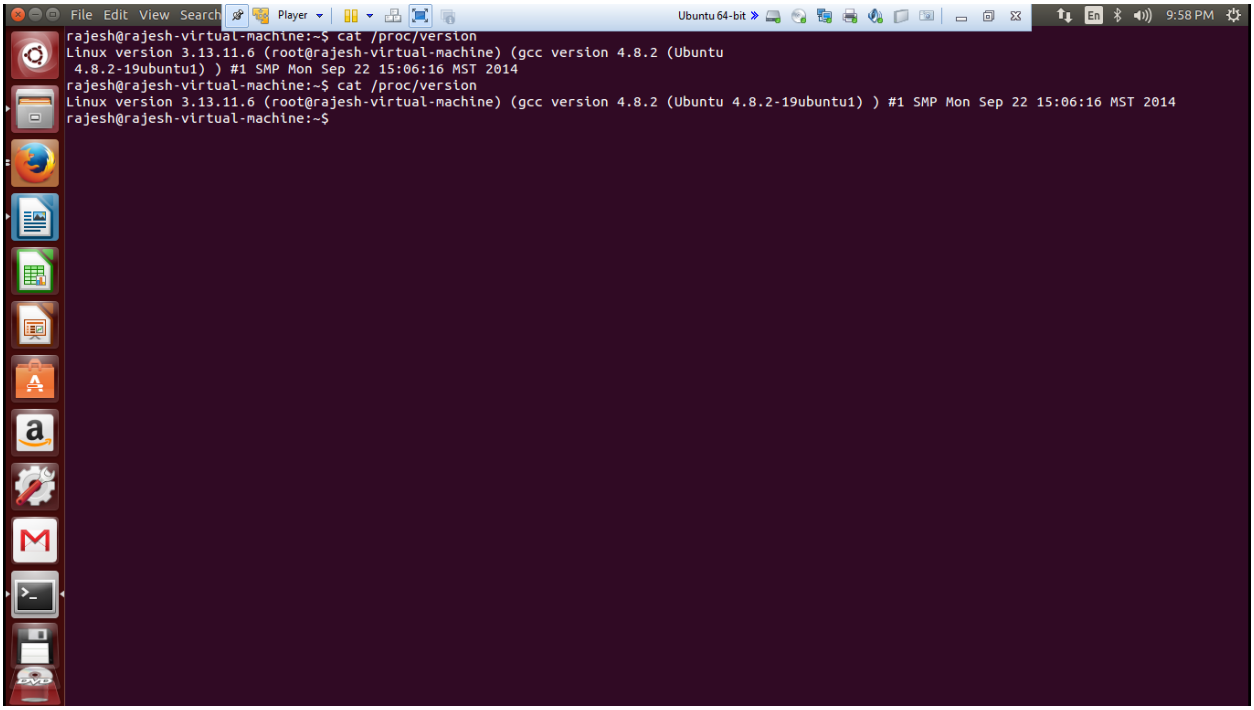
sudo dpkg -i ../ linux-image-3.13.11.6_1.0.cse536_amd64.deb



sudo shutdown -r now



cat /proc/version



Outcome and Experience:-

Here, we learned how to download the kernel source code as well as utilities needed to perform the compile and packaging of the kernel. The major experience of this step was to understand how to compile the source code of the kernel and making an installation image that we used to upgrade to the latest kernel.

Part III – Build and Debug .ko

In part III, our goal is to round out our skill set by building and installing a simple .ko and an application program that accesses the .ko as a device driver. For part III, we will be using the simplest of the driver types, a character I/O driver.

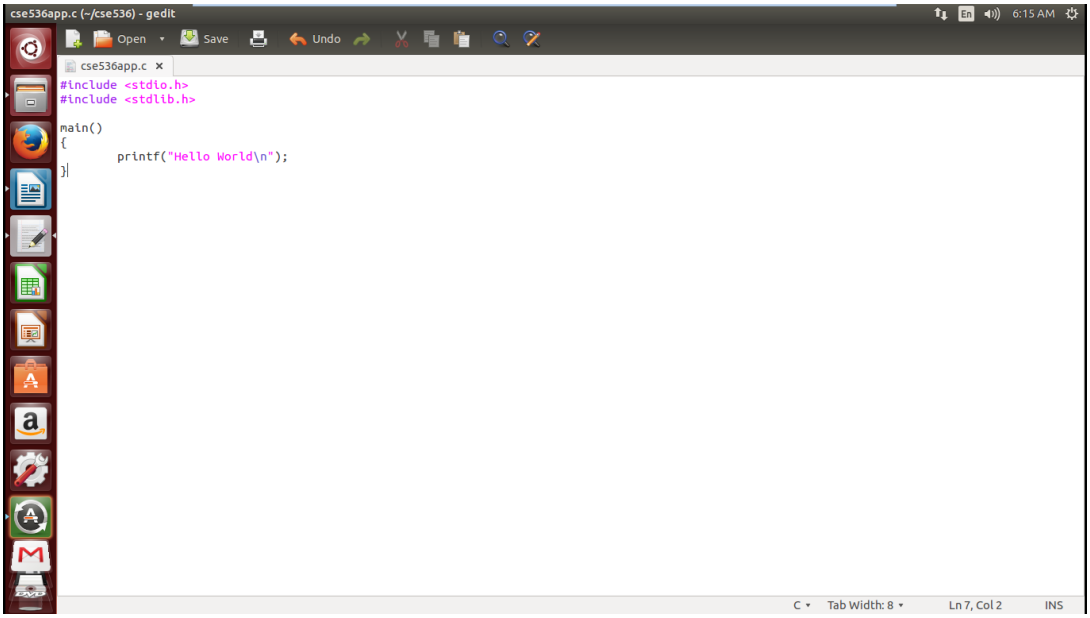
Application Name: - CSE536app

Kernel Module: - CSE5361.ko

This step is sub-divided into 5 steps.

Step 1 – <u>Compile the following program cse536app.c in our development directory (CSE536) on Ubuntu</u>

```
BEGIN COPY AFTER THIS LINE
/*
      Hello World
*/

#include <stdio.h>
#include <stdlib.h>

main()
{
      printf("Hello World\n");
}
END COPY BEFORE THIS LINE
```

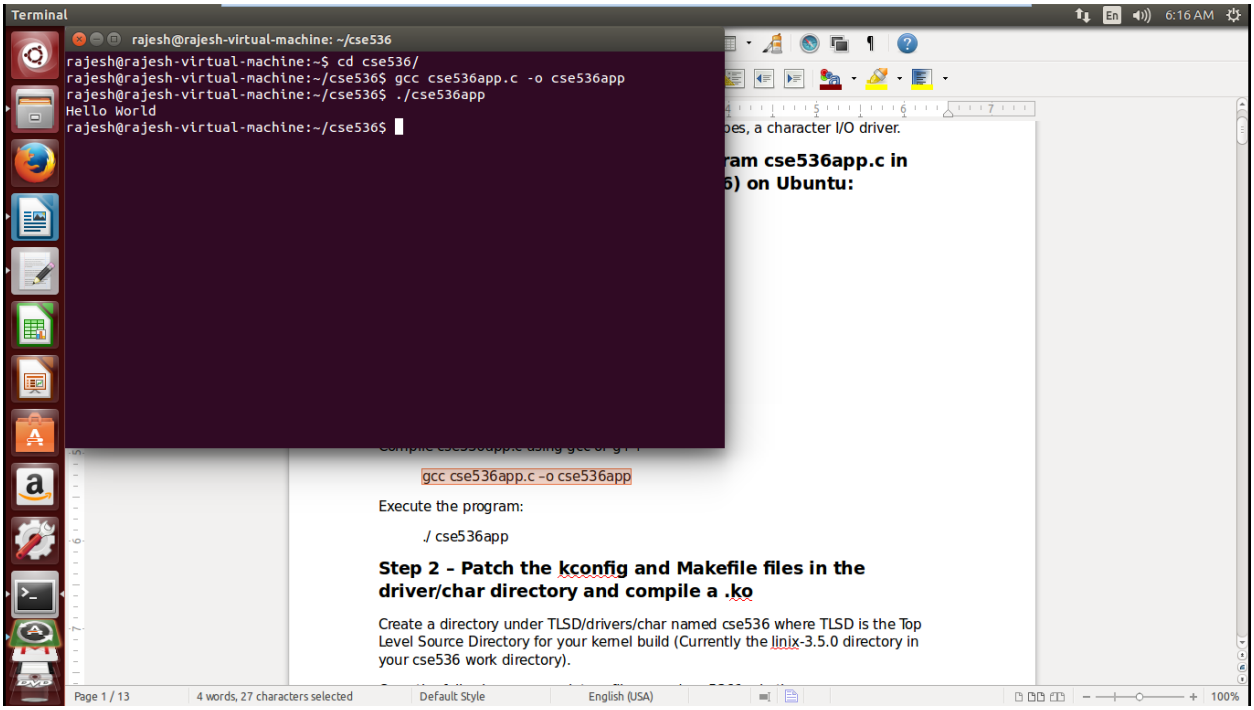


Compile cse536app.c using gcc or g++

gcc cse536app.c –o cse536app

Execute the program:

./ cse536app



Step 2 – Patch the kconfig and Makefile files in the driver/char directory and compile a .ko

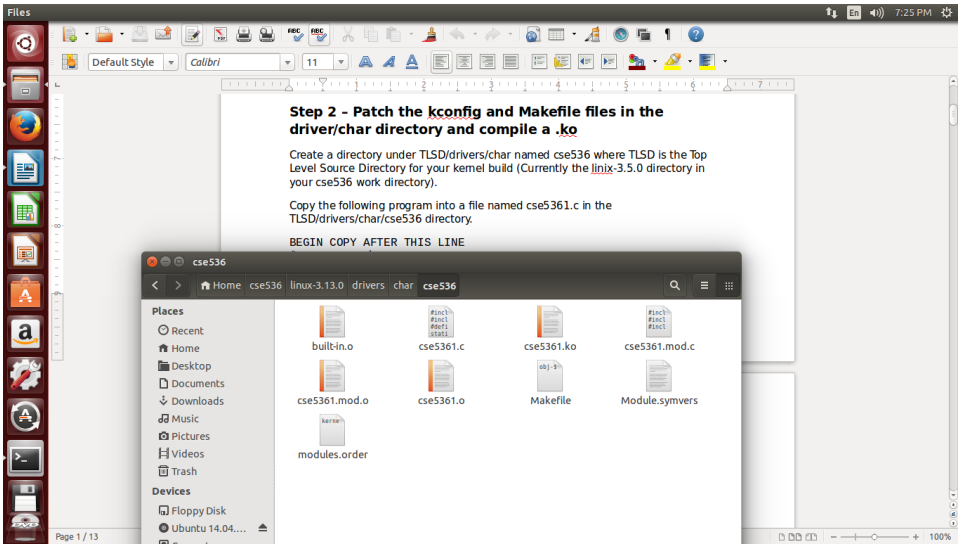
Create a directory under TLSD/drivers/char named cse536 where TLSD is the Top Level Source Directory for our kernel build (Currently the linix-3.5.0 directory in our cse536 work directory).

Copy the following program into a file named cse5361.c in the TLSD/drivers/char/cse536 directory.

