Debugging the Linux Kernel with VirtualBox

By Chris Carlson

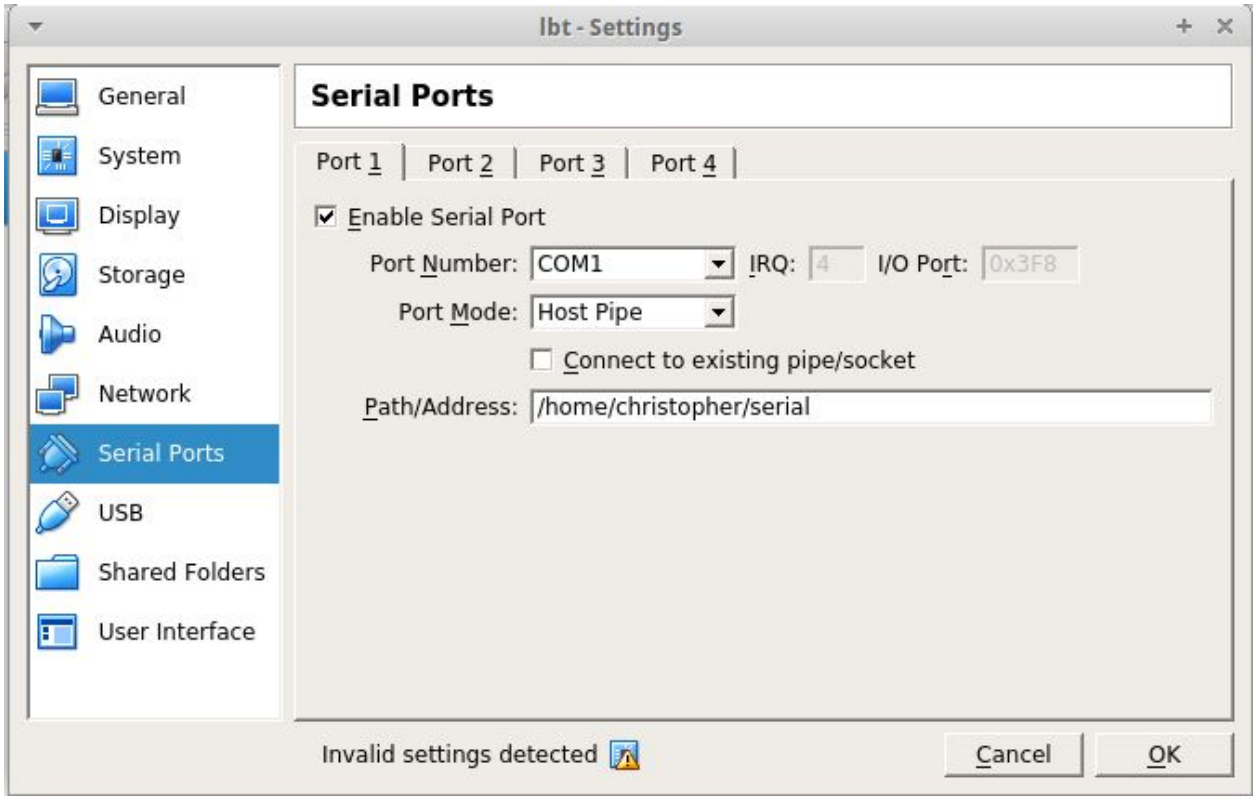
I did this running Virtualbox on XUbuntu 16.04. Inside virtual box I ran LUbuntu 16.04. If I did this more often I would run the same system on both environments. This was my first time doing this. Experts may know better ways to do these things. This did work for me.

Setting up the VirtualBox environment

You'll need a bit of space to compile the kernel, especially if you compile all the modules, I used a 20 gigabyte virtual drive (VDI). Initially I had a 10 gigabyte drive and I ran out of space during the compilation process.

In order to share files between the virtual machine and the host machine, a shared folder is required. This can be setup using the VirtualBox manager under settings for the virtual machine.

Secondly, a serial connection is required in order to facilitate communication between the virtual machine and the host. This can also be setup in the VirtualBox manager under the settings for the virtual machine. I set this up with the following settings:



Compiling the Kernel And Modules on VirtualBox - *This process took a long time*

I downloaded the latest stable kernel, 4.19.6, from kernel.org. This needs to be compiled on the vm, so transfer it to the vm through the shared folder. However, don't compile it in the shared folder because you may run into permissions issues. (It did for me anyway).

