

Custom Kernel Guide

by Brian Fraser, modified by Mohamed Hefeeda, Julian Rushi

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. Reconfiguring 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!`
3. Almost all commands are case sensitive.

1. Compiling Linux Kernel from Source

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. The suggested build location for your Linux kernel is under your home directory, create a `oscourse` folder:

```
$ cd ~  
$ mkdir oscourse  
$ cd oscourse
```

2. If you are low on disk space (< 2GB free), see the troubleshooting steps at the end of this section for how to acquire the code without using GIT. (You still need 1GB free though!)

3. 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 1.2Gb 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!

4. Change to the newly created `linux-stable` directory:

```
$ cd linux-stable
```

5. Find latest stable tag:

```
$ git tag -l | less
```

- Find biggest non-`-rcX` version. The `-rcX` versions are release candidates which were subsequently improved on to build the non-`-rcX` versions.
- Note that it's a lexical sort: 3.11 comes before 3.9!
- As of this writing (November 2017) the latest non-`-rcX` version is v4.13.9

6. 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 v4.13.9
```

7. View the files:

```
$ ls
```

- Your output should look something like the following:

```
brian@ubuntu:~/oscourse/linux-stable$ ls  
arch      CREDITS          firmware   ipc       lib       net       security  vi  
block     crypto            fs         Kbuild   MAINTAINERS README   sound  
certs     Documentation    include    Kconfig  Makefile  samples  tools  
COPYING   drivers           init      kernel   mm       scripts  usr
```

8. 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
```

9. Build the Linux kernel:

```
$ make -j4
```

- Where 4 is the number of cores your system has (discover with command `nproc`). Using the

wrong number of course (i.e., more cores than you actually have) is known to cause the build to fail.

- Running this command may take a while. For example, it took 6 minutes on an Intel i5 quad-core in a virtual machine, or 17 minutes on a single core VM on the same computer.
- Building the kernel takes an additional ~300 Mb of hard drive space.

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 Mar 29 23:45 arch/x86_64/boot/bzImage -> ../../x86/boot/bzImage
```

- On a 32-bit OS:

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

- Expected output:

```
-rw-rw-r-- 1 brian brian 5948480 Mar 30 21:31 arch/x86/boot/bzImage
```

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

11. Troubleshooting

- Out of space?

Checking out and building the kernel takes a lot of space (git clone takes 1.8Gb, and building takes an additional 0.3Gb).

- You can avoid downloading the code via GIT (steps 1-2 above) and save much of the space required by instead downloading the archive of the code directly from the course website. This file is ~160Mb, and expands to be ~650Mb (consumes ~1Gb after building the kernel).

- Extract the contents of the file. This will create a sub-folder:

```
$ unzip linux-stable-v3.19.1.zip
```

- In this document's directions when it refers to the linux-stable/ folder, this is your linux-stable-v3.19.1/ folder if you used this archive.

- If space is tight, you could delete/move the .zip file now. However, it could be useful if you later need to revert to the original code.

- Proceed with step 2 (above).

- 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 often 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, 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.
- 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.
 - Enable 3D graphics acceleration to the VM.
 - (All of these changes must be done when the VM is powered down).

2. Running a Custom Kernel in QEMU

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

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

- These are the commands for Ubuntu . If using a different distribution then consult the documentation for your system.
- If you are working in the CS labs (under Linux), then QEMU will already be installed.

NOTE: If you are in the CS labs, do not run the “`sudo`” command because it will be logged that you are trying to run commands that you do not have permission to run and will automatically notify our system administrators. You may, however, run `sudo` commands *inside* your VM (i.e., inside QEMU), but not on the host machine.

2. Download a small root file system from the following URL using a web browser (64-bit OS):

<https://people.debian.org/~aurel32/qemu/amd64/>

- From this URL, download the file using `wget`:
`debian_squeeze_amd64_standard.qcow2`
- Save into the `~/oscourse/` folder (the folder one level above `linux-stable/`)
- If using a 32-bit OS, you'll need to browse to here instead and download the i386 image:
<https://people.debian.org/~aurel32/qemu/i386/>

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

```
$ cd ~/oscourse/linux-stable/
```

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).

- Launch using the current terminal window (note “`sda1`” ends in a one, not an ‘L’):