BEGIN COPY AFTER THIS LINE

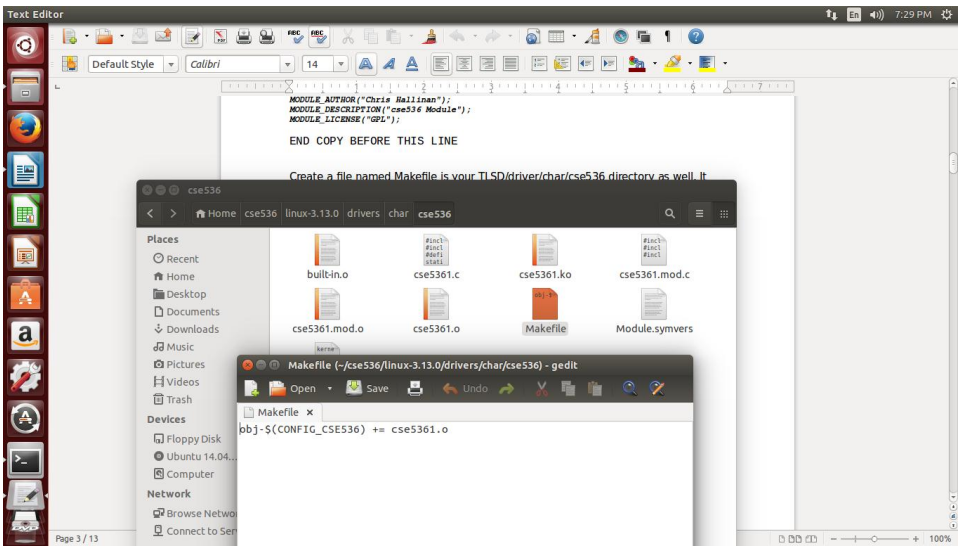
```
#include <linux/module.h>
#include <linux/fs.h>
#define CSE536_MAJOR 234
static int debug_enable = 0;
module_param(debug_enable, int, 0);
MODULE_PARM_DESC(debug_enable, "Enable module debug mode.");
struct file_operations cse536_fops;
static int cse536_open(struct inode *inode, struct file *file)
{
       printk("cse536_open: successful\n");
       return 0;
}
static int cse536_release(struct inode *inode, struct file *file)
{
       printk("cse536_release: successful\n");
       return 0;
}
static ssize_t cse536_read(struct file *file, char *buf, size_t count,
loff_t *ptr)
{
       printk("cse536_read: returning zero bytes\n");
       return 0;
}
static ssize_t cse536_write(struct file *file, const char *buf,
size_t count, loff_t * ppos)
{
       printk("cse536_write: accepting zero bytes\n");
       return 0;
}
static long cse536_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{
       printk("cse536_ioctl: cmd=%d, arg=%ld\n", cmd, arg);
       return 0;
}
static int __init cse536_init(void)
{
       int ret;
       printk("cse536 module Init - debug mode is %s\n",
       debug_enable ? "enabled" : "disabled");
       ret = register_chrdev(CSE536_MAJOR, "cse5361", &cse536_fops);
       if (ret < 0) {
              printk("Error registering cse536 device\n");
       goto cse536_fail1;
       }
       printk("cse536: registered module successfully!\n");
       /* Init processing here... */
       return 0;
       cse536_fail1:
       return ret;
}
static void __exit cse536_exit(void)
{
       unregister_chrdev(CSE536_MAJOR, "cse5361");
       printk("cse536 module Exit\n");
```

```
}
struct file_operations cse536_fops = {
       owner: THIS_MODULE,
       read: cse536_read,
       write: cse536_write,
       unlocked_ioctl: cse536_ioctl,
       open: cse536_open,
       release: cse536_release,
};
module_init(cse536_init);
module_exit(cse536_exit);

MODULE_AUTHOR("Chris Hallinan");
MODULE_DESCRIPTION("cse536 Module");
MODULE_LICENSE("GPL");

END COPY BEFORE THIS LINE
```



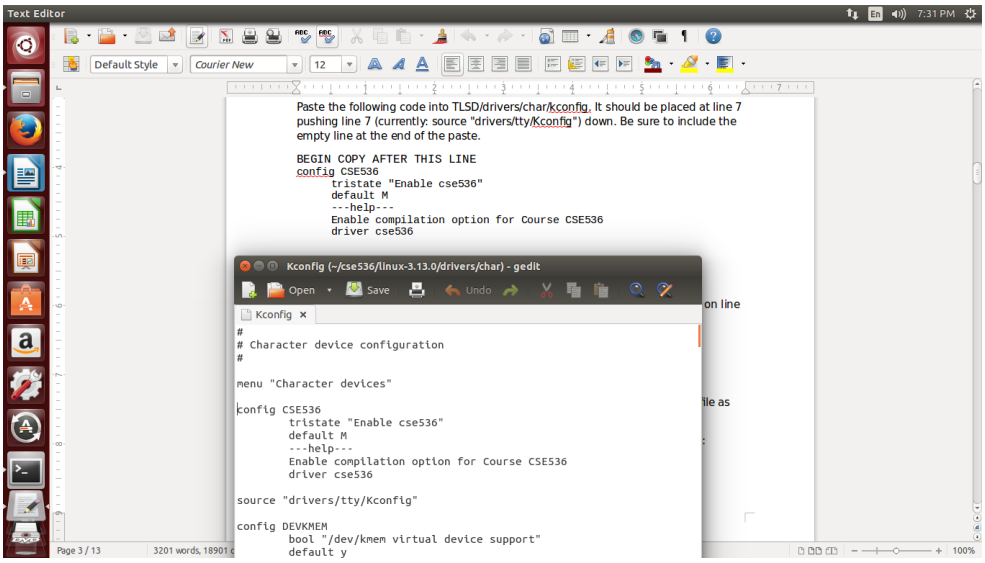
Create a file named Makefile is our TLSD/driver/char/cse536 directory as well. It should contain the line:

```
BEGIN COPY AFTER THIS LINE
obj-$(CONFIG_CSE536) += cse5361.o
END COPY BEFORE THIS LINE
```



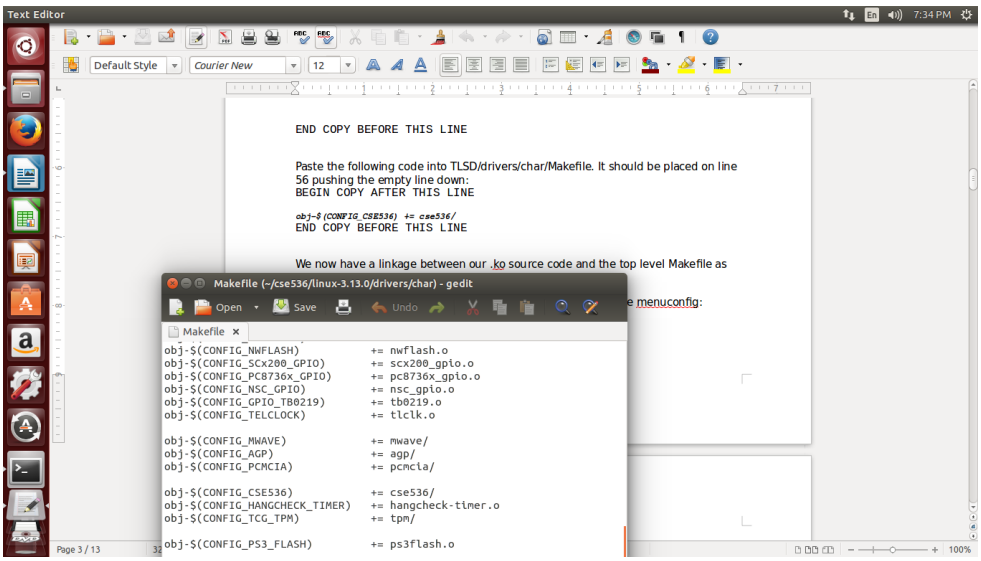
Paste the following code into TLSD/drivers/char/kconfig. It should be placed at line 7 pushing line 7 (currently: source "drivers/tty/Kconfig") down. Be sure to include the empty line at the end of the paste.

```
BEGIN COPY AFTER THIS LINE
config CSE536
	tristate "Enable cse536"
	default M
	---help---
	Enable compilation option for Course CSE536
	driver cse536

END COPY BEFORE THIS LINE
```



Paste the following code into TLSD/drivers/char/Makefile. It should be placed on line 56 pushing the empty line down:

```
BEGIN COPY AFTER THIS LINE
obj-$(CONFIG_CSE536) += cse536/
END COPY BEFORE THIS LINE
```



We now have a linkage between our .ko source code and the top level Makefile as well as the top level kconfig file.

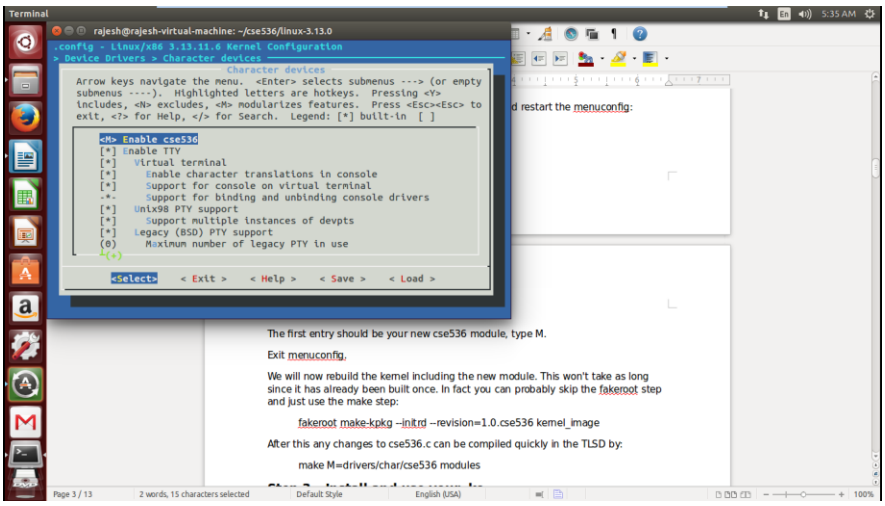
Change working directory back to the TLSD and restart the menuconfig:

make menuconfig

Locate Device Drivers in the menu and hit return

Locate Character Devices and hit return

The first entry should be your new cse536 module, type M.



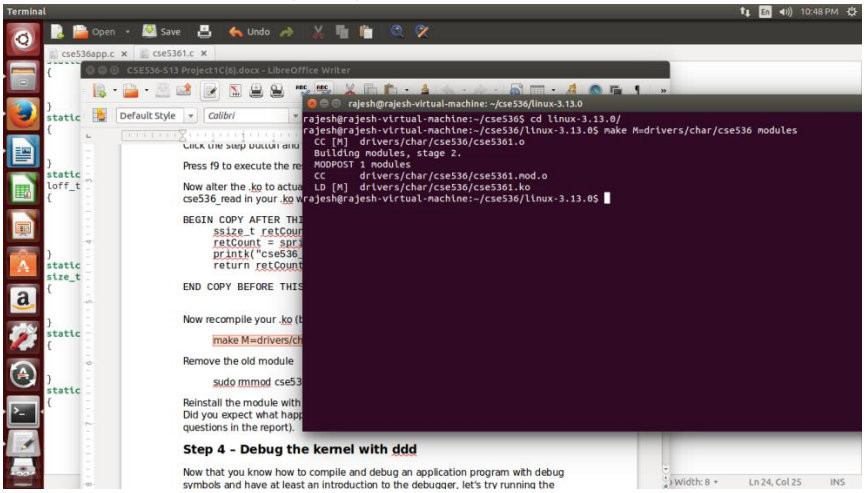
Exit menuconfig.