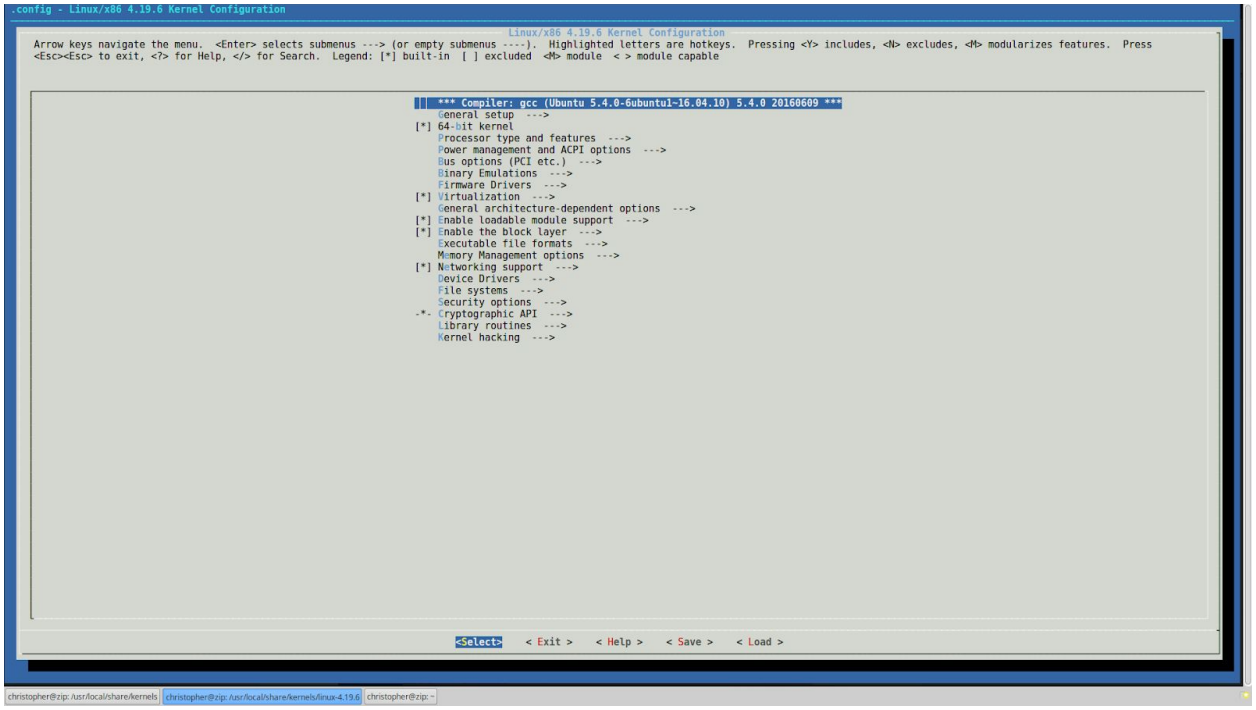
The kernel needs to be configured for debugging.
In order to get the configuration defaults currently in use by the system I used the following command (from the root folder of the linux kernel I wasbuilding):

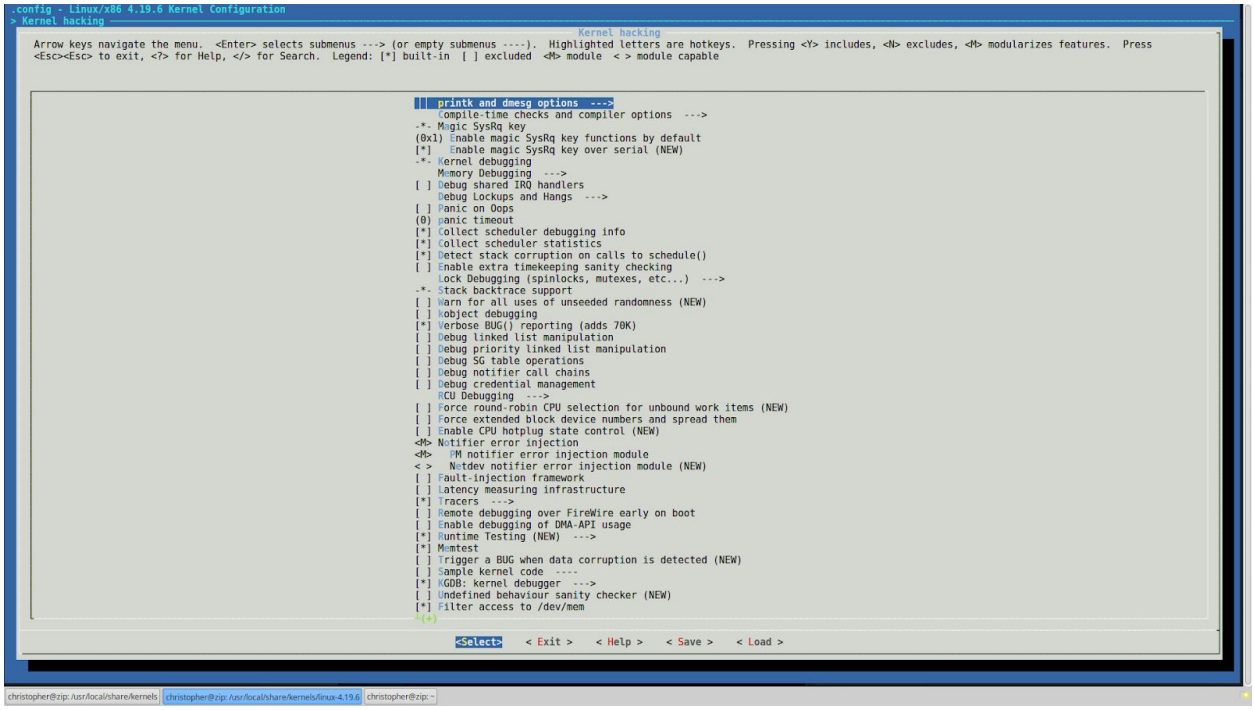
```
$ cp /boot/config-`uname -r` .config
```