```
$ 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
```

- 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 take around three minutes to complete booting. During this time, you should see many log messages to the screen.
- 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`

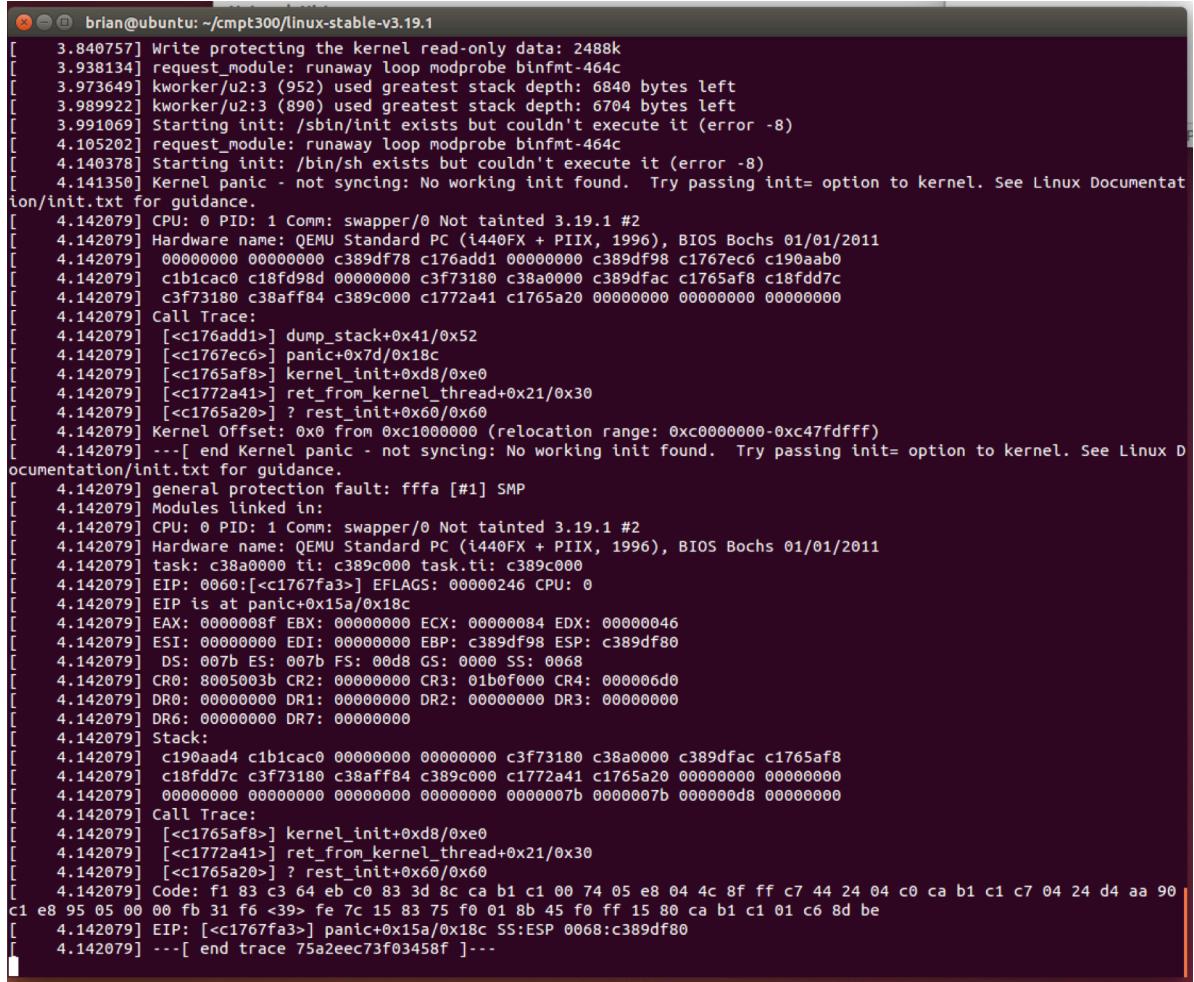
5. When the QEMU virtual machine has finished booting, you should see the following login prompt:

```
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: user
Password: user
 - Name: root
Password: root
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 you are done using the QEMU virtual machine, you should tell Linux running in it to power down:
`# poweroff`
9. 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-stable$ 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-stable$ 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.
10. 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 1 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 Documentation/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 c18fd7c
[ 4.142079] c3f73180 c38aff84 c389c000 c1772a41 c1765a20 00000000 00000000 00000000
[ 4.142079] Call Trace:
[ 4.142079] [<c176add1>] dump_stack+0x41/0x52
[ 4.142079] [<c1767ec6>] panic+0x7d/0x18c
[ 4.142079] [<c1765af8>] kernel_init+0xd8/0xe0
[ 4.142079] [<c1772a41>] ret_from_kernel_thread+0x21/0x30
[ 4.142079] [<c1765a20>] ? 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 Documentation/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: 8000003b 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] c18fd7c c3f73180 c38aff84 c389c000 c1772a41 c1765a20 00000000 00000000
[ 4.142079] 00000000 00000000 00000000 00000000 00000007b 00000007b 0000000d8 00000000
[ 4.142079] Call Trace:
[ 4.142079] [<c1765af8>] kernel_init+0xd8/0xe0
[ 4.142079] [<c1772a41>] ret_from_kernel_thread+0x21/0x30
[ 4.142079] [<c1765a20>] ? 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 01 c6 8d be
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 75a2eec73f03458f ]---

```

Figure 1: 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 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 (.qcow2 file). Redownload the root file system image.
 VFS: Cannot open root device "sdal" 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.
- 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.

3. Modify Kernel and Rebuild

1. If running on your own computer, install the necessary library to configure the kernel. (This is already done in the lab machines).

```
$ sudo apt-get install libncurses5-dev
```

2. Form within the `linux-stable/` 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 "`-wwu_id`", where `wwu_id` is **your** student number

4. Rebuild the kernel:

```
$ make
```

- 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
```

Example for the output: Linux debian-amd64 4.13.9-your_student_id #2 SMP Fri Nov 3 16:09:37 PDT 2017 x86_64 GNU/Linux

- Alternatively, you can check the kernel version inside your QEMU VM using:

```
# cat /proc/version
```

4. Build & Deploy Linux App to QEMU

You can use the SCP command to copy a file from your host machine into the QEMU virtual machine.

1. Programs that you want to copy into the QEMU VM should be compiled with static linking so that all necessary libraries are included in your executable. This can solve issues related to library versions.
 - 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 2222 on the host OS to the QEMU VM's port 22 (all on one line).
`$ 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:2222::22`
 - Port 22 is the default SSH port, which is also used for SCP.
 - The `-net` options explicitly tell QEMU to enable networking (usually done by default).
 - The `-redir` option redirects TCP traffic on the host's port 2222 (otherwise unused) to the guest's port 22 (SSH).
3. Copy your file (`helloWorld`, for example) to the QEMU virtual machine via SCP using the following command executed in the host OS:
`$ scp -P 2222 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 2222 helloWorld myApp fooFile2 root@localhost:~`
4. Run your application in the QEMU VM:
`# cd /root
./helloWorld`
 - These commands are run inside the QEMU OS, not the host.
5. Troubleshooting
 - 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)`
 - If you get the message “could not set up host forwarding rule ‘tcp:2222::22’” when launching QEMU, try using a different host port (such as 8383) instead of 2222.
 - If you get the following message, it means QEMU is either not running, or has not correctly been started with the `-redir` argument.
`ssh: connect to host localhost port 2222: Connection refused
lost connection`
 - If your system seems to hang when you call `scp`, or you get the error below, it likely means

that the OS running in QEMU may not have finished booting yet. Try waiting until the OS successfully boots and you can log in. You may also need to verify SSH is running in the guest OS (beyond the scope of this guide).

```
ssh_exchange_identification: read: Connection reset by peer  
lost connection
```

4.1 SSH into QEMU VM

You can use SSH to create additional terminals connecting to your QEMU virtual machine. You might want to do this if you want to run multiple different test programs at once.

1. Launch the QEMU virtual machine using the configuration described in the previous section.
2. SSH from host into guest VM

```
$ ssh root@localhost -p2222
```