We will now rebuild the kernel including the new module. This won't take as long since it has already been built once. In fact we can probably skip the fakeroot step and just use the make step:

fakeroot make-kpkg --initrd --revision=1.0.cse536 kernel_image

After this any changes to cse536.c can be compiled quickly in the TLSD by:

make M=drivers/char/cse536 modules



Step 3 – Install and use your .ko

To see the output from our .ko module printk function we can use:

dmesg | grep cse536

after the fact or you can see it as it happens by starting xconsole from the CLI and lock it to the launcher for future use:

xconsole&

Alternatively you can start a background display of the prink output using the following command:

tailf /var/log/syslog |grep -i cse536 &

which shows additions to the log as they happen.

The xconsole window can be sized by grabbing the sides and pulling them with your mouse. The left bar allows you to scroll up or down using the right and left mouse botton clicks.

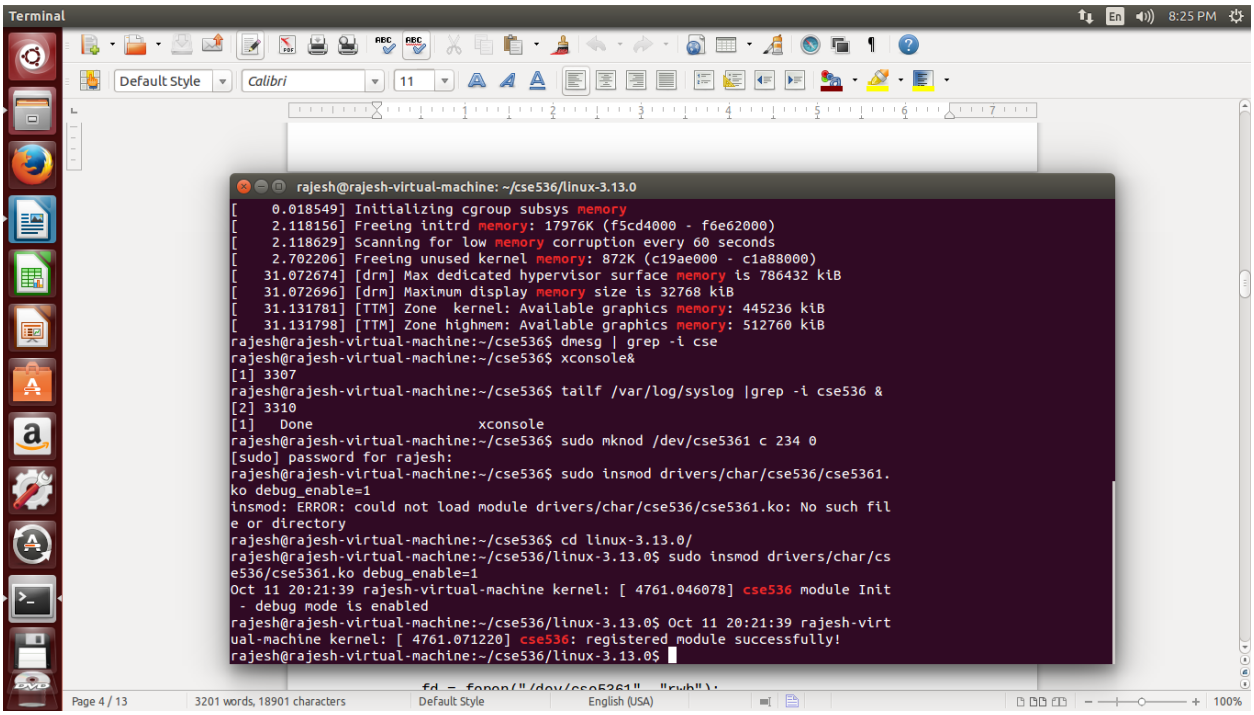
To install your .ko on your local system enter the following commands:

This first command creates a device to allow your programs to read and write through the driver:

sudo mknod /dev/cse5361 c 234 0

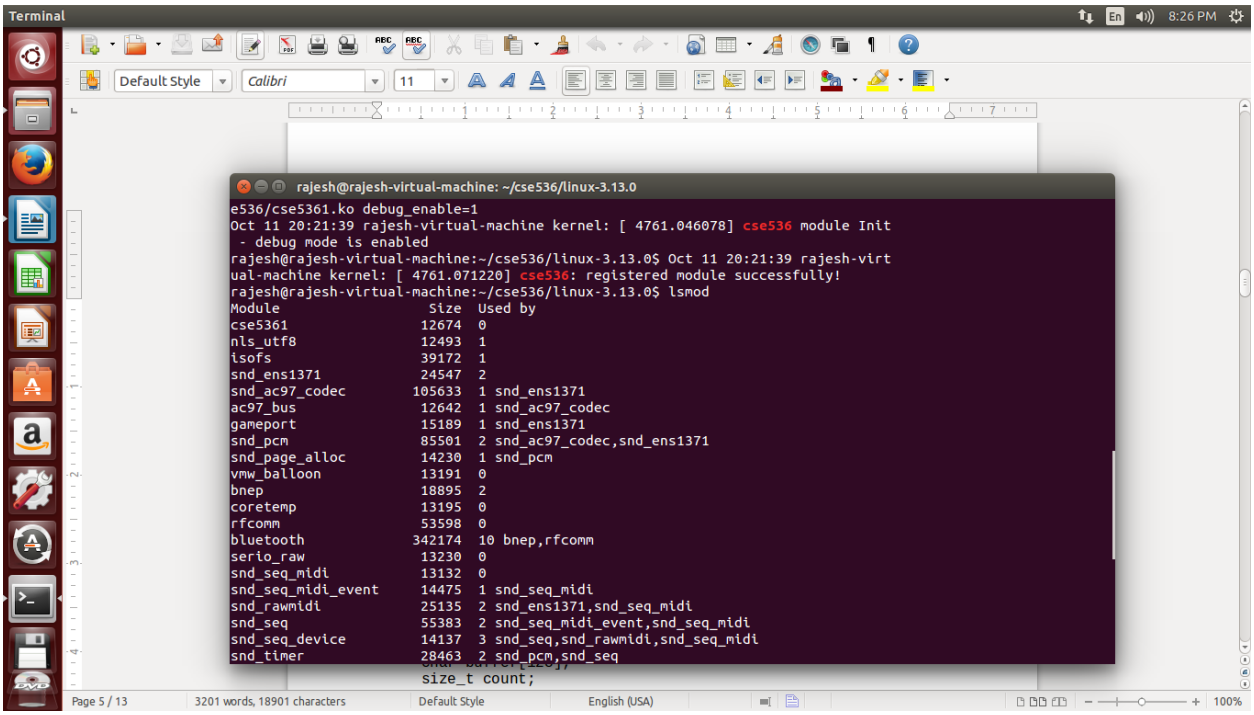
This command actually installs the .ko module in the active kernel (I assume you are in the TLSD):

sudo insmod drivers/char/cse536/cse5361.ko debug_enable=1



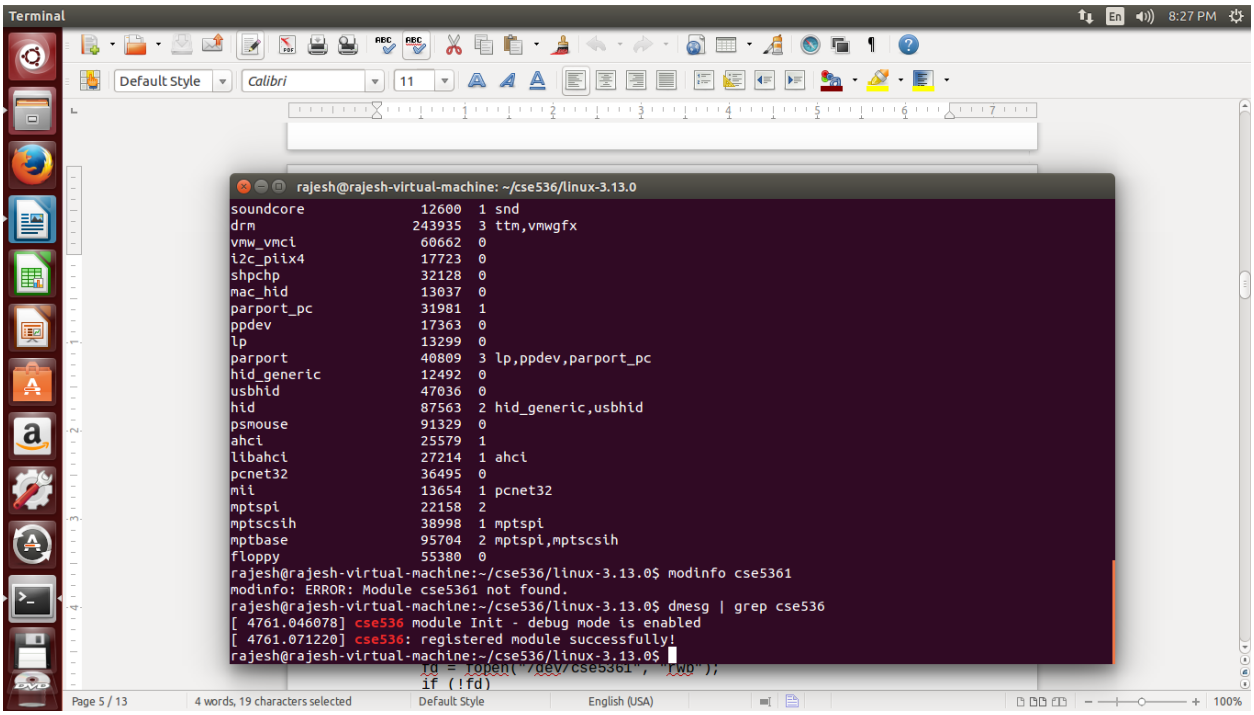
This command lists the currently loaded modules so you can verify that the module is correctly loaded:

Lsmod



This command quires the loaded module to display information about it:

modinfo cse5361



This command displays the output from the printk statements in the module from the system log:

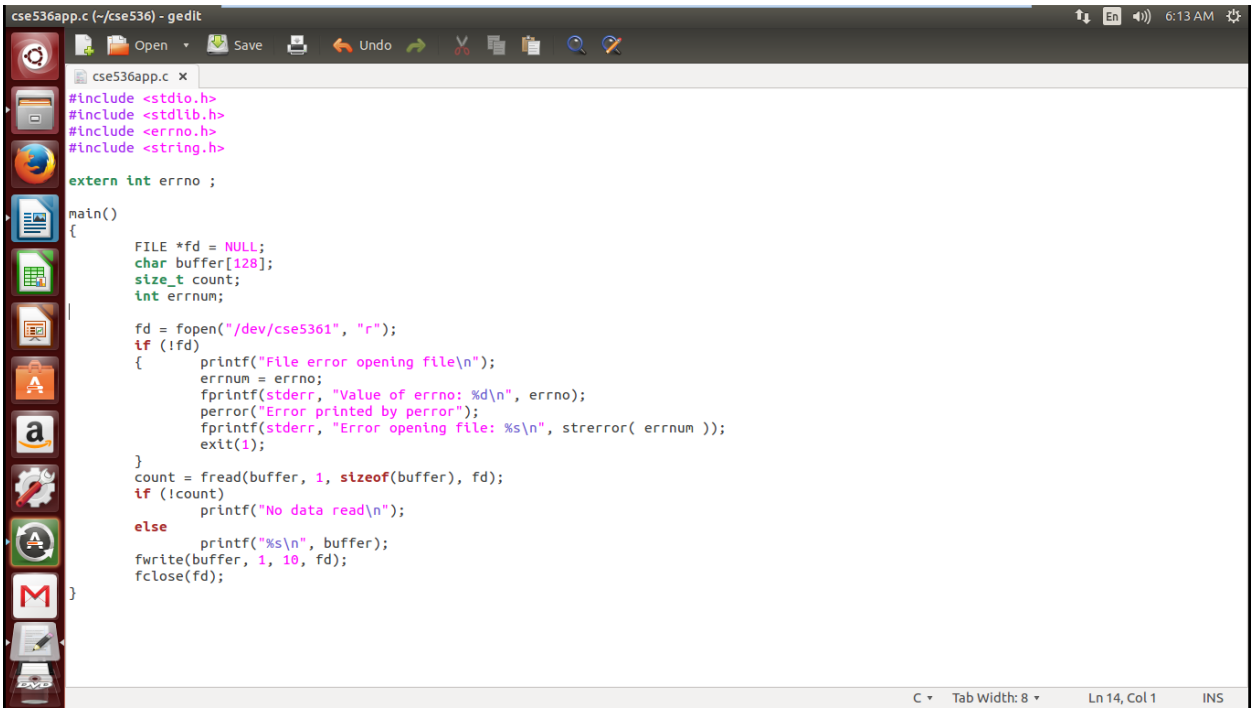
dmesg | grep cse536

This command removes the module so you can change it and reload it if necessary:

sudo rmmod cse5361

Now modify your cse536app program by replacing the printf statement with:

```
BEGIN COPY AFTER THIS LINE
      FILE *fd = NULL;
      char buffer[128];
      size_t count;

      fd = fopen("/dev/cse5361", "rwb");
      if (!fd)
      {     printf("File error opening file\n");
            exit(1);
      }
      count = fread(buffer, 1, sizeof(buffer), fd);
      if (!count)
            printf("No data read\n");
      else
            printf("%s\n", buffer);
      fwrite(buffer, 1, 10, fd);
      fclose(fd);
END COPY BEFORE THIS LINE
```



