

# Custom Kernel Guide

by Brian Fraser

Last update: June 11, 2020

## This document guides the user through:

1. Downloading and compiling the Linux kernel's source code.
2. Running a custom kernel inside a text-only virtual machine (via QEMU).
3. Re-configuring and rebuilding the kernel.
4. Compiling an application to run in the QEMU VM and copying the file into it.

## Table of Contents

1. Compiling Linux Kernel form Source.....	2
2. Running a Custom Kernel in QEMU.....	5
3. Modify Kernel and Rebuild.....	8
4. Build & Deploy Linux App to QEMU.....	9
4.1 SSH into QEMU VM.....	10

## Formatting

1. Commands starting with \$ are Linux console commands on the host PC:  
\$ echo Hello world!
2. Commands starting with # are Linux commands on the QEMU target VM:  
# echo Hello QEMU world!  
(The # will be highlighted to draw attention to it being in QEMU!)
3. Almost all commands are case sensitive.

## Revision History

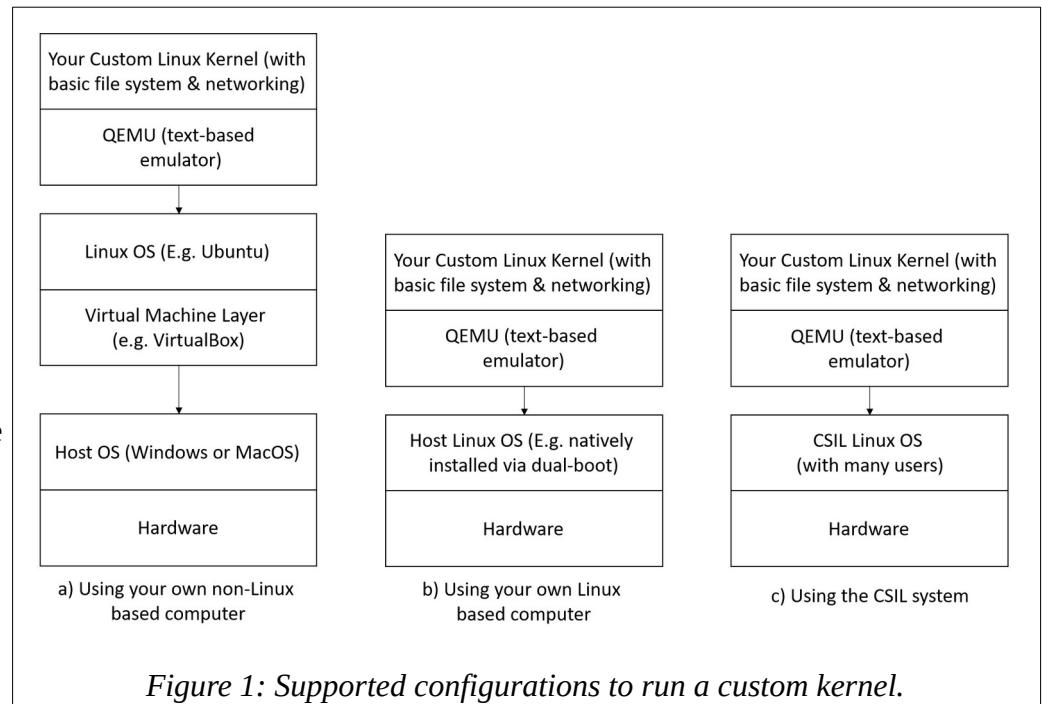
- June 1: Initial version
- June 9:
  - Added support for older version of QEMU with networking;
  - Added introduction to each section
- June 10: Added overview section; minor clarifications.
- June 11: Added extra troubleshooting steps for SCP

# 1. Overview

For this assignment you will be writing a couple Linux kernel system calls (syscalls). Syscalls are how user-space programs (like our standard Hello World) can communicate with the kernel directly.

