

# Recompiling the Linux Kernel

Sheetal Joseph

21st Jan 2005

Some months ago my boss asked me to recompile the Kernel in some of the servers which I was managing. It seemed that these servers were running a pre-historic kernel. It was a clear open invitation for the hackers who are running around all over the place.

I suddenly felt I was in no mans land. Recompiling/patching the kernel!!! I was under the impression that it is a job meant for super-geeks only. I had no clue what I was supposed to do? I was gripped with fears like will I reduced an existing working system into some unusable form. I did a silent curse on all those malicious hackers out there, whose only purpose in their life seemed to me to cause lot of headaches and increase the overtime hours of poor sys-admins like me.

I tried sweet talking with my boss to let me out of the hook. But it seemed he won't relent and was quite adamant about the whole stuff. I desperately cried for help to Google. And out he spat out lot of results. Though I spend 2 entire working days on getting the whole procedure right (I know it is quite embarrassing), but I realised recompilation of kernel is no rocket science. In fact it is quite easy.

## 1 What is the kernel?

Kernel is the heart.... Nay, the brain of an Operating System. A computer is nothing but a machine, which we can instruct around, so as to achieve lots of things. So how will we instruct these machines? Well, there are programs that do it for us, which we collectively call as the Operating System. Kernel is the core OS program which does all the ground and dirty work of instructing different parts of the machine. Most of the programs that we use like Open Office, Gimp, xmms etc (application programs) gets its job done by asking Kernel to give certain services. Now, the kernel has to run around, communicate with lot of hardware and some software to get the finished product from these places and get it delivered to the application programs. Quite a thankless job, I guess.

## 2 Why is it important to recompile the kernel?

You see, if you want to get secret inside information about some institution, you will have to bribe, or find the vulnerabilities of the correct person. As far as a system is considered this "correct person" is the kernel. So that is what most of the hackers try to do.

Though we can be rest assured that the kernel won't take any sort of bribe. since he is managing lot of stuffs together, there is a high chance that he sometimes will accidentally spit out some secret information or will forget to close some doors to restricted areas. Now, c'mon, you should give this guy a break, He is doing such a lot of things for you. He is entitled to do putches. These vulnerabilities potentially a hacker can make use of, and gain access and control over a system. So it is crucial to notify the kernel the mistakes and tell him not to commit the same mistake again.

Another. maybe more common reason to update the kernel is to get more hardware support. As new and new hardware is added to the system, we have to familiarise these hardware to kernel, and also teach the Kernel to communicate with these devices.

Okay, now lets get into business, and discuss more technical details regarding how to recompile the kernel and other details.

Now that I have convinced you the need for a Kernel recompiling, let me ask the inverse question

## 3 Is Recompiling The Kernel Necessary?

Is recompiling the kernel necessary? Hmm...That depends on your situation (and who you ask). Here are a few things to think about when considering recompiling your kernel:

1. New kernels are released rather frequently and the difference between two consecutive patch levels is usually minimal. Updating your kernel every time a new kernel is released is usually pointless unless the new version addresses an issue that directly affects you.
2. In the past, the kernel was not as modular as it is today. This means that older kernels used to load many unneeded modules into memory, thus increasing system load and also increasing the chances that bugs in those unneeded modules could adversely affect the system. Recompiling the kernel to remove the unneeded modules had noticeable benefits. Newer kernels, however, usually load modules into memory only when they are needed. Manually removing these modules has little positive effect since they are not called by the kernel anyway.
3. Recompiling the kernel may be necessary if new hardware is added to the system that the current kernel does not support.
4. The process of compiling the kernel places a heavy load on the system, especially the RAM. On a busy server, there is be a noticeable degradation of system performance. Also, after the kernel has been compiled and installed, the system must be rebooted so that the new kernel

can be used. Depending on the role of the machine, the downtime involved with rebooting the server can be costly. Consideration should be given to the items listed above before recompiling the kernel.

## 4 There are so many Kernel Version!!! How do you decode these Kernel Version numbers?

The Linux kernel version numbers consist of three numbers separated by decimals, such as 2.2.14. The first number is the major version number. The second number is the minor revision number. The third number is the patch level version.

At any given time there is a group of kernels that are considered "stable releases" and another group that is considered "development." If the second number of a kernel is even, then that kernel is a stable release. For example, the 2.4.14 kernel is a stable release because the second number is even. If the second number is odd, then that kernel is a development release. For example, the 2.5.51 is a development release because the second number is odd. When the 2.5.x branch was considered finished, then it became the 2.6.0 kernel. Patches are appearing for the 2.5.x branch and active development work has begun on the 2.6.x branch. If the 2.5.x advancements were significant enough to be considered a major revision, the 2.5.x branch would have become 3.0.0 and development work would have begun on the 3.1.x branch.