Compile the program with debugging symbols and run it:

gcc cse536app.c –o cse536app –g

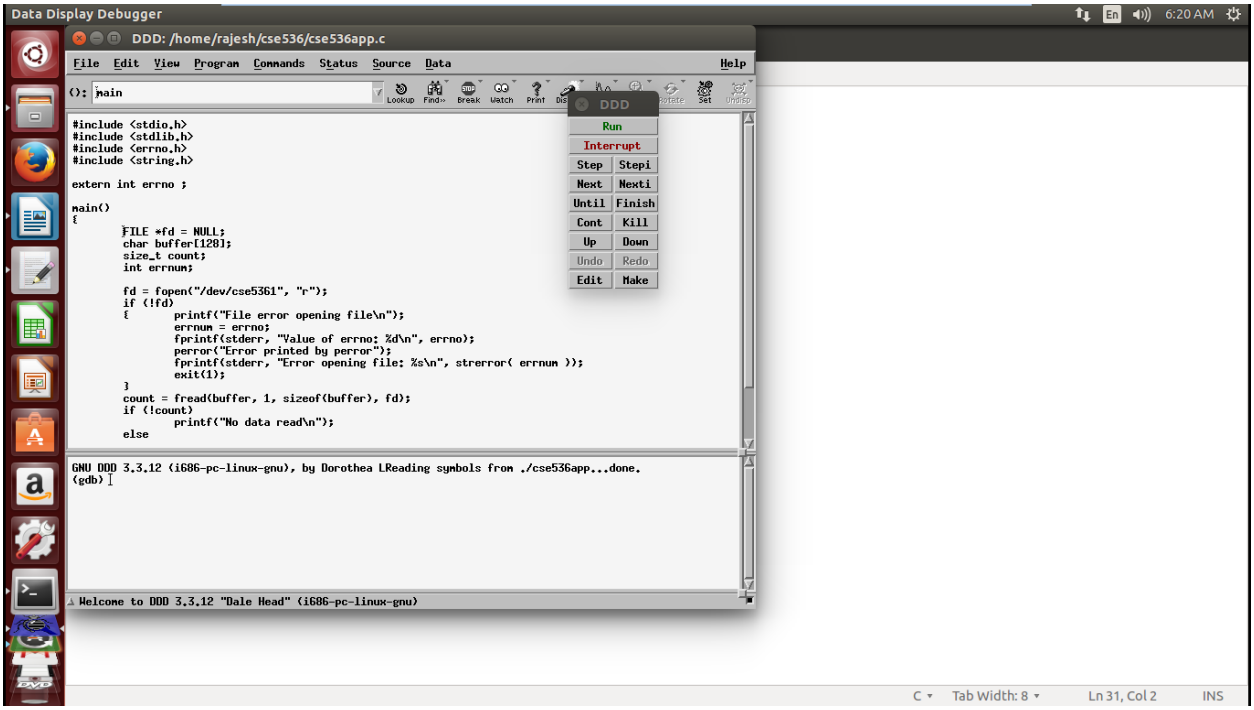
./cse536app

If you haven't done so already download ddd:

sudo apt-get install ddd

Now run the program in the debugger. ddd is the GUI for gdb (you can always just use gdb instead of ddd):

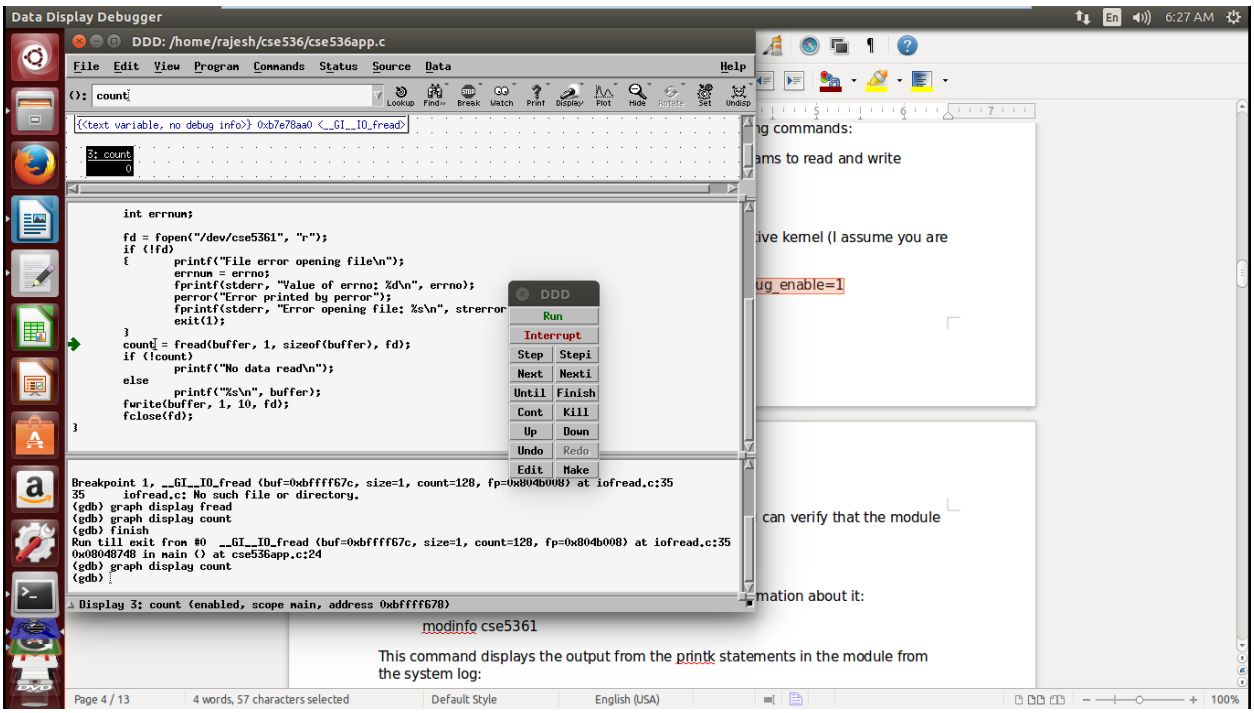
ddd ./cse536app



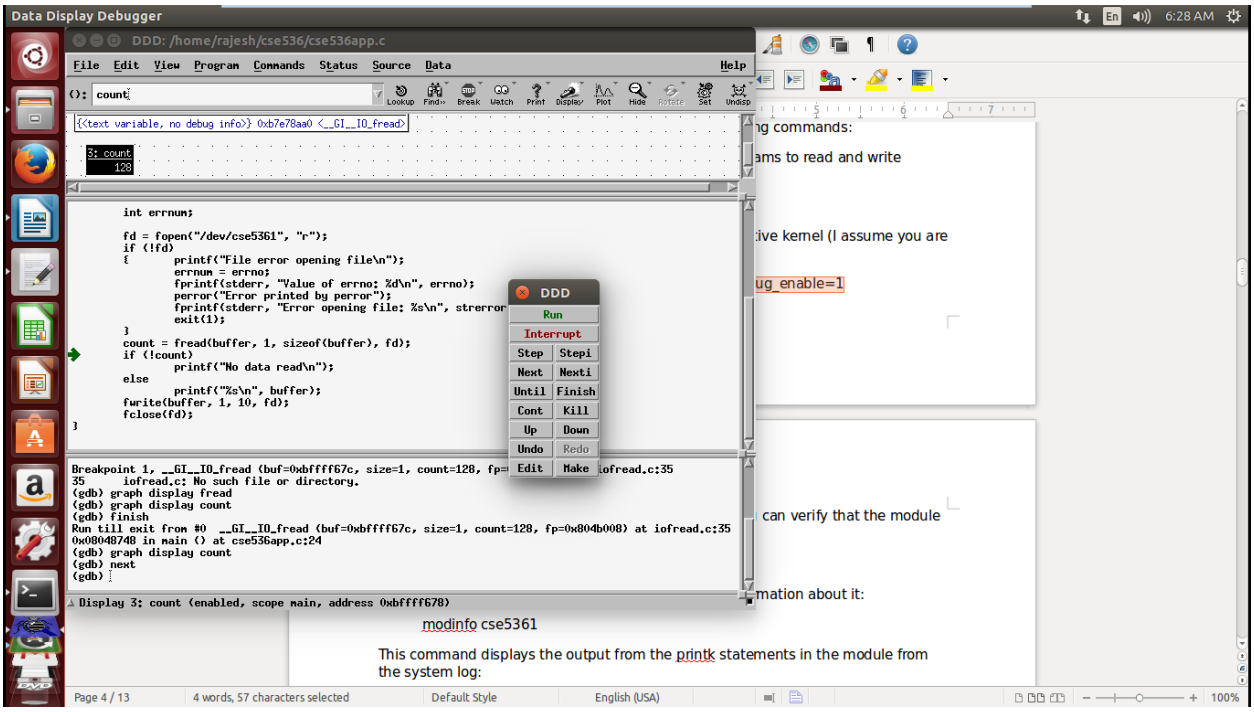
Place the cursor in front of the fread line of code and click the stop sign to set a breakpoint (assuming you are using ddd).

Click the run button and the program will stop at the break point.

Click on the count variable on line 19 and then click the display button. It will show you the value of count.



Click the step button and watch the value of count change.



Press f9 to execute the rest of the program.