Since syscalls are part of the kernel, we must:

1. Get the kernel's source code
2. Add our syscall implementation to the kernel
3. Compile and run our custom kernel



Since our custom kernel may be buggy, we want to run it in it's own virtual machine. There are a few different supported scenarios in which you can complete this guide and this assignment (see Figure 1):

- a) If you are running Windows or MacOS on your home computer, then install Linux (such as Ubuntu 20.04) inside a virtual machine (such as VirtualBox). Inside your Ubuntu VM you download the kernel code, edit it, compile it, and write your test programs. Also inside your Ubuntu VM you run the text-based emulator QEMU (which is a virtual machine) to run your custom kernel.
- b) If you are already running Linux as your primary ("native") OS, you work on the kernel inside your Linux install, and then install QEMU to run your custom kernel.

Ubuntu Linux (or other "distros") can be installed to "dual boot" on most computers. In this configuration, Ubuntu installs itself along side your existing OS (such as Windows) so that when your computer boots you select which OS you'd like to boot. Running Linux natively makes it faster; however, you must be in either Linux or your normal OS at one time, and there is a possibility of things going wrong and you spending time to fix your system.

- c) You can use the CSIL computers, which run Linux natively, to remotely complete the assignment. We have special space allocated for us (in `/usr/shared/CMPT/faculty/.....`) because compiling the kernel takes a lot of space. I recommend you connect to CSIL using a remote graphical connection (instead of the command-line only) so you can have multiple windows open at once, edit your code in VS Code, and even browse the web all over the remote connection. See assignments page for directions (at top of page).

## 2. Compiling Linux Kernel from Source

**Section Goal:** Compile our own Linux kernel from source so we can later add our own syscall.

This works best if run on a 64-bit Linux OS. 32-bit is marginally supported with tips on how to make it work. **You can check using command “`uname -m`”: i686 means 32-bit, x86\_64 means 64-bit.**

1. This guide has been tested under Ubuntu 20.04 and CSIL. It should work well under other versions of Linux. It has not been tested under WSL or native MacOS and we are unable to offer support for these OS's (you should install a virtual machine with Linux on it).

2. If on your own computer, install necessary tools; (don't execute `sudo` commands on CSIL!):

```
$ sudo apt-get update
$ sudo apt-get install gcc flex bison libelf-dev libssl-dev
```

3. Create a directory for your Linux kernel source code.

- **If working on your own computer:** create a folder for your Linux kernel source code:

```
$ mkdir ~/cmpt300
$ cd ~/cmpt300
```

It will require ~1.7GiB to download the kernel code and compile it.

You can check how much disk space the current drive has free:

```
$ df -h .
```

- **If working on CSIL Linux machines:** you can access a large storage area which is mapped for your personal (protected) use. Use this space as the base folder for all work done in this guide. So instead of `~/cmpt300`, use:

```
/usr/shared/CMPT/faculty/bfraser/cmpt300/student-working-directories/%sfu-user%
```

```
$ cd /usr/shared/CMPT/faculty/bfraser/cmpt300/
$ cd student-working-directories/%sfu-user%
```

Note that on CSIL machines the '`df -h .`' may be inaccurate due to quotas, mounting, etc.

4. Get the Linux kernel source code and extract it:

```
$ wget https://mirrors.edge.kernel.org/pub/linux/kernel/v5.x/linux-5.7.tar.xz
$ tar -xvf linux-5.7.tar.xz
```

The tar.xz file is ~100MB.

5. Change to the newly created `linux-5.7` directory:

```
$ cd linux-5.7
```

6. View the files:

```
$ ls
```

- Your output should look something like the following:

```
~/cmpt300/linux-5.7$ ls
arch      CREDITS    fs          Kbuild     LICENSES   net         security   virt
block     crypto     include     Kconfig    MAINTAINERS  README     sound
certs     Documentation  init        kernel     Makefile    samples    tools
COPYING   drivers    ipc         lib         mm          scripts    usr
```

7. Setup the default `config` file for building the kernel. This file has all the build options, the defaults will be sufficient for the moment:

```
$ make defconfig
```

- Read any errors you see. It may tell you certain packages are missing. If so, use `apt-get` to install them.

8. Determine how many processors your system has:

```
$ nproc
```

9. Build the Linux kernel:

```
$ make -j4
```

- Where 4 is the number of cores your system has (discover above). Using more cores than you actually have can cause the build to be slow or to fail.
- Running this command may take a while: it took 15 minutes on my PC's VM with 4 cores. It took 2H on the CSIL machine with 8 cores (networked file system).

