Compiling Your Own Kernel

In this meeting, we will use several tools to:

- download the latest stable kernel
- customize the kernel config file
- compile the kernel and kernel modules
- set the default system kernel to be a kernel of your choosing
- explore the kernel source code using the LXR Cross Referencer

Getting the Kernel

The kernel source code is available at kernel.org. On the main page, there is a big yellow box that says **Latest Stable Kernel**. It may be tempting to press that button. Resist the urge: clicking that link will download the latest stable kernel as a compressed "tarball". A tarball is good for today, but we don't want to live in the present. The kernel source is actively developed by a heterogeneous community of developers, with updates and changes managed using git. We want to get the Linux source using git so that we can access the history of old kernel versions, as well as access new releases when they are published.

Step one for today's meeting is to find the "mainline" Linux source's git repository.

- Where is mainline kernel's git repository?

What are the meanings of the following terms (in relationship to the Linux development process), and how are they represented in the Linux git repository?

- mainline
- longterm
- stable
- prepatch

(Hint: Use git to clone this repository to a directory called `linux-stable-src/`, and explore the git repository using `git-log`, `git-branch`, and `git-tag`.)

- Now execute the command:

  ```
  $ uname -r
  ```

  What is the output?
- What is a "distribution" kernel?
- Is your system using a mainline or distribution kernel?

Configuring the Kernel

There are *many* good online guides for building your kernel. To build a kernel for an Ubuntu system, for example, this guide is great: `https://wiki.ubuntu.com/KernelTeam/GitKernelBuild`. However, it uses some Debian-specific tools to assemble the kernel into a Debian package, and then installs it using the Debian package framework. This only works on Ubuntu, so it will not work for our barebones `kvm` virtual machines. Plus, we don't just want to follow instructions, we want to understand what is going on when we execute the commands!

So, please `cd` into your `linux-stable-src/` directory and type:

```
$ ls -a
```

Do you see a file called `.config`? That is the file that we will edit to control what code is compiled into our kernels. Do not edit it directly!

There are several options for editing the config file if it exists (or creating it if it doesn't). Please document what the following options will do:

- make config
- make menuconfig
- make defconfig
- make oldconfig
- make olddefconfig
- make allyesconfig
- make allnoconfig
- make allmodconfig
- make randconfig
- make localyesconfig
- make localmodconfig

Please use `make defconfig`, followed by `make oldconfig`, followed by `make menuconfig`. You should see a wonderfully modern interface (just kidding). `make menuconfig` is the method most kernel builders use to control the features that are excluded from their kernel, built into their kernel directly, or built into their kernel as modules. You can use the space-bar to toggle between `[*]` or `[ ]` for kernel options that you want to include or exclude. For options that can optionally be built as modules, you can toggle between `<*>`, `< >` or `<M>` for yes, no, or module. Some options also take text/numeric inputs, and are noted with `( )`.

On this site, there is a very good description of the trade-offs of these choices:

> Code in the Linux kernel can be put in the kernel itself or made as a module. For instance, users can add Bluetooth drivers as a module (separate from the kernel), add to the kernel itself, or not add at all. When code is added to the kernel itself, the kernel requires more RAM space and boot-up time may take longer. However, the kernel will perform better. If code is added as modules, the code will remain on the hard-drive until the code is needed. Then, the module is loaded to RAM. This will reduce the kernel's RAM usage and decrease boot time. However, the kernel's performance may suffer because the kernel and the modules will be spread throughout the RAM. The other choice is to not add some code. For illustration, a kernel developer may know that a system will never use Bluetooth devices. As a result, the drivers are not added to the kernel. This improves the kernel's performance. However, if users later need Bluetooth devices, they will need to install Bluetooth modules or update the whole kernel.

The interface for configuring your kernel isn't slick, but it is functional. Use `/` to search for the string "localversion".

- What does the `CONFIG_LOCALVERSION` option do?

Use the search results to find and set the `CONFIG_LOCALVERSION` option to `wsp11`.

Finally, just play around with `make menuconfig`. To start, you can build a kernel without many changes from the system defaults. But there are things we obviously don't have, like Bluetooth, that we can turn off. In addition to making our kernel images smaller, turning unnecessary options off actually reduces the compile time.

When we run our kernels inside our virtual machines later, we will focus on building smaller, more efficient kernels (and in the case of `kvm` users, we will turn on some features take advantage of paravirtualization).

When you are satisfied with your configuration changes, save and exit. This will update the `.config` file in your current directory. If you ever want to save your .config file, you may want to copy it elsewhere—otherwise it will be overwritten by the next `make *config`.