Now alter the .ko to actually return some data on a read. Replace the contents of cse536_read in your .ko with:

```
BEGIN COPY AFTER THIS LINE
      ssize_t retCount;
      retCount = sprintf(buf, "cse536");
      printk("cse536_read: returning %d bytes\n", retCount);
      return retCount;
END COPY BEFORE THIS LINE
```

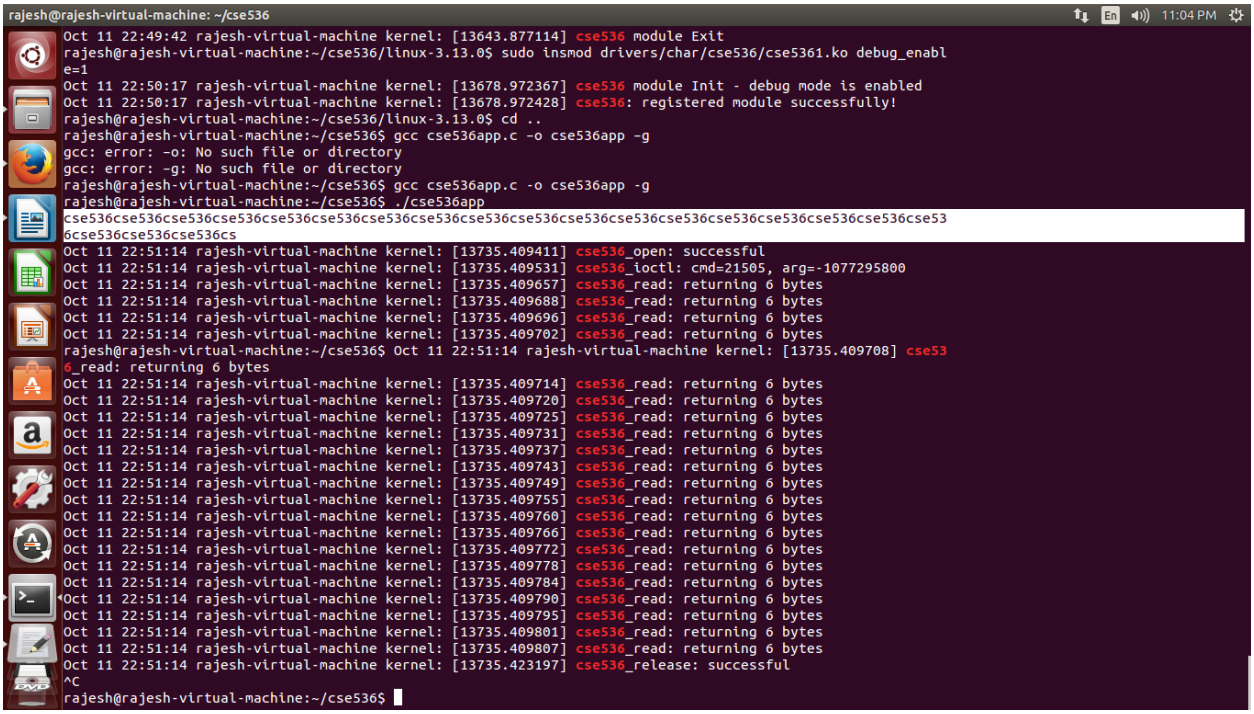
Now recompile your .ko (be sure you are in the TLSD):

make M=drivers/char/cse536 modules

Remove the old module

sudo rmmod cse5361

Reinstall the module with the changes and run your program again. What changed? Did you expect what happened? Why did it happen?



Answer:-

Here, output is 128 characters consisting of string “cse536” repetitively. Previously the output was ‘No data read’.

Because, kernel module put “cse536” on our device ‘cse5361’. We read in our cse536app that device using file pointer till 128 characters. Previously, read count was 0 which is now 128.

Step 4 – Debug the kernel with ddd

Now that you know how to compile and debug an application program with debug symbols and have at least an introduction to the debugger, let's try running the debugger on the kernel.

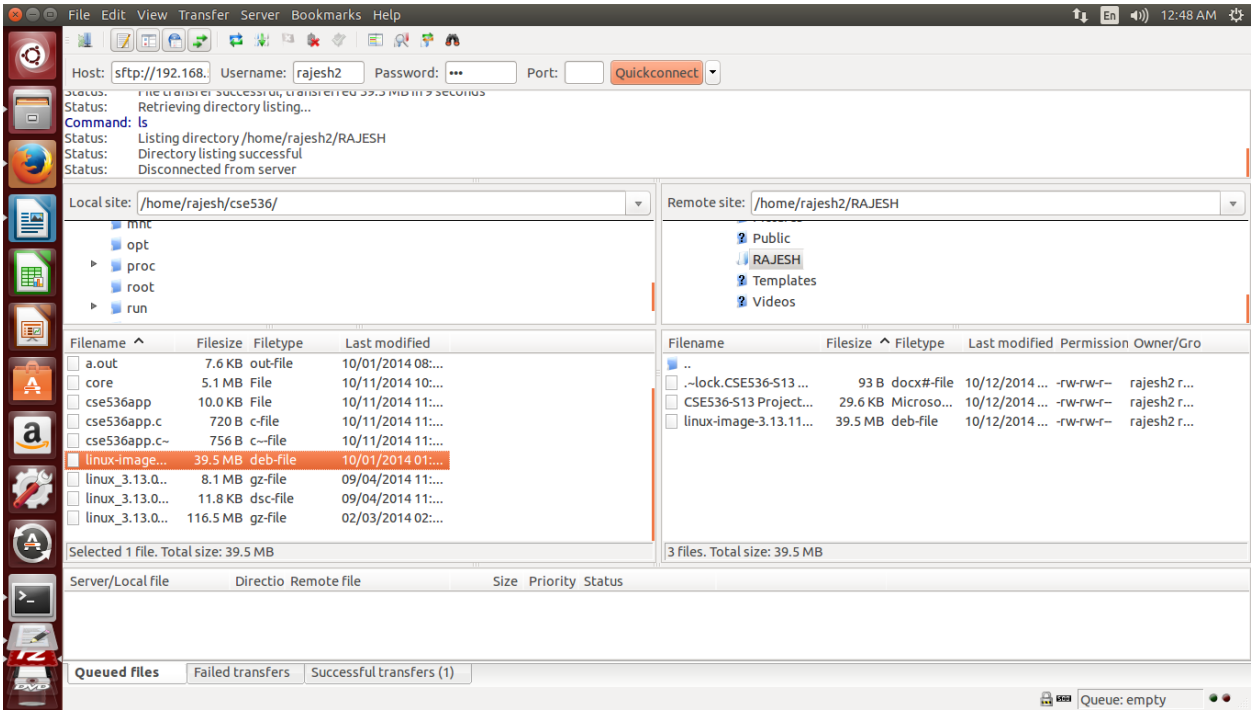
If you think about it you will realize that the debugger stops what it is executing to allow you to step through each line of code and analyze the variable values in the process. If the debugger does this to the kernel what happens to the programs on the computer including the debugger? Because everything stops we cannot debug the kernel on the same machine that is hosting the ddd program.

To solve this problem we must debug the kernel on a target machine apart from the machine hosting the debugger. We will do this by creating another virtual machine and connect them via the network or a named pipe. Two virtual machines residing on the same host machine can communicate via a named pipe. Our first Ubuntu VM will act as the debug VM and we will create another Ubuntu VM to remotely debug the kernel which we will refer to as the target VM.

Create a target VM and install Ubuntu. Perform all updates and upgrades. You may then use the .deb file in the cse536 directory that was generated by compiling the TLSD to install our target OS on the new VM.

Once the target VM is up and running there are a number of mechanisms that can be used to transfer files between the VMs. Installing an ssh server on the target VM and using filezilla will work. On your debug VM install filezilla:

Debug: sudo apt-get install filezilla



Install an ssh server on your target VM:

Target: sudo apt-get install openssh-server

Now you can start filezilla on your debug host VM and transfer the .deb install file, .ko module and application programs as needed. You will need to look up the ip number of your target VM or set it statically. You can use:

Target: ifconfig

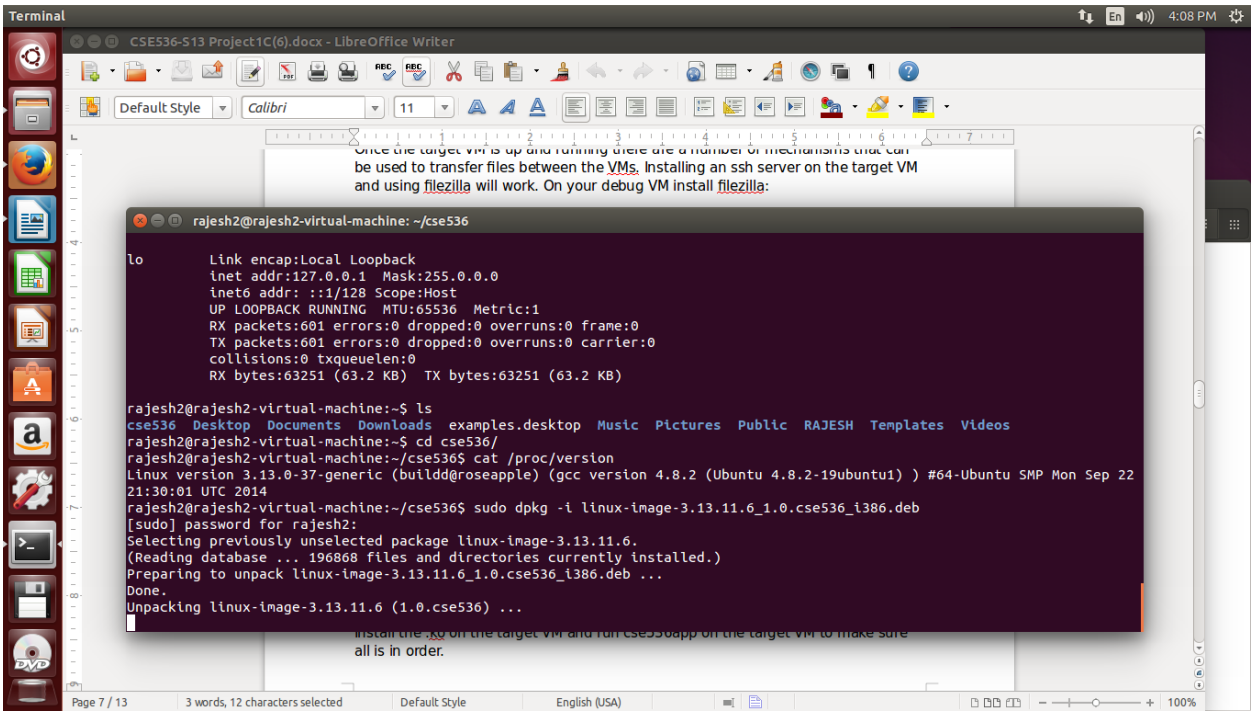
to discover the ip number. Use port 22 in filezilla where it asks for a port number.