10. View the kernel file that you just built.

- On a 64-bit OS:

```
$ ls -l arch/x86_64/boot/bzImage
```

- Expected output (the `->` indicates it is a symbolic link to the `x86/` folder):

```
lrwxrwxrwx 1 brian brian 22 Jun  1 10:50 arch/x86_64/boot/bzImage
-> ../../x86/boot/bzImage
```

- On a 32-bit OS:

```
$ ls -l arch/x86/boot/bzImage
```

- If it's not there see the troubleshooting section below.

11. Troubleshooting

- Kernel build failed?

- If there is a problem building, look at the build output to see if there are errors. Try searching the web for hints on resolving your error.

- Ensure that you are not out of space on the current drive (read the "Avail" column):

```
$ df -H .
Filesystem      Size  Used Avail Use% Mounted on
/dev/sda1       57G   47G   7.7G  86% /
```

- Run a `"make clean"`, followed by a `"make -j1"`. Specifying more cores than your system (VM) has can cause build problems. List the number of cores on your system with:

```
$ nproc
```

- If the `x86_64/` directory did not show up when building on a 64-bit OS, it could indicate a build problem. However, if the `x86` folder is there (32-bit), you may be able to use it instead.

- If running in a VM and sharing the folder with the guest OS, make sure that the file system you are working inside of supports case-sensitive file names. For example, building in a folder shared from Mac OS will cause files whose name differ only in letter case to overwrite each other and cause the build to fail.

- If running in CSIL, ensure you are not in your home folder! Must be under `/usr/...`
- VM Too Slow? If running this in a VM, try the following:
  - Give the VM more memory (RAM) to work with.
  - Give the VM more CPU cores to work with, and “`make -j4`” (4 cores, ...)
  - Enable 3D graphics acceleration to the VM.

All of these changes must be done when the VM is powered down.

- It's possible to get the Linux kernel source code via Git instead of the ZIP file.  
Checkout the Linux code from Linus Torvald's “`linux-stable`” repository:  

```
$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git
```

  - This will download approximately 1.4GiB of data into a new `linux-stable/` folder. This command will take a fair amount of time because you are downloading all of the Linux source code!

Find latest stable tag (option is a lower case ‘-L’):

```
$ cd linux-stable
$ git tag -l | less
```

- Find highest non -rcX version. The -rcX versions are release candidates which were subsequently improved on to build the non-rcX versions. Note it's a lexical sort: 3.11 comes before 3.9. As of this writing (June 2020) the latest non-rcX version is v5.7

Check out the code for this latest stable release so you have the desired version to work on.

```
$ git checkout -b stable vX.Y.Z
```

- Where `vX.Y.Z` is the version you identified in the previous step. For example:  

```
$ git checkout -b stable v5.7
```

### 3. Running a Custom Kernel in QEMU

**Section Goal:** Run our custom kernel so we can see if it works! We'll run it inside a lightweight virtual machine (QEMU). We run it in its own virtual machine so that if things go wrong we can just terminate QEMU, recompile, and re-run our kernel.

Note that this will work fine even if you are developing in a Linux virtual machine. For example, I am developing on a Windows 10 PC, which is running Ubuntu for my CMPT 300 coding (inside a VirtualBox VM), which is in turn running my custom kernel (inside a QEMU virtual machine).

1. If you are working on **your own machine**, then install QEMU:

```
$ sudo apt-get update
$ sudo apt-get install qemu-system-x86
```

If you are working in the **CSIL labs** then QEMU will already be installed.

NOTE: Never run “sudo” in CSIL. You may, however, run `sudo` commands *inside* your VM (i.e., inside QEMU), but not on the host machine.

2. Download a small pre-built root file system. The root file system contains all the installed programs, such as the programs the kernel will run after it boots.<sup>1</sup>

```
$ cd ~/cmpt300/
$ wget https://www2.cs.sfu.ca/~bfraser/cmpt300/debian_squeeze_amd64_standard.qcow2
```

- If using CSIL, use the path under `/usr/shared/...`:  

```
$ cd /usr/shared/CMPT/faculty/bfraser/cmpt300/
$ cd student-working-directories/%sfu-user%/
```
- If using a 32-bit OS, you'll instead need to download the file `debian_squeeze_i386_standard.qcow2` from the same director.