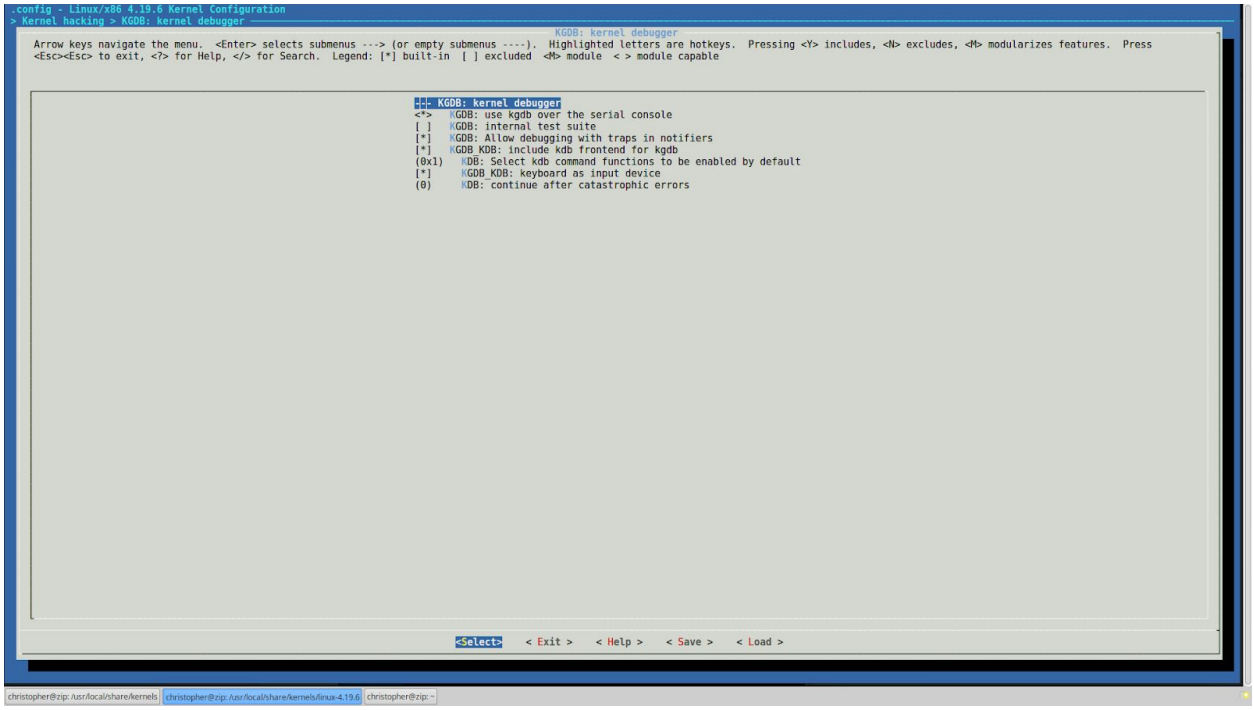
Then to update the configuration options for debugging:

```
$ make menuconfig
```

This command opens a gui utility that allows a few options to be selected. ("Kernel Debugging", and Under the KGDB submenu, "use kgdb over serial console", and a few more, see images below).







Once these options are selected, you have to compile the kernel, and this is the slow part. I started this process at night before I went to bed, so I don't actually know how long it took...