Okay... Let's get into some action.. Now how will you go about the whole business of this recompilation.

## 5 Download the Kernel

First thing you have to do is to get hold of the required Linux Kernel source code. The archive of Linux kernels are available at <http://www.kernel.org/>. The latest stable release, while I am writing this document is 2.6.10. We are looking for a file called linux-2.6.10.tar.bz2 or linux-2.6.10.tar.gz (or similar), which is under the "F" link at the end of the line.

The source code for the kernel that is on your machine right now is in the directory '/usr/src/linux/'. It is wise to keep this particular source code safe, for example by renaming the linux directory as follows:

```
$ cd /usr/src
$ mv linux linux-2.4.21 (if the original source code is for 2.4.21).
```

Only after you have safely stored the original kernel should you unpack the newest kernel. In this example I rename the 'linux' directory right after unpacking the kernel source code 'linux-2.6.10', and I create a symbolic link with the name 'linux' to the directory 'linux-2.6.10'. The advantage

of this procedure is that you can see the system's present kernel version immediately. In addition it is easier to install a kernel upgrade. The commands are (as root, remember):

```
$ cd /usr/src
$ cp ~/linux-2.6.10.tar.bz2
( assuming that the tarball was downloaded to your home-directory ('~') )
$ tar -xjvf linux-2.6.10.tar.bz
(this may take a while)
$ mv linux linux-2.6.10
$ ln -s /usr/src/linux-2.6.10 /usr/src/linux
```

The final step in installing the source tree is to clean up any stale files:

```
$ cd linux
$ make mrproper
```

## 6 Kernel Configuration

This is a bit tricky stage. In a nutshell, this stage basically comprises of selecting which modules you want, and which you don't want. So it is rather good idea, for you to know what all devices and peripherals are plugged into your computer, and what all hardware you plan to add to your system in immediate future. Go through this phase a bit slowly. A well configured kernel will do a lot of good for you in future. Also, beware this is the stage where you potentially can screw up. If you get Kernel configuration right, then it is most highly likely that the rest of the procedure is a cake walk for you. Take a bit of your time and proceed carefully.

A full description of all the configuration options is really beyond the scope of this article. You can check following links to get a better overview <http://www.faqs.org/docs/securing/chap7sec80.html>

As I mentioned earlier configuring the kernel means specifying a lot of different options:

- Do you want this kernel to support modules, and if so, which ones? (this accounts for the bulk of configuration work)
- What processor will this kernel run on?
- Is this kernel for an SMP system?
- Do you want support for the PCI/ISA/EISA/MCA buses?

And so on, and on, and on..... :o .... There are hundreds of options, many relating to support for hardware such as SCSI and RAID controllers, flash devices, as well as network options, filesystem

options and the like. Then there are the device drivers - for network cards, ISDN cards, serial cards, audio cards, and so on, each of which can be compiled either directly into the kernel or as a module for loading after the system has booted.

The kernel configuration options are stored in a file called ".config". There are three ways of producing this file: make config, make menuconfig, make xconfig

**make config:** This command asks about each option in turn. Most questions can be answered with Y or N

```
$ make config
rm -f include/asm
( cd include ; ln -sf asm-i386 asm)
/bin/sh scripts/Configure arch/i386/config.in
#
# Using defaults found in arch/i386/defconfig
#
*
* Code maturity level options
*
Prompt for development and/or incomplete code/drivers (CONFIG_EXPERIMENTAL) [N/y/?]
```

The script prompts, showing the possible answers - in this case "N", which is the default (shown first in upper case) if you press Enter, "Y", or you could press "?" to get some online help or explanatory text. In the case of device drivers, the question can usually be answered with "Y", "N" or "M" - "Y" compiles the driver into the kernel, while "M" will cause it to be built as a separate module. For example:

```
Power Management support (CONFIG_PM) [Y/n/?]
Advanced Power Management BIOS support (CONFIG_APM) [N/y/m/?]
```

**make menuconfig:** This command will provide a full-screen menu-driven interface which reads the existing .config file upon startup and allows you to adapt or fine-tune it. Operation is fairly intuitive, with navigation by the arrow and Tab keys, with Enter to select; at the end of the procedure, you are asked "Do you wish to save your new kernel configuration?" and can answer "Y" or "N" as appropriate.

**make xconfig:** Assuming that the Tk widgets are installed and you are running an X server, this will produce a graphical interface which will allow you to step through the various options (usually by clicking on the "Next" button) or refine an existing .config file by selecting just the desired options.