3. From a terminal, change to the `linux-5.7/` folder:

```
$ cd linux-5.7/
```

- If on your own machine, you'll now be in `~/cmpt300/linux-5.7/`

4. Run QEMU using one of the following commands (command is all on one line!). See course website for script you can copy-and-paste from (from a PDF works poorly).

- **[Recommended]** Launch using the current terminal window (note “sda1” ends in a one):

```
$ qemu-system-x86_64 -m 64M -hda ../debian_squeeze_amd64_standard.qcow2 \
-append "root=/dev/sda1 console=tty0 console=ttyS0,115200n8" \
-kernel arch/x86_64/boot/bzImage -nographic
```

- **[Alternate]** Launch another window using SDL:

```
$ qemu-system-x86_64 -m 64M -hda ../debian_squeeze_amd64_standard.qcow2 \
-append "root=/dev/sda1 console=ttyS0,115200n8 console=tty0" \
-kernel arch/x86_64/boot/bzImage &
```

- Press CTRL and ALT together to leave QEMU window.
- It may be slow! During this time, you should see many log messages to the screen.

It took **180 seconds (3m)** in a **Linux VM** to complete booting my kernel in QEMU.

<sup>1</sup> RFS original source: <https://people.debian.org/~aurel32/qemu/amd64/>

It took **30 seconds on a native Linux** (CSIL) to complete booting my kernel in QEMU.

- Improve speed of QEMU inside a Linux VM:
    - If you have an AMD processor (only) power off your VM, edit the settings of the VM, under System → Processor; check “Enable Nested VT-x/AMD-v”. This may improve your performance.
    - If you have an Intel processor, this suggestion here may give you better performance when running QEMU: <https://stackoverflow.com/a/57229749/3475174>
  - If using a 32-bit system, you'll need to change the command in the following two ways:
    - 1) use `debian_squeeze_i386_standard.qcow2`
    - 2) use the kernel image path: `arch/x86/boot/bzImage`
5. When the QEMU virtual machine has finished booting, you should see the following login prompt:  
Debian GNU/Linux 6.0 debian-amd64 ttyS0
- debian-amd64 login:
6. Log into the machine. It has the following two users configured by default. I recommend logging in as root (not usually a good practice, but not a big problem in this case).
- Name: **root**  
Password: **root**
  - You could also log in with “user” and “user” as the username and password.
7. Congratulations! You are now running a Linux kernel that you compiled! At this point, you could now do anything you wanted with the system.
- Changes you make to the file system inside QEMU **will** be saved for the next time you boot the system.
8. When done using the QEMU VM, you should tell Linux running in it to power down:  
`# poweroff`
- If you simply close the QEMU VM it is like unplugging your computer: it may lead to a corrupted file system inside the VM for which you'll need to redownload the root file system `.qcow2` file.

If QEMU is unable to shut down gracefully, you may need to kill it from your host OS:

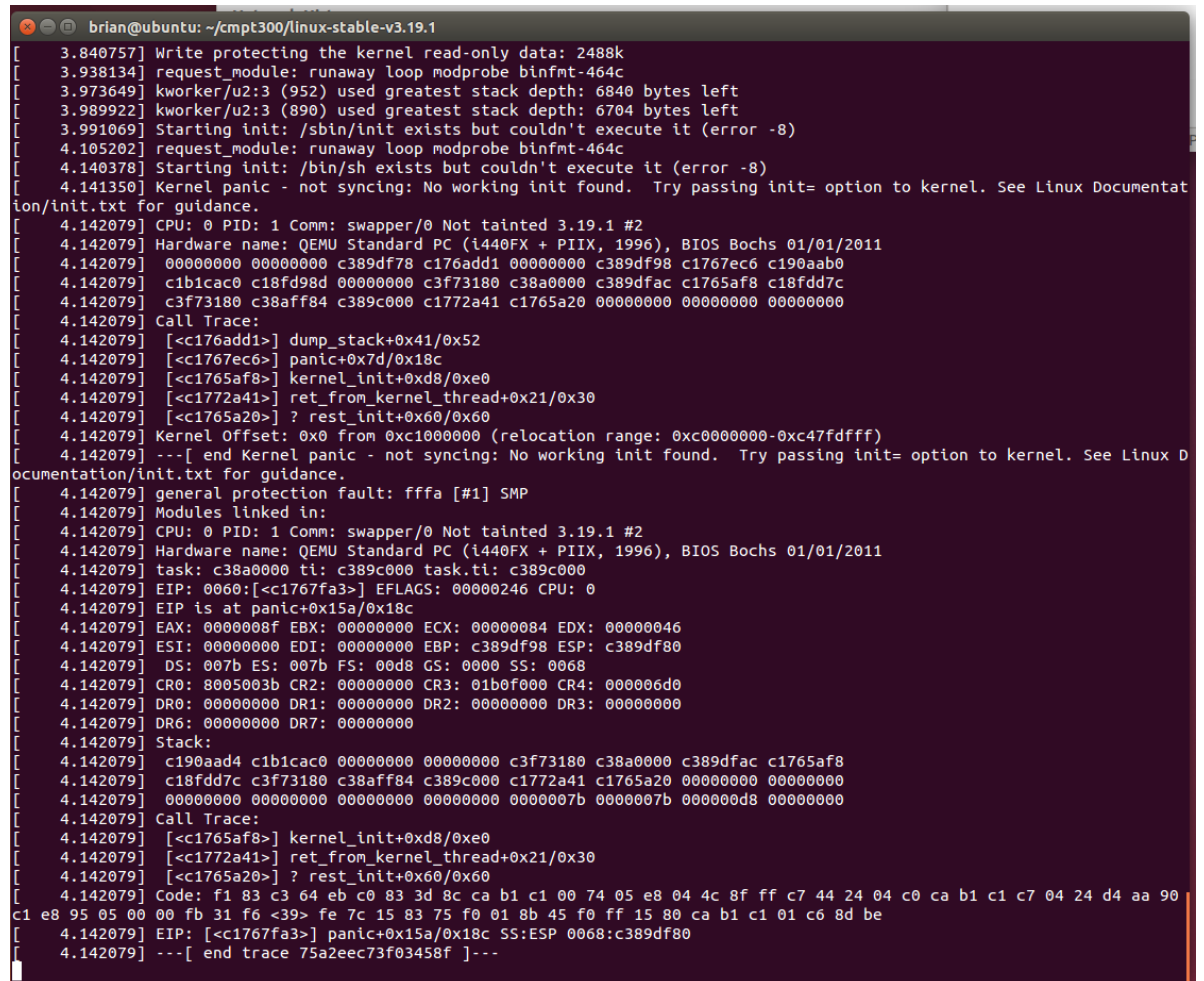
- You can kill all QEMU systems with:  
`$ killall qemu-system-x86_64`
- Or, you can find it in the list of processes you are running, and signal it directly:  
brian@ubuntu:~/cmpt300/linux-5.7\$ **ps -a**  

PID	TTY	TIME	CMD
9535	pts/18	00:00:06	gedit
11357	pts/18	00:00:00	alsamixer
61981	pts/26	00:01:51	qemu-system-x86
62029	pts/18	00:00:00	ps

  
brian@ubuntu:~/cmpt300/linux-5.7\$ **kill 61981**
- You should only kill the QEMU process (or close its window) when you are unable to execute the `poweroff` command. Failing to do so may corrupt the file system.