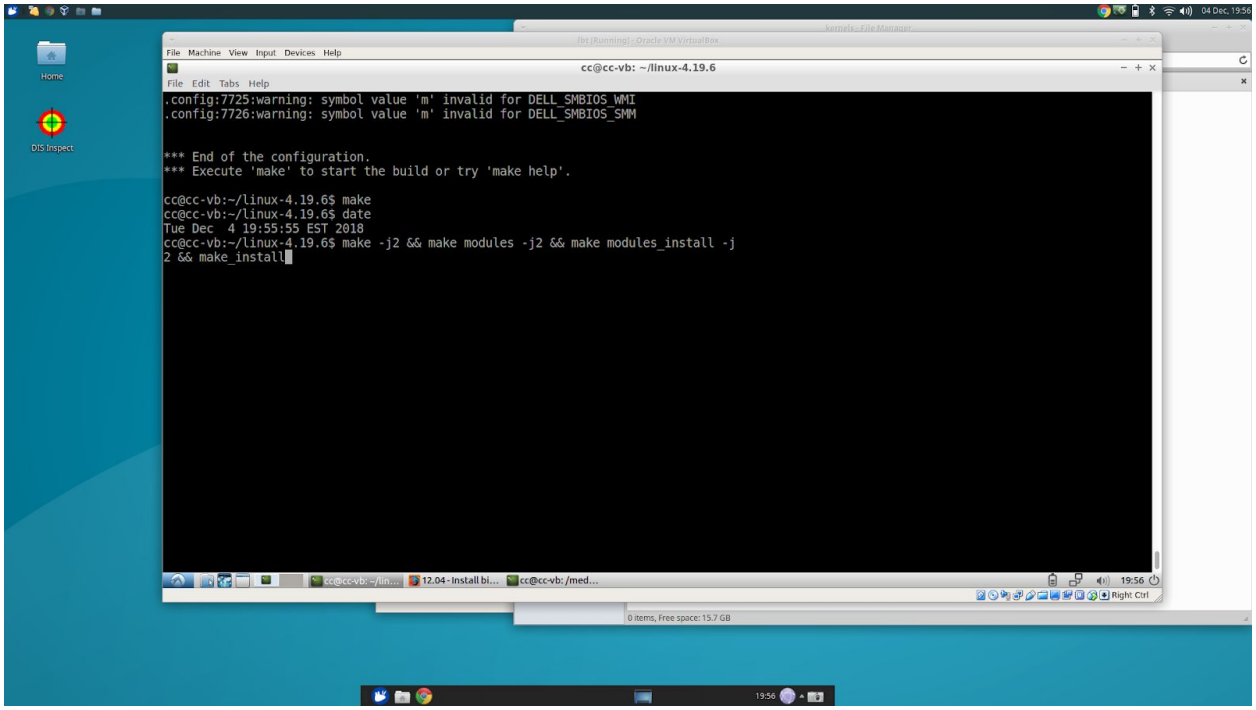
The commands used are:

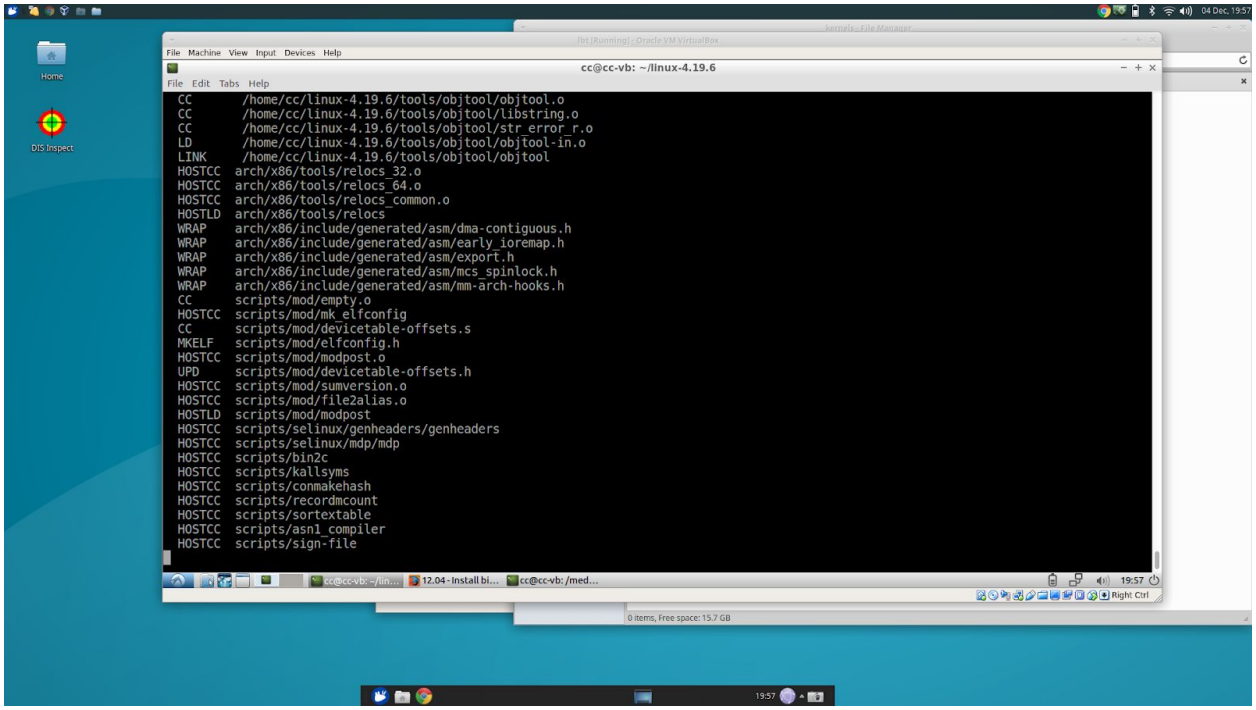
```
$ make -j2 && make modules -j2
```

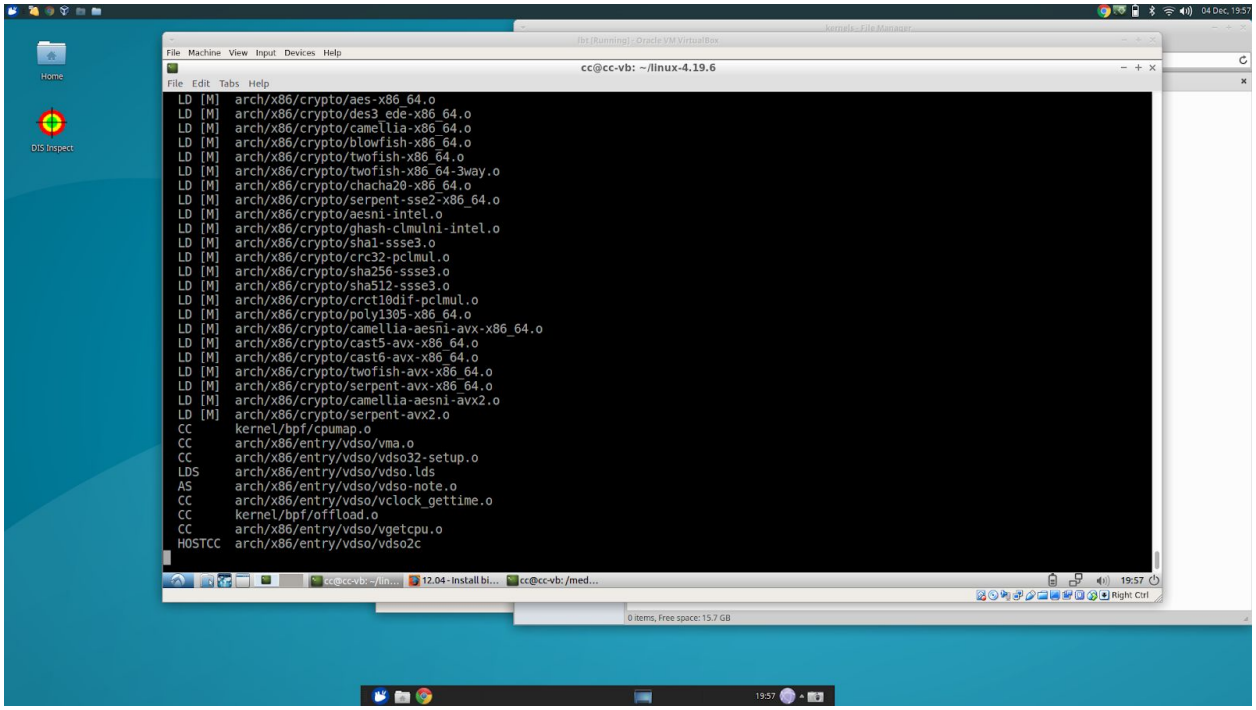
Then, once it has been compiled, install everything with:

```
$ sudo make modules_install && make install
```

(Here are some screenshots from the compile process.)