## 7 Build Dependencies

The next step is to create the necessary include files and generate dependency information. This step is only required for the 2.4.x kernel tree.

```
$ make dep
```

Lots of messages will scroll by. Depending on the speed of your machine and on what options you chose, this may take several minutes to complete. Once the dependency information is created we can clean up some miscellaneous object files. This step is required for all versions of the kernel.

```
$ make clean
```

## 8 Build the Kernel

We are now (finally) ready to start the actual kernel build. At the prompt type:

```
$ make bzImage
```

## 9 Build the Modules

There is one more step needed for the build process, however. You have created the kernel, but now you need to create all the loadable modules if you have them configured. To build the modules we run:

```
$ make modules
```

Again, lots of messages will scroll by on the screen. Here also the 2.6.x series is less talkative, outputting only summary information. Once the modules are built they can be installed. If you were building as a non-privileged user you will now need to switch to root to complete this next step:

```
$ su
password:
root# make modules_install
```

The freshly baked modules will be copied into `/lib/modules/KERNEL_VERSION`.

## 10 Create Initial RAMDisk

If you have built your main boot drivers as modules (e.g., SCSI host adapter, filesystem, RAID drivers) then you will need to create an initial RAMdisk image. The `initrd` is a way of sidestepping the chicken and egg problem of booting – drivers are needed to load the root filesystem but the filesystem cannot be loaded because the drivers are on the filesystem. The answer to this problem is to create an Initial Root Disk (or Initial Ram Drive) - the kernel loads a small filesystem image into memory after itself and then loads the required modules from there. This image is created with the `mkinitrd` script, which reads `/etc/fstab` and `/etc/modules.conf` (looking for `scsi_hostadapter` entries), and will then create an `initrd` file containing the appropriate modules. The basic syntax is

```
mkinitrd image-file kernel-version
```

In our case

```
root# mkinitrd /boot/initrd-2.6.10.img 2.6.10
```

## 11 Installation-Copy the Kernel and System.map

Once your kernel is created, you can prepare it for use. From the `./linux` directory, copy the kernel and `System.map` file to `/boot`. In the following examples change `KERNEL_VERSION` to the version of your kernel.

```
root# cp arch/i386/boot/bzImage /boot/bzImage-KERNEL_VERSION
root# cp System.map /boot/System.map-KERNEL_VERSION
root# ln -s /boot/System.map-KERNEL_VERSION /boot/System.map
```

The next step is to configure your bootloader. Since the default Red Hat 9 (the distro in my system) has GrUB as a boot loader I will be explaining how to configure grub.

## 12 GrUB Configuration

Once you have copied the `bzImage` and `System.map` to `/boot`, edit the grub configuration file located in `/etc/grub.conf`

```
# Note that you do not have to rerun grub after making changes to this file
```

```
#boot=/dev/hda
default=0
timeout=10
title Red Hat Linux (2.4.21)
root (hd0,1)
kernel /boot/vmlinuz-2.4.21 ro root=LABEL=/
initrd /boot/initrd-2.4.21.img
```

Edit the file to include your new kernel information. Keep in mind that GrUB counts starting from 0, so (hd0,1) references the first controller, second partition. If you have created an initial RAMdisk be sure to include it here too. A typical configuration may look something like this:

```
title Test Kernel (2.6.10)
root (hd0,1)
kernel /boot/bzImage-2.6.10 ro root=LABEL=/
initrd /boot/initrd-2.6.10.img
```

When the system restarts, select "Red Hat Linux with 2.6.10 Kernel" and watch carefully for any error messages. If this is your first attempt at building a kernel, there will almost certainly be several! You might have forgotten to compile a module that is required by a network card, sound card or other peripheral. A common mistake is forgetting to compile the ext3 support into the kernel, or forgetting to run mkinitrd if you've compiled it as a module.

Make a note of the error messages, and see if you can figure out what you've forgotten. Then reboot, select the old kernel (it's still the default at this stage anyway) and go back to the make xconfig stage to fix up your kernel configuration. Repeat as required.

Good Luck Folks!!! and Have Fun

## References

1. <http://www.kernel.org/>
2. <http://www.kernelnewbies.org/>
3. <http://kerneltrap.org/>
4. <http://kerneltrap.org/node/799?PHPSESSID=3c160a1faa3a472b5a6071c606fbde61>
5. <http://thomer.com/linux/migrate-to-2.6.html>
6. <http://linuxguide.sourceforge.net/linux-kernel.html#compile-configure>
7. <http://linuxreviews.org/sysadmin/kernel-configuration/>
8. <http://www.gentoo.org/doc/en/handbook/handbook-hppa.xml?part=1&chap=7>
9. <http://www.linuxheadquarters.com/howto/tuning/kernelconfig.shtml>