## 9. Troubleshooting:

- If you get a kernel panic at boot, it could be a problem access the root file system. A couple things to check:
- If you get a kernel panic, you may need to scroll back a bit to the start of the panic to see what went wrong. You want to read the couple lines above the “Kernel panic” lines. You may need to use the `-nographic` option so that the output is in your current terminal window and hence you can scroll back. Figure 2 shows a sample kernel panic (this one for trying to run the 64-bit root file system with a 32 bit kernel).



```
brian@ubuntu: ~/cmpt300/linux-stable-v3.19.1
[ 3.840757] Write protecting the kernel read-only data: 2488k
[ 3.938134] request_module: runaway loop modprobe binfmt-464c
[ 3.973649] kworker/u2:3 (952) used greatest stack depth: 6840 bytes left
[ 3.989922] kworker/u2:3 (890) used greatest stack depth: 6704 bytes left
[ 3.991069] Starting init: /sbin/init exists but couldn't execute it (error -8)
[ 4.105202] request_module: runaway loop modprobe binfmt-464c
[ 4.140378] Starting init: /bin/sh exists but couldn't execute it (error -8)
[ 4.141350] Kernel panic - not syncing: No working init found. Try passing init= option to kernel. See Linux Documentat
ion/init.txt for guidance.
[ 4.142079] CPU: 0 PID: 1 Comm: swapper/0 Not tainted 3.19.1 #2
[ 4.142079] Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS Bochs 01/01/2011
[ 4.142079] 00000000 00000000 c389df78 c176add1 00000000 c389df98 c1767ec6 c190aab0
[ 4.142079] c1b1cac0 c18fd98d 00000000 c3f73180 c38a0000 c389dfac c1765af8 c18fdd7c
[ 4.142079] c3f73180 c38aff84 c389c000 c1772a41 c1765a20 00000000 00000000 00000000
[ 4.142079] Call Trace:
[ 4.142079] [] dump_stack+0x41/0x52
[ 4.142079] [] panic+0x7d/0x18c
[ 4.142079] [] kernel_init+0xd8/0xe0
[ 4.142079] [] ret_from_kernel_thread+0x21/0x30
[ 4.142079] [] ? rest_init+0x60/0x60
[ 4.142079] Kernel Offset: 0x0 from 0xc1000000 (relocation range: 0xc0000000-0xc47fdfff)
[ 4.142079] ---[ end Kernel panic - not syncing: No working init found. Try passing init= option to kernel. See Linux D
ocumentation/init.txt for guidance.
[ 4.142079] general protection fault: fffa [#1] SMP
[ 4.142079] Modules linked in:
[ 4.142079] CPU: 0 PID: 1 Comm: swapper/0 Not tainted 3.19.1 #2
[ 4.142079] Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS Bochs 01/01/2011
[ 4.142079] task: c38a0000 ti: c389c000 task.ti: c389c000
[ 4.142079] EIP: 0060:[<c1767fa3>] EFLAGS: 00000246 CPU: 0
[ 4.142079] EIP is at panic+0x15a/0x18c
[ 4.142079] EAX: 0000008f EBX: 00000000 ECX: 00000084 EDX: 00000046
[ 4.142079] ESI: 00000000 EDI: 00000000 EBP: c389df98 ESP: c389df80
[ 4.142079] DS: 007b ES: 007b FS: 00d8 GS: 0000 SS: 0068
[ 4.142079] CR0: 8005003b CR2: 00000000 CR3: 01b0f000 CR4: 000006d0
[ 4.142079] DR0: 00000000 DR1: 00000000 DR2: 00000000 DR3: 00000000
[ 4.142079] DR6: 00000000 DR7: 00000000
[ 4.142079] Stack:
[ 4.142079] c190aad4 c1b1cac0 00000000 00000000 c3f73180 c38a0000 c389dfac c1765af8
[ 4.142079] c18fdd7c c3f73180 c38aff84 c389c000 c1772a41 c1765a20 00000000 00000000
[ 4.142079] 00000000 00000000 00000000 00000000 0000007b 0000007b 000000d8 00000000
[ 4.142079] Call Trace:
[ 4.142079] [] kernel_init+0xd8/0xe0
[ 4.142079] [] ret_from_kernel_thread+0x21/0x30
[ 4.142079] [] ? rest_init+0x60/0x60
[ 4.142079] Code: f1 83 c3 64 eb c0 83 3d 8c ca b1 c1 00 74 05 e8 04 4c 8f ff c7 44 24 04 c0 ca b1 c1 c7 04 24 d4 aa 90
c1 e8 95 05 00 00 fb 31 f6 <39> fe 7c 15 83 75 f0 01 8b 45 f0 ff 15 80 ca b1 c1 01 c6 8d be
[ 4.142079] EIP: [<c1767fa3>] panic+0x15a/0x18c SS:ESP 0068:c389df80
[ 4.142079] ---[ end trace 75a2ec73f03458f ]---
```

Figure 2: Sample terminal window showing a kernel panic on boot.

- Make sure you have the correct root file system downloaded for your OS version (64-bit vs 32-bit) and that it is in the expected location.
- The following message (seen when booting QEMU) likely means you have the wrong root file system version (32bit vs 64bit)for the kernel you computed.  
Starting init: /bin/sh exists but couldn't execute it (error -8)  
Kernel panic - not syncing: No working init found. Try passing  
init= option to kernel. See Linux Documentation/init.txt for  
guidance.