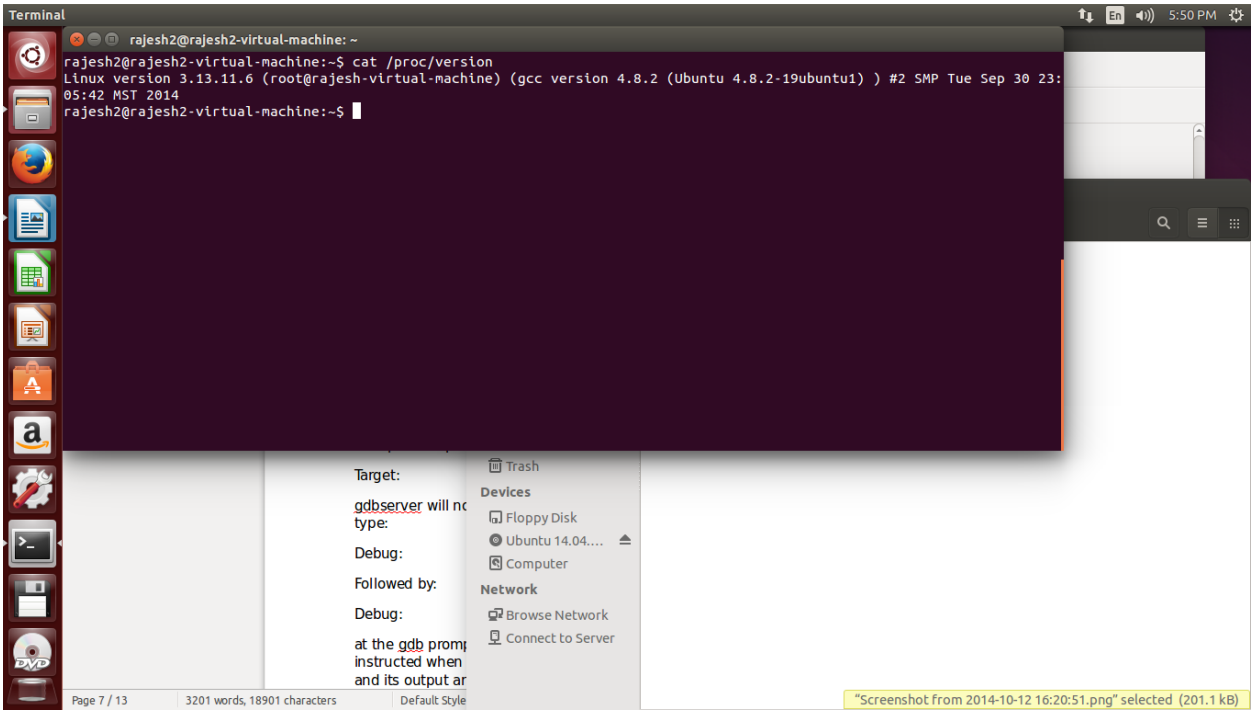
Install the compiled kernel from the debug VM onto the target VM:

Target: sudo dpkg -i linux-image-3.5.7.2_1.0.cse536_i386.deb

Reboot the target VM and verify the correct installation using:

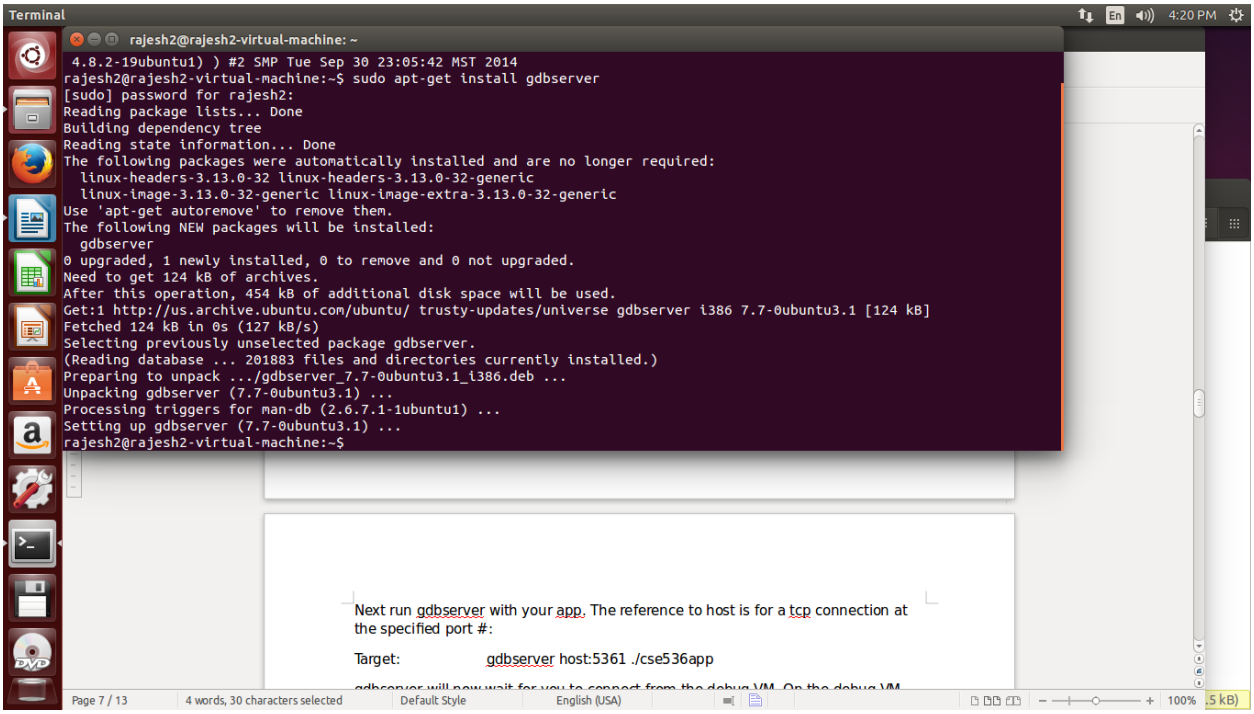
Target: cat /proc/version





Now let's try debugging a remote program before we take on the kernel. Download gdbserver to the target VM:

Target: sudo apt-get install gdbserver

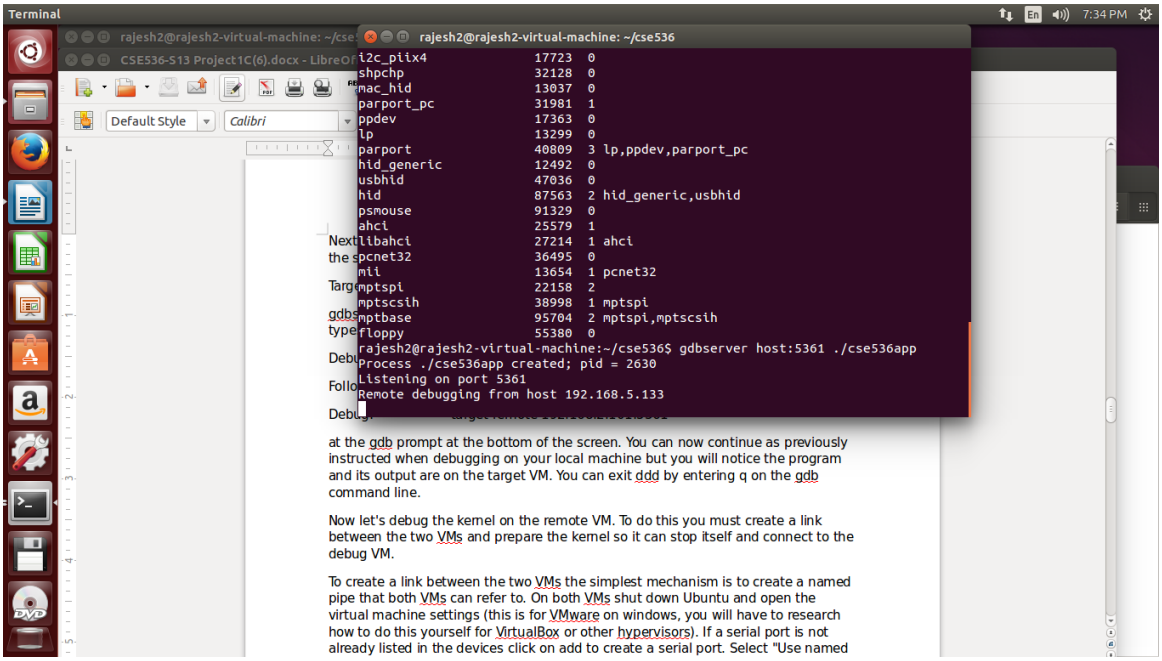


Install the .ko on the target VM and run cse536app on the target VM to make sure all is in order.

Next run gdbserver with your app. The reference to host is for a tcp connection at the specified port #:

Target: gdbserver host:5361 ./cse536app

gdbserver will now wait for you to connect from the debug VM.

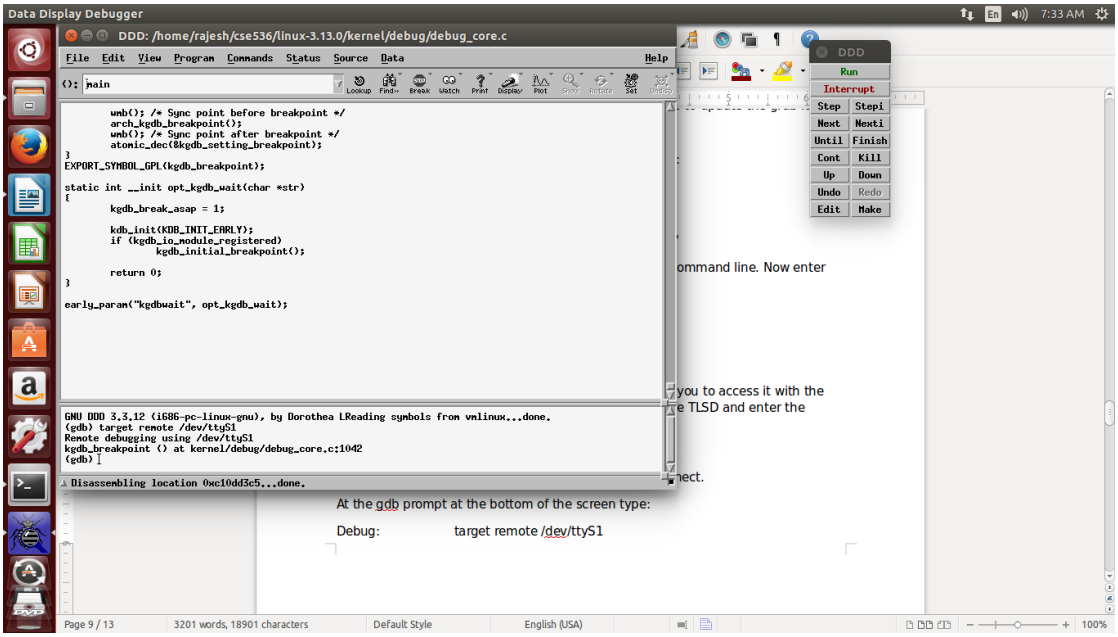


On the debug VM type:

Debug: ddd ./cse536app

Followed by:

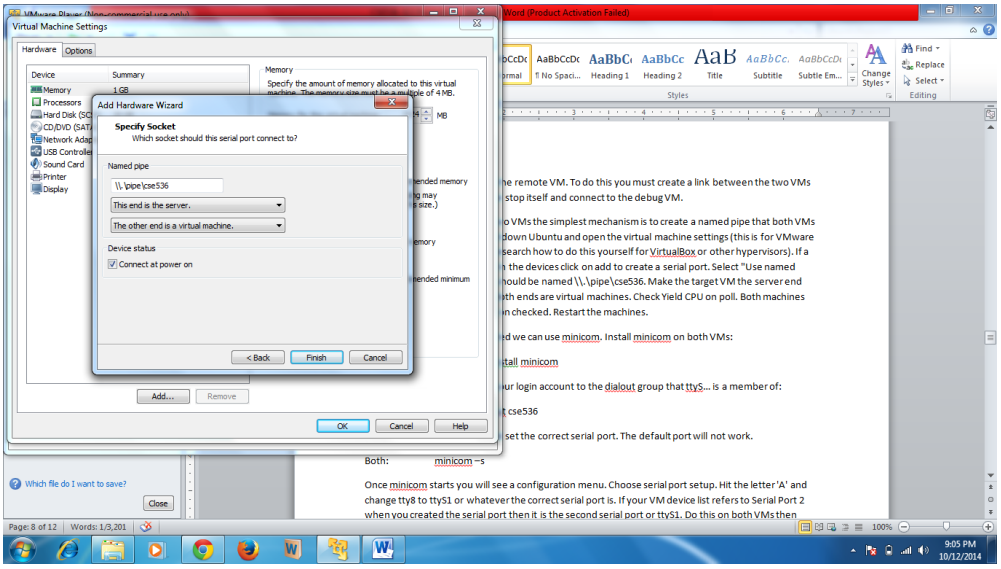
Debug: target remote 192.168.2.101:5361



at the gdb prompt at the bottom of the screen. You can now continue as previously instructed when debugging on your local machine but you will notice the program and its output are on the target VM. You can exit ddd by entering q on the gdb command line.