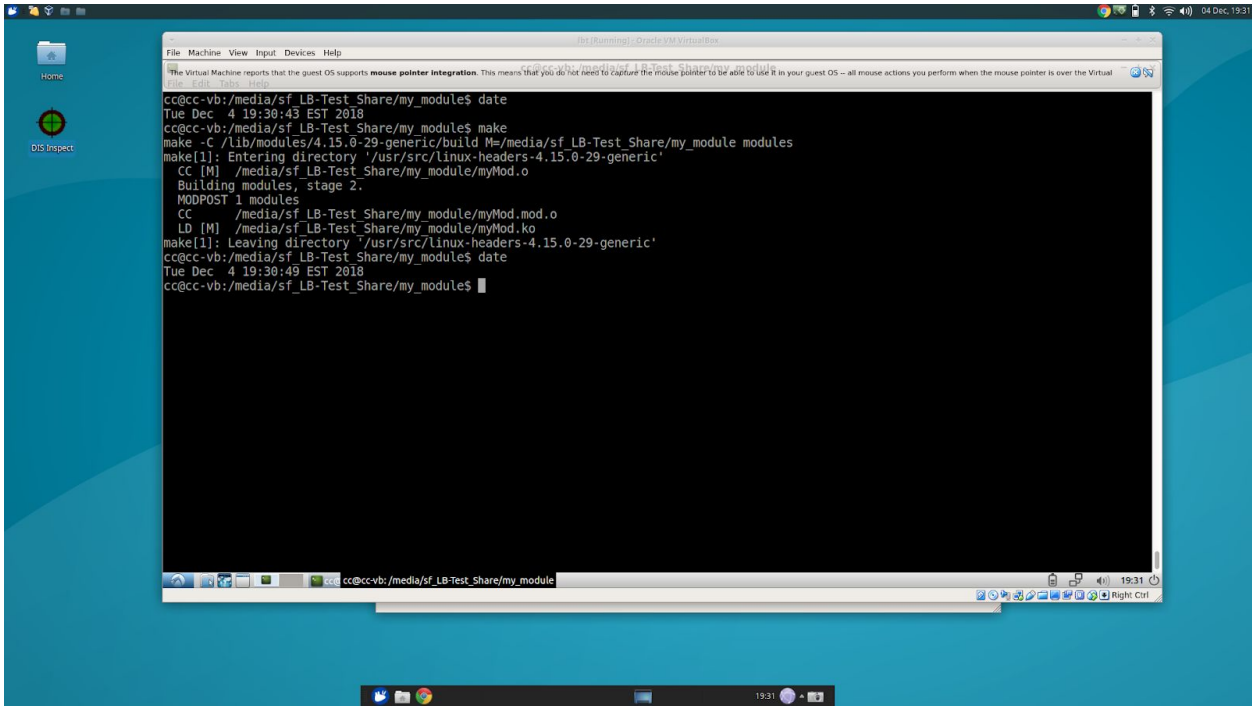




Next I compiled the loadable module I wrote. If your makefile is correct, it should only require a single command:

```
$ make
```

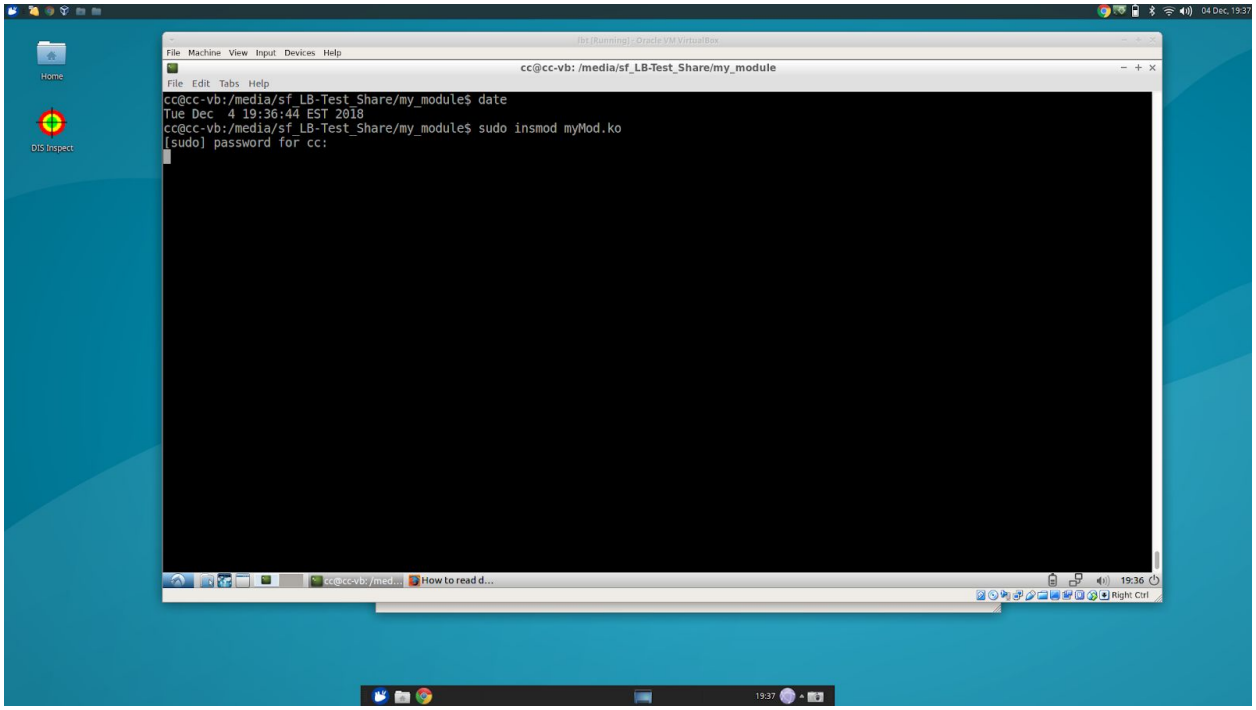
Date Command, Compiling Loadable Module



To insert the loadable module into the system, the command:

```
$ sudo insmod myMod.ko
```

Inserting Loadable Module Crashes the System



Debugging the System

On the host system, I installed gdb and a program called socat. Socat facilitates communication between the host and the vm.