- The following message likely means that you have a corrupt/invalid root file system image. Redownload the root file system .qcow2 image file.  
VFS: Cannot open root device "sda1" or unknown-block(8,1): error -6
- Try running the QEMU launch script found on the course website to ensure there was no typing problem in entering the command.
- When launching QEMU, if you see an error:  
qemu-system-x86\_64: -hda ../debian\_squeeze\_amd64\_standard.qcow2: Failed to get "write" lock  
Is another process using the image [../debian\_squeeze\_amd64\_standard.qcow2]?

Then you likely have a copy of QEMU running already. Close any running instances (use the 'poweroff' command inside QEMU, or use the Linux kill command outside of QEMU).

- If you get messages about file system corruption and `fsck`, it likely means the root file system image has corrupted and you need to re-download a new one. The corruption is likely due to the VM being killed or closed instead of using the `poweroff` command.
- While trying to download the .qcow2 file, if you get a 403 (forbidden) or 404 (file not found) error, then double check the path.
- When booting your custom kernel in QEMU, if you get any errors about "udev" failing, this seems to not be a problem as it happens for me too.

## 4. Modify Kernel and Rebuild

**Section Goal:** Modify the kernel a little to show that it is indeed our kernel we are running! This will show during marking that you have rebuilt your own kernel, and help you see a change that you have made. It's the "hello-world" of building your own kernel.

1. If running on your own computer, install the necessary library to configure the kernel. (This is already done on CSIL machines).  
`$ sudo apt-get install libncurses5-dev`
2. Form within the `linux-5.7/` folder, launch the Linux kernel build configuration menu:  
`$ make menuconfig`
3. Change the kernel's "Local version". This string is appended to the kernel's version number:
  - Under "General setup --->"
  - Under "Local version - append to kernel release"
  - Type in "-sfuid", where `sfuid` is **your** SFU ID, such as for me I type "-bfraser"
  - Press Enter to accept change, then use right-arrow key to select Save and Exit.
4. Rebuild the kernel (set `-j4` to be the number returned by `nproc` command):  
`$ make -j4`
  - This should build much faster this time because it is only rebuilding the parts of the kernel which change, as opposed to rebuilding the entire kernel.
  - If you want to trigger a full kernel rebuild, first run the following before running `make`:  
`$ make clean`
5. Use QEMU, as before, to boot your custom kernel. Since you have now rebuilt the kernel, it will now load your new kernel.
6. Once logged into the QEMU virtual machine, check that the kernel version has changed:  
`# uname -a`
  - Alternatively, you can check the kernel version inside your QEMU VM using:  
`# cat /proc/version`

## 5. Build & Deploy Linux App to QEMU

**Section Goal:** Have a hello-world user-space application running on our custom kernel. In order to do anything interesting with our custom kernel (running inside QEMU), we'll need to have a user-space application to interact with it.

We'll now write and compile hello-world in our Linux development system (which is Ubuntu for me), and then copy it to my custom kernel running in QEMU. Since QEMU is a virtual machine, it has its own file system, so we need to copy from one VM to another. To do this, we'll use SCP (part of SSH) to copy the file using networking.

We'll connect port 10022 on your Linux development system (Ubuntu for me) to port 22 your QEMU virtual machine. The QEMU VM already has an SCP server listening to its port 22. So we'll SCP a file to your Linux development system's port 10022, and inside the QEMU VM it will look like a request coming in via port 22.