Now let's debug the kernel on the remote VM. To do this you must create a link between the two VMs and prepare the kernel so it can stop itself and connect to the debug VM.

To create a link between the two VMs the simplest mechanism is to create a named pipe that both VMs can refer to. On both VMs shut down Ubuntu and open the virtual machine settings (this is for VMware on windows, you will have to research how to do this yourself for VirtualBox or other hypervisors). If a serial port is not already listed in the devices click on add to create a serial port. Select "Use named pipe:". The pipe on both VMs should be named \\.\pipe\cse536. Make the target VM the server end and the debug VM the client. Both ends are virtual machines. Check Yield CPU on poll. Both machines should have connect at power on checked. Restart the machines.



To verify that they are connected we can use minicom. Install minicom on both VMs:

Both: sudo apt-get install minicom

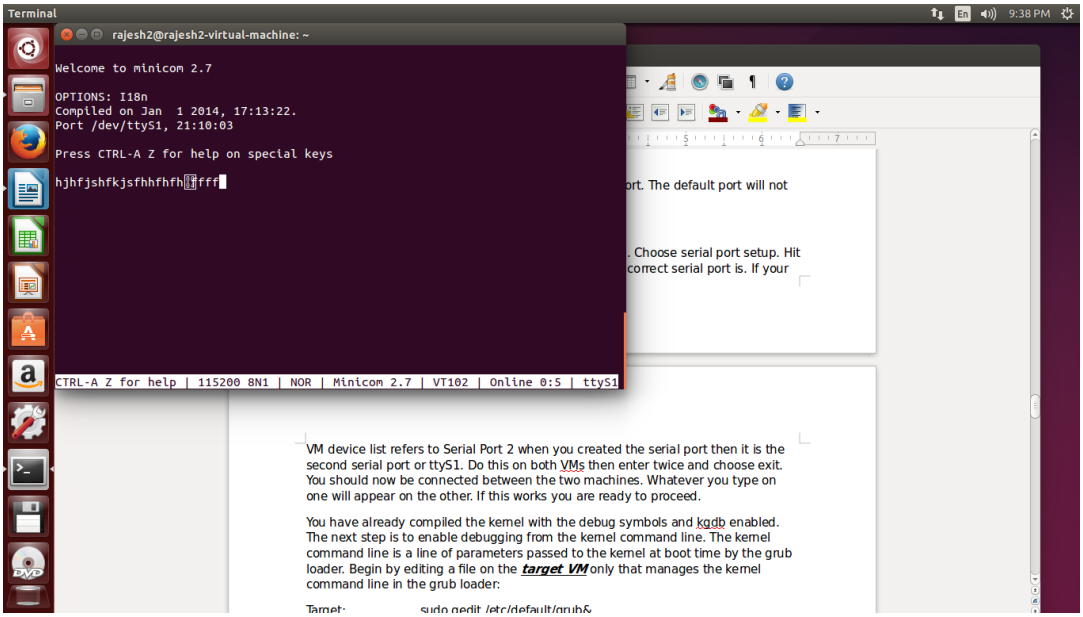
Before you use miinicom add your login account to the dialout group that ttyS… is a member of:

sudo useradd –G dialout cse536

Start minicom on both VMs and set the correct serial port. The default port will not work.

Both: minicom –s

Once minicom starts you will see a configuration menu. Choose serial port setup. Hit the letter 'A' and change tty8 to ttyS1 or whatever the correct serial port is. If your VM device list refers to Serial Port 2 when you created the serial port then it is the second serial port or ttyS1. Do this on both VMs then enter twice and choose exit. You should now be connected between the two machines. Whatever you type on one will appear on the other. If this works you are ready to proceed.



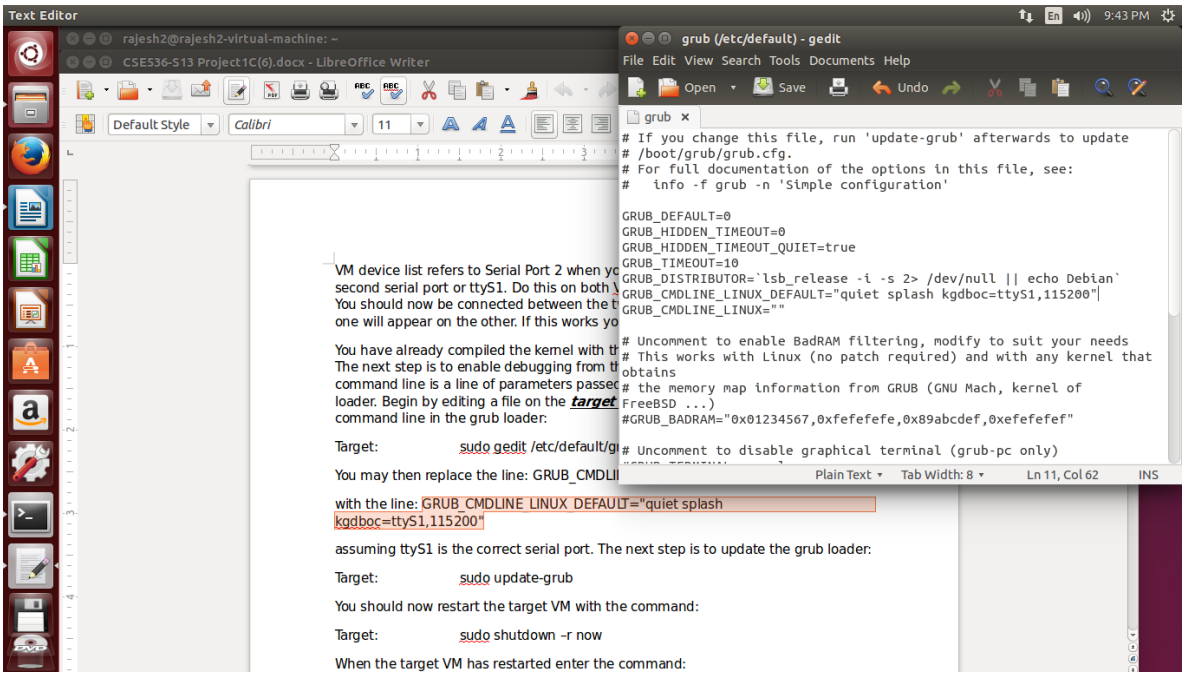
You have already compiled the kernel with the debug symbols and kgdb enabled. The next step is to enable debugging from the kernel command line. The kernel command line is a line of parameters passed to the kernel at boot time by the grub loader. Begin by editing a file on the ***target VM*** only that manages the kernel command line in the grub loader:

Target: sudo gedit /etc/default/grub&

You may then replace the line: GRUB_CMDLINE_LINUX_DEFAULT="quiet splash"

with the line: GRUB_CMDLINE_LINUX_DEFAULT="quiet splash kgdboc=ttyS1,115200"

assuming ttyS1 is the correct serial port.



The next step is to update the grub loader:

Target: sudo update-grub

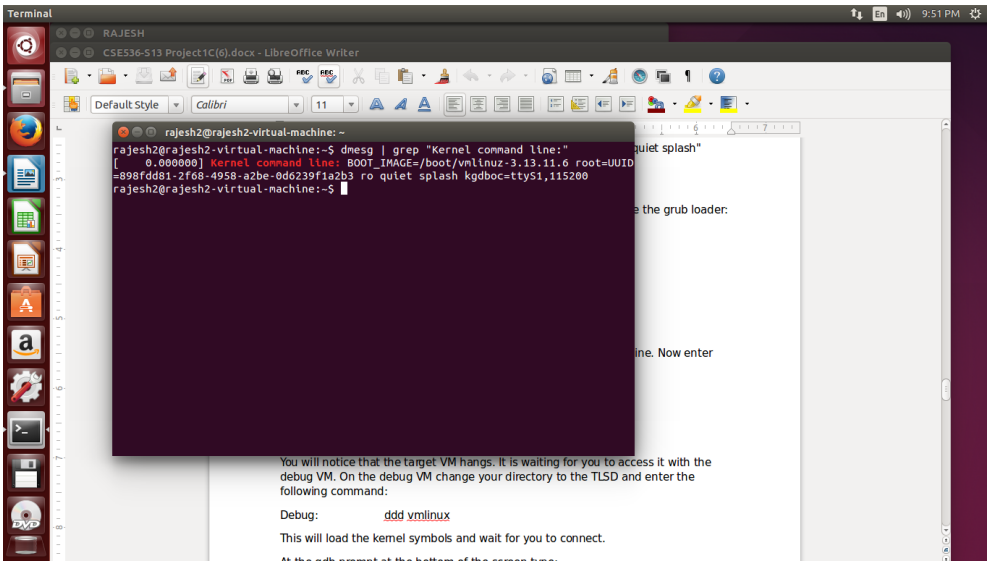
You should now restart the target VM with the command:

Target: sudo shutdown –r now

When the target VM has restarted enter the command:

Target: dmesg | grep "Kernel command line:"

to verify that the kernel restarted with your new kernel command line.

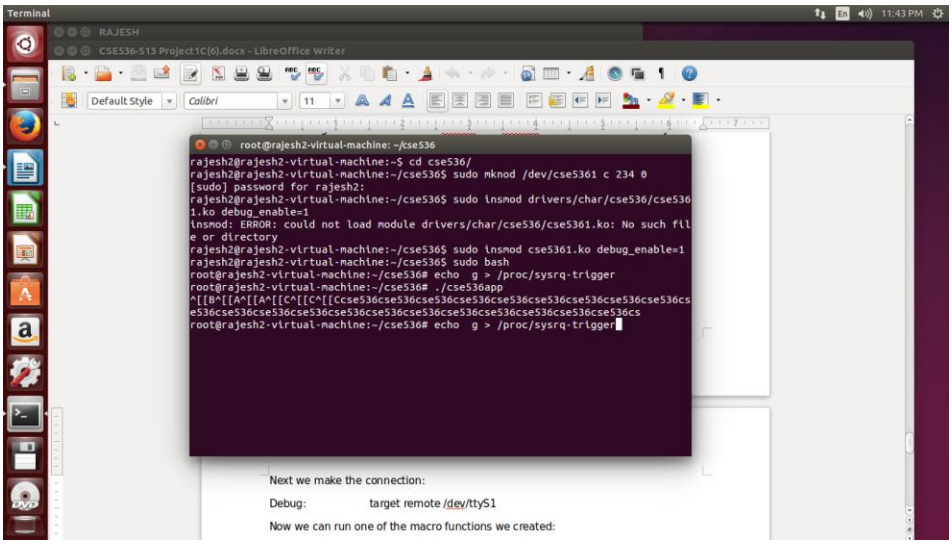


Now enter the command:

Target: sudo bash

followed by:

Target: echo g > /proc/sysrq-trigger



You will notice that the target VM hangs. It is waiting for you to access it with the debug VM. On the debug VM change your directory to the TLSD and enter the following command:

Debug: ddd vmlinux

This will load the kernel symbols and wait for you to connect.