On the test machine the boot options need to be updated so that the system knows to wait on startup for gdb to connect. This can be done from the grub menu by pressing ‘e’ in the grub menu during startup of the vm, or these options can be added to default boot options of the system. For my system this was located at /etc/default/grub. The boot options are:
kgdboc=ttyS0,115200 kgdbwait
*note, S0 is chosen because the serial connection I set up for the VM was COM1. COM2 would use ttyS1, COM3 ttyS2, etc. 115200 is the baud rate, which must match the baud rate given to gdb. Kgdbwait tells the kernel to wait for gdb before it resumes booting.

If you place these options in the default/grub file, they are appended to the option: “GRUB_CMDLINE_LINUX_DEFAULT”

At this point, the vm can be started and it should pause during startup waiting for gdb to connect. The terminal will look something like:

```
[    0.589816] pstore: using deflate compression
[    0.593709] Key type asymmetric registered
[    0.593962] Asymmetric key parser 'x509' registered
[    0.594227] Block layer SCSI generic (bsg) driver version 0.4 loaded (major 2
44)
[    0.594689] io scheduler noop registered
[    0.594916] io scheduler deadline registered
[    0.595183] io scheduler cfq registered (default)
[    0.595734] ACPI: AC Adapter [AC] (on-line)
[    0.596021] input: Power Button as /devices/LNXSYSTM:00/LNXPWRBN:00/input/inp
ut0
[    0.596421] ACPI: Power Button [PWRF]
[    0.596711] input: Sleep Button as /devices/LNXSYSTM:00/LNXSLPBN:00/input/inp
ut1
[    0.597149] ACPI: Sleep Button [SLPF]
[    0.597972] Serial: 8250/16550 driver, 32 ports, IRQ sharing enabled
[    0.598546] battery: ACPI: Battery Slot [BAT0] (battery present)
[    0.620593] 00:02: ttyS0 at I/O 0x3f8 (irq = 4, base_baud = 115200) is a 1655
0A
[    0.623351] KGDB: Registered I/O driver kgdboc
[    0.623862] KGDB: Waiting for connection from remote gdb...

Entering kdb (current=0xffff8a91f6c95b00, pid 1) on processor 0 due to Keyboard
Entry
[0]kdb> _
```

To connect the host to gdb first I used socat to create a serial connection with the machine. The command I used to setup a serial connection point was:

```
$ socat -d -d /home/christopher/serial pty
```

The output from this looks like:

```
christopher@zip:/usr/local/share/kernels$ socat -d -d /home/christopher/serial pty
2018/12/06 13:22:35 socat[11624] N opening connection to AF=1 "/home/christopher/serial"
2018/12/06 13:22:35 socat[11624] N successfully connected from local address AF=1 "\xEE\xEE\xEE\xEE\xEE\xEE"
2018/12/06 13:22:35 socat[11624] N successfully connected via <anon>
2018/12/06 13:22:35 socat[11624] N PTY is /dev/pts/2
2018/12/06 13:22:35 socat[11624] N starting data transfer loop with FDs [5,5] and [6,6]
```


The socat command informs us that a serial connection to the test machine can be made at /dev/pts/2.

Finally, a connection can be made to the test command through GDB. I used an init file with all the settings I intend to use. The contents of the init file are:

```
file ./vmlinux
set serial baud 115200
target remote /dev/pts/2
```

These options tell gdb three things. First, that the file to debug is called vmlinux. This is the image of the linux kernel that is running. Second, that the baud rate is 115200 (which we

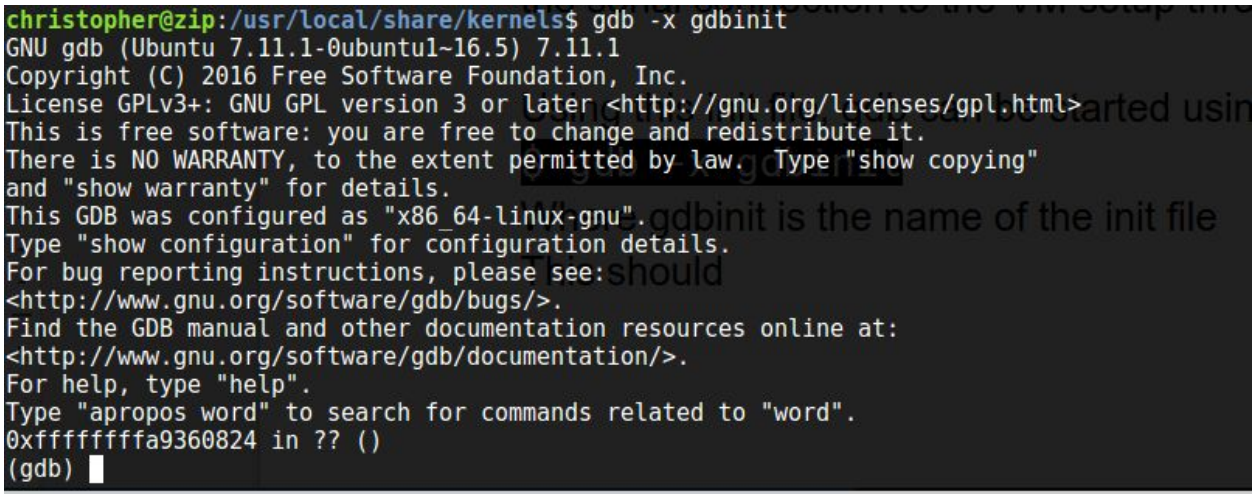
specified already on the test machine). And third, that the remote target is /dev/pts/2, which is the serial connection to the VM setup through socat.