1. Our QEMU VM has only limited libraries installed. Rather than trying to install extra libraries inside our QEMU VM, we'll just link all necessary libraries into our programs that we want to run inside QEMU's VM. This is called statically-linking an executable (binary).
  - Create a simple `helloWorld.c` program which outputs some happy message.
  - Use the `-static` option for GCC to have it link a static binary.  

```
$ gcc helloWorld.c -std=c99 -static -o helloWorld
```
  - If you are using a Makefile, you may want to add `-static` to the `CFLAGS`.
2. **Redirect port 10022 on the host OS to the QEMU VM's port 22 (all on one line).**
  - Find out which version of QEMU your system has:  

```
$ qemu-system-x86_64 --version
```
  - **QEMU emulator version 4.2.0**  
(This is the version with Ubuntu 20.04, for example)  

```
$ qemu-system-x86_64 -m 64M -hda ../debian_squeeze_amd64_standard.qcow2 \
-append "root=/dev/sda1 console=tty0 console=ttyS0,115200n8" \
-kernel arch/x86_64/boot/bzImage -nographic \
-nic user,hostfwd=tcp::10022-:22
```
  - **QEMU emulator version 2.11.1**  
(This is the version with Ubuntu 18.04, for example)  

```
$ qemu-system-x86_64 -m 64M -hda ../debian_squeeze_amd64_standard.qcow2 \
-append "root=/dev/sda1 console=tty0 console=ttyS0,115200n8" \
-kernel arch/x86_64/boot/bzImage -nographic \
-net nic,vlan=1 -net user,vlan=1 -redir tcp:10022::22
```
  - The `-nic` (or `-net`) option sets up a network connection between the host and guest OS's.
  - Port 10022 is the host OS's port we'll redirect to the guest OS inside QEMU.
  - Port 22 is the default SSH port (also used for SCP): SSH server inside QEMU will listen to this port.
3. Once guest OS in QEMU has booted, bring up networking:  

```
# ifup eth0
```

4. Copy your file (helloWorld, for example) to the QEMU virtual machine via SCP using the following command executed on in the host OS:

```
$ scp -P 10022 helloWorld root@localhost:~
```

- It will ask you for the root password on the target; this will be root
- This will copy the file helloWorld from your current directory on the host OS into the /root/ folder of the guest QEMU OS.
- Your QEMU VM must have finished booting in order for SCP to work.
- You can copy multiple files (for example helloWorld, myApp, fooFile2) with:  

```
$ scp -P 10022 helloWorld myApp fooFile2 root@localhost:~
```

5. Run your application in the QEMU VM:

```
# cd /root
# ./helloWorld
```

- These commands are run inside the QEMU OS, not the host.

6. Troubleshooting

- If you get the message “could not set up host forwarding rule 'tcp:10022::22'” when launching QEMU, try using a different host port (such as 8383) instead of 10022.
- When running SSH/SCP, if you get the following message, it means QEMU is either not running, or has not correctly been started with the -nic argument:  
ssh: connect to host localhost port 10022: Connection refused  
lost connection
  - Ensure you are running SCP from your Linux system (\$ prompt) vs inside QEMU (# prompt)
  - Double check the command you are using to launch QEMU. Ensure the command is executing as one command. Try copy-and-pasting the command from the “raw commands” link on the course website.
  - Double check networking is working from within QEMU:

```
# ifup eth0           # enable networking inside QEMU
# ping 8.8.8.8         # test connection to internet
# ping google.ca      # other test of internet connection
# ifconfig            # view networking information
```
  - Double check your SCP command to ensure you have the correct port.
- If command seems to hang (does nothing) when you call scp, or you get the error  
ssh\_exchange\_identification: read: Connection reset by peer  
lost connection
  - Ensure the OS running in QEMU has finished booting to the point where you can log in.
  - Ensure you ran QEMU with the networking enabled (-nic option)
  - Ensure you have enabled the network device inside QEMU:

```
# ifup eth0
```

    - List enabled networking devices:

```
# ifconfig
```

- List all networking devices on OS inside QEMU (some may not yet be enabled)  
# ifconfig -a
- Check if you can connect to the internet from inside the guest OS:  
# ping 8.8.8.8  
# ping google.ca
- You may also need to verify SSH is running in the guest OS (beyond the scope of this guide).
- When trying to run your application in QEMU, if you get the following message it means your application is likely not statically linked. Add the `-static` GCC option:  
./myApp: /lib/libc.so.6: version 'GLIBC\_2.17' not found (required by ./myApp)

## 5.1 SSH into QEMU VM

**Section Goal:** Get an extra command-line connection into the QEMU VM so we can run multiple programs, or watch for debug information from the kernel. We'll use SSH to connect to the host OS's 10022 port, which is already mapped to port 22 inside the QEMU VM. An SSH server is listening to this port.

1. Launch the QEMU virtual machine using the configuration described in the previous section.
2. SSH from host into guest VM  
\$ ssh root@localhost -p10022