Compiling your Kernel

Once you have configured, compiling is easy. In the simple case, there are four commands:

- make
- make modules
- make modules_install
- make install

Each command does something unique, so please try them and document what these commands do. (HINT: use `make -jX` where X is some number less than or equal to the number of cores on your processor. This option will allow your compiler to do things in parallel, and greatly speed up the amount of time it takes to build.)

There is a really good answer on stack-exchange (the accepted one) that describes each command in high-level detail. But the answer is vague. Before and after executing each command, look around your system to see what files get put where.

```
$ ls -l
$ ls -l /boot
$ ls -l /lib/modules
```

Please enhance the description of each step in entries on our own wiki; be sure to include the specifics of file names and what each file does. (Does it copy files, or create links? Are names based on kernel versions? Are there existing files that have equivalent functions (where did they come from)?)

Booting into your Kernel

When turn on your machine, the Grand Unified Bootloader GRUB is responsible for transferring control to your kernel. From the GNU GRUB website:

> Briefly, a boot loader is the first software program that runs when a computer starts. It is responsible for loading and transferring control to the operating system kernel software (such as the Hurd or Linux). The kernel, in turn, initializes the rest of the operating system (e.g. GNU).

Since we already had a kernel, and we just compiled a new one, it would be nice to be able to control which kernel we actually boot each time our machine turns on. There is a file `/etc/default/grub` that lets us edit the default behavior of GRUB. Here are the first few lines of that (unedited) file:

```
GRUB_DEFAULT=0
GRUB_HIDDEN_TIMEOUT=0
GRUB_HIDDEN_TIMEOUT_QUIET=true
GRUB_TIMEOUT=10
GRUB_DISTRIBUTOR=`lsb_release -i -s 2> /dev/null || echo Debian`
GRUB_CMDLINE_LINUX_DEFAULT=''quiet splash''
GRUB_CMDLINE_LINUX="
```

Please document the GRUB options. Once you have a basic understanding of these options, please set GRUB so that it always shows boot options, and it waits 10 seconds for input before choosing the default. Before editing this file, copy the original version to a safe location so that you have a backup.

Editing the file does not actually update the settings. The last step is to issue the command:n

```
$ sudo update grub
```

Running Your New Kernel

Now you can reboot your system and verify that everything works. If you configured it properly, GRUB should now display several options. Using the arrow keys, explore the GRUB menu. Can you find the kernel that you just built? Does it work?

LXR

The LXR Cross Referencer is a really useful tool for navigating source code. Go to lxr.linux.no/, then the *Linux 2.6.11 and later* link. At the top, you should see a drop-down menu of version numbers and a search bar. As actual page content, you should see a series of links along the left. Go to your Linux source directory and type `ls`. What do you notice?

This site is a web interface for navigating source code. You can search for functions, files, or just text strings. LXR was originally used for Linux (LXR used to be Linux Cross Referencer), but it can index any large project. Try and search for `link_path_walk` in the `3.11` version of the kernel. It is a function that appears in one file, so there are only a few links. Now try and search for `inode` in the `3.11` version of the kernel. The `inode` structure is one of the main data structures used by Linux file systems, so there are many many links. However, the LXR output does a nice job of categorizing the links by use so you can find exactly what you are looking. For example, I was searching for the `inode` structure definition, which is line 523 of `include/linux/fs.h`.

LXR is something you can download and run to index your own code. Doing so will generate a webpage that you (and others) can use to navigate your project.

Next Steps

We probably want to use the default Ubuntu distribution kernel on our Panics. However, compiling our own kernel was not a useless task: we want to compile a minimal kernel for our virtual machines.

If you are using VirtualBox:

- Inside your VirtualBox VM, repeat these steps, but compile the smallest kernel that you can. How small is your kernel? (You can start with a default config, then look through the different options and find things that you obviously don't need. Or you can start with all NOs, and turn options on. Feel free to consult the internet for this task (but don't just download a config that you don't understand).)

If you are using KVM:

- Replace the kernel in your KVM boot command with your own custom kernel. Does it boot? Now try to create a minimal configuration.
- In your kernel's config, find and enable all `virtio` guest features. Then, edit your KVM boot command to use the corresponding `virtio` drivers.
- How much faster does your VM boot with your minimal kernel then the one that was provided? How much faster does it take for the minimal kernel to compile?
- At the minimum, you should have a running VM with functioning networking. You should be able to `ssh` into and out of your VM.