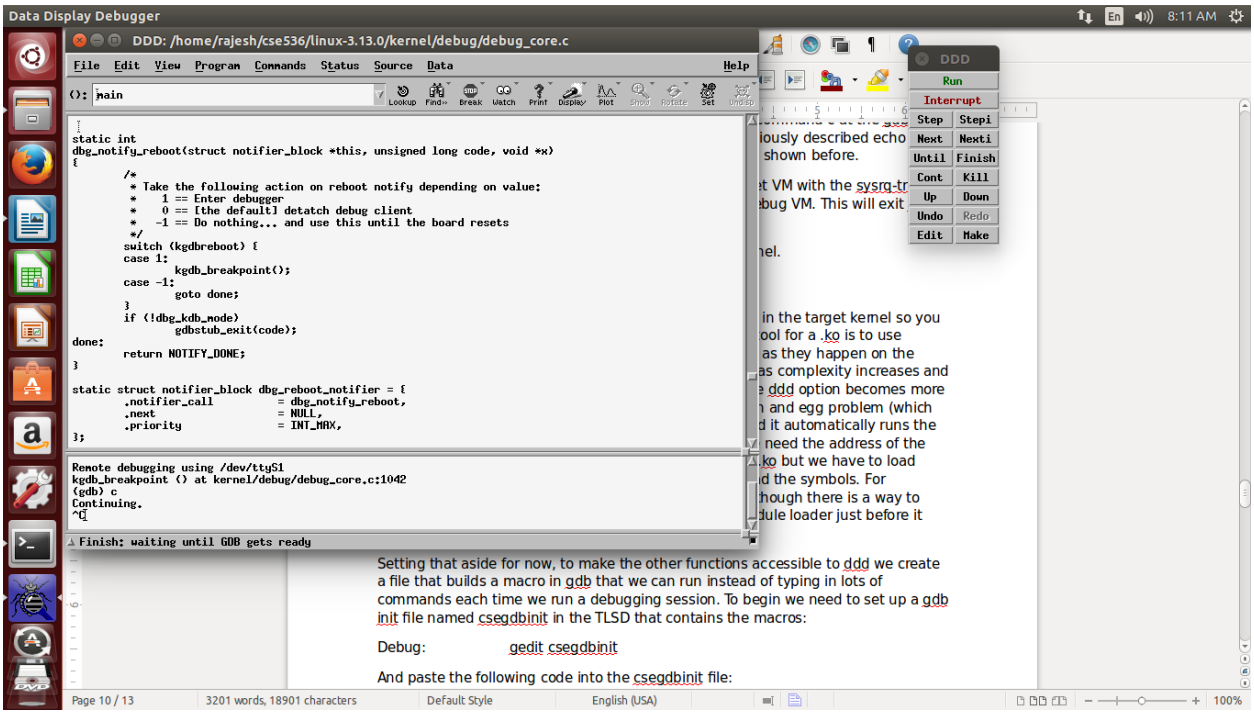
At the gdb prompt at the bottom of the screen type:

Debug: target remote /dev/ttyS1

using the correct serial port if it is not ttyS1. The debug VM and the target VM should now be connected. You can now set a breakpoint on any symbol in the kernel and restart the remote kernel by entering the continue command c at the gdb prompt. To stop the remote kernel any time use the previously described echo g > /proc/sysrq-trigger command in a root bash session as shown before.

When ending your ddd session you should stop the target VM with the sysrq-trigger and then enter q and return at the gdb prompt on the debug VM. This will exit ddd and resume the remote kernel.

Feel free to experiment and look around the running kernel.



Step 5 – Dubug your .ko

The last step in this project is to access your running .ko in the target kernel so you can debug it using ddd. Clearly the simplest debugging tool for a .ko is to use imbedded printk statements and view them on xconsole as they happen on the debug machine (no target machine required). However, as complexity increases and the number of variables we want to display increases the ddd option becomes more attractive. To access the .ko using ddd we have a chicken and egg problem (which came first, the chicken or the egg?). When a .ko is loaded it automatically runs the init function of the module. To debug the init function we need the address of the loaded .ko in order to correctly load the symbols for the .ko but we have to load the .ko and run the init function to get the address to load the symbols. For simplicity we will use printk to debug the init function although there is a way to make it work with ddd by setting a breakpoint in the module loader just before it calls init.

Setting that aside for now, to make the other functions accessible to ddd we create a file that builds a macro in gdb that we can run instead of typing in lots of commands each time we run a debugging session. To begin we need to set up a gdb init file named csegdbinit in the TLSD that contains the macros:

Debug: gedit csegdbinit

And paste the following code into the csegdbinit file:

```
BEGIN COPY AFTER THIS LINE
define lsmod
      printf "Address\t\t.text\t\tModule\n"
      set $m=(struct list_head *)&modules
      set $done=0
      set $count=$arg0
      while ( $count && !$done )
            # list_head is 4-bytes into struct module
            set $mp=(struct module *)((char *)$m->next - (char *)4)
            printf "0x%08X\t0x%08x\t%s\n", $mp, $mp->module_core, $mp->name
            if ( $mp->list->next == &modules)
                  set $done=1
            end
            set $m=$m->next
            set $count = $count - 1
      end
      if (!$count)
            add-symbol-file ./drivers/char/cse536/cse5361.ko $mp->module_core
      end
end

document lsmod
List the first n loaded kernel modules and their start addresses and load the
symbol table for the nth module
end
```

```
define cse536
#       file vmlinux
        target remote /dev/ttyS1
        lsmod $arg0
        b cse536_open
        c
end

document cse536
load the symbol table at .text address of the nth module and set break at
cse536_open
end
END COPY BEFORE THIS LINE
```

When loaded during ddd or gdb startup it provides two macro functions named cse531 and lsmod. Go ahead and start ddd (or gdb if you don't want a gui interface):

Debug: ddd vmlinux --command csegdbinit

On the target side make sure the mknod and insmod commands have already been run:

Target: sudo mknod /dev/cse5361 c 234 0

Target: sudo insmod drivers/char/cse536/cse5361.ko debug_enable=1

We can now execute the following commands:

Target: sudo bash

to start a root access CLI session on the target machine. We then tell the target kernel to wait for a connection as before:

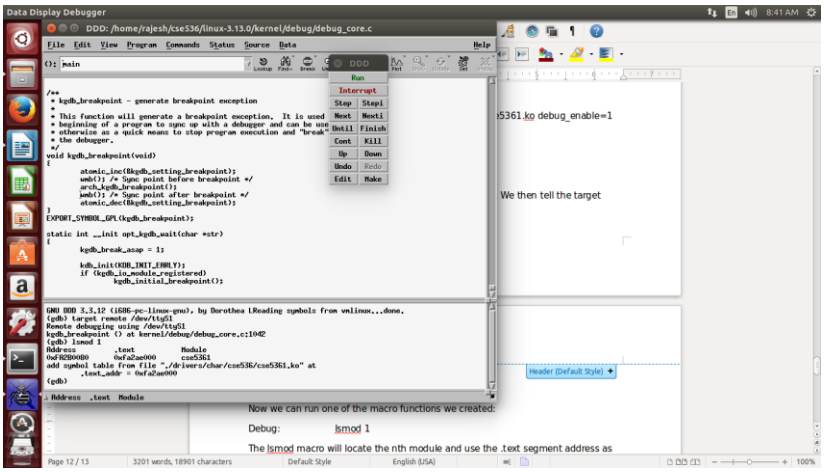
Target: echo g > /proc/sysrq-trigger

Next we make the connection:

Debug: target remote /dev/ttyS1

Now we can run one of the macro functions we created:

Debug: lsmod 1



The lsmod macro will locate the nth module and use the .text segment address as the basis for loading the cse5361.ko symbols. If your module is not the first in the list of modules change the 1 to the nth position in the list of your module. You can now set a breakpoint to occur at the cse536_open function:

Debug: b cse536_open

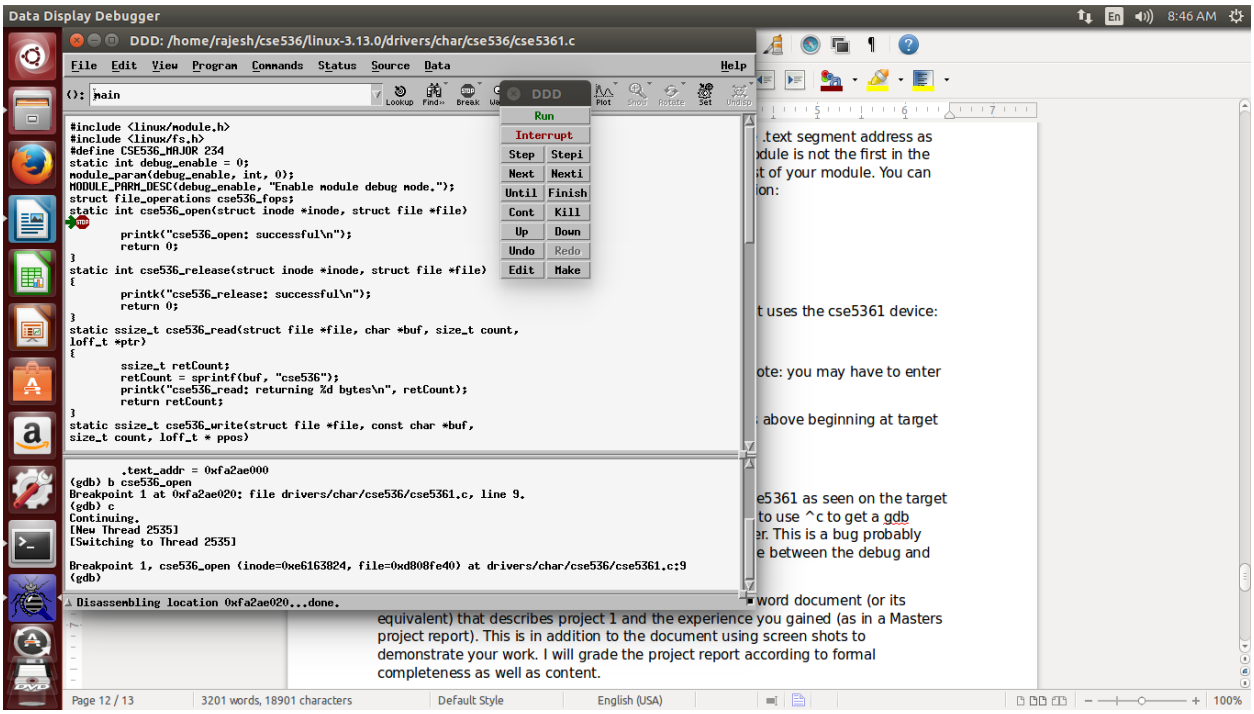
and continue execution of the kernel:

Debug: c

On the now active target machine start the program that uses the cse5361 device:

Target: ./cse536app

You should have a break at the cse536_open function. Note: you may have to enter ^c at this point on the debug VM to get a gdb prompt.



I have provided a macro that replaces some of the steps above beginning at target remote…

Debug: cse536 1

where the 1 is again the position in the module list of cse5361 as seen on the target VM using lsmod. Note again that you will probably need to use ^c to get a gdb prompt after starting cse536app on the remote computer. This is a bug probably caused by a change in the protocol used to communicate between the debug and target VMs in the 3.7 kernel.

Note:

sudo cat /sys/module/cse5361/sections/.text will also provide the address of the .text of the cse5361 module on the target machine if you want to load the symbols manually.

Outcome and Experience:-

In this part of the project we learnt how to build and install a kernel module which is used as a device driver by an application. Here, we used character I/O driver. Most importantly we used the ddd debugger to debug our .ko module. Here, we observed that if we use debugger to debug the kernel then debugger stops the kernel for step by step execution and as a ramification everything running on the kernel also stops. To solve this problem we used two virtual machines- target and debug. This was the most interesting part.