Using this init file, gdb can be started using the command:

```
$ gdb -x gdbinit
```

Where gdbinit is the name of the init file

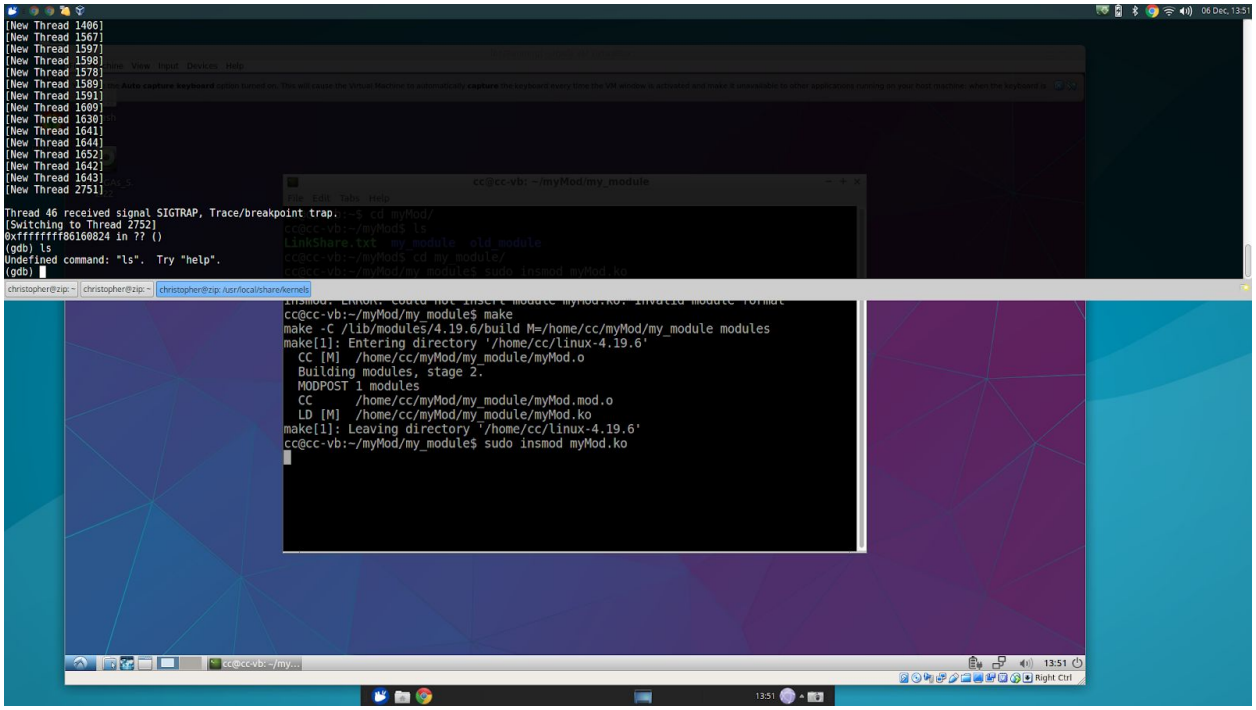
A successful connection looks like:



Now typing the continue command (“c”) into gdb and hitting enter will continue the boot sequence in your vm. From here I was able to debug my module as expected.

Debugging the System

The following screenshots demonstrate the phases of system debugging. This first image demonstrates the point where I have inserted the bad module. I have set a breakpoint in the bad module, and gdb has paused execution of the program reporting, “Thread 46 received signal SIGTRAP, Trace/breakpoint trap.”. Notice that the terminal in the vm, shown below is frozen.



In this second image, I have stepped a few more instructions to the point where the program actually crashes. Here gdb reports, “Thread 46 received signal SIGSEGV, Segmentation fault.” Of course, the terminal shown below is still frozen because now the kernel has crashed. Clearly it was thread 46 that was loading the kernel module.