Building a Linux Kernel from source

This is how I build my 2.6 kernels.

X and su

I run the commands from an X terminal, and at some point start the X based kernel configurator. I run my desktop as “myself” but I build my kernels as root. In order to let root use my X display, I do the following in my X terminal: get root rights; merge my own (alien's) Xauthority file with the one from the root user, and set the DISPLAY variable. After doing that, I can run X applications from the “su” terminal.

```
echo $DISPLAY                  # you'll be needing this value 3 lines below
sudo -i                        # or "su -" on older Slackwares
xauth merge ~alien/.Xauthority # use your own username here instead of
"alien"
export DISPLAY=:0.0            # use the value of DISPLAY you've seen 3
lines before
```

Alternatively, you can run the following two commands which will give you the same end result:

```
sudo -s                        # a side effect of '-s' is that it allows
root to run X programs
. /etc/profile                 # sourcing the global profile ensures
                               # that root has the ''sbin'' directories in
the $PATH
```

Downloading and configuring

Now that the build environment is set up, let us continue with obtaining the sources.

Download a new kernel, unpack it into `/usr/src` and create the “linux” link so that the commands are a little more generic. I will take a kernel version of “2.6.37.6” as an example. If yours is a different version, you'll know where to change the version strings in the rest of the story below. If you want to know how to verify the integrity of the source code archive using the kernel GPG key, read the last chapter below.

```
wget http://www.us.kernel.org/pub/linux/kernel/v2.6/linux-2.6.37.6.tar.bz2
tar -C /usr/src -jxvf linux-2.6.37.6.tar.bz2
cd /usr/src
rm linux                     # remove the existing symlink
ln -s linux-2.6.37.6 linux   # create a symlink pointing to your new linux
source
```

Changing the “`linux`” symbolic link is safe to do. No applications will break if you let it point to another kernel than the one Slackware installed for you. You will probably notice more linux-* directories in `/usr/src`. It is common to let the “`linux`” link point to the kernel you are currently

working with. It is however no *requirement* to have this symbolic link. Modern software that needs to know the location of the source code of an installed kernel will look at where the symbolic link `/lib/modules/<kernelversion>/build` points to instead.

> There is a debate whether you should build your kernels in the `/usr/src` tree or somewhere entirely else.
> The cause is an old post by Linus Torvalds (from July 2000) where he advises people to build from within their home directory. I believe this advice is irrelevant for Slackware and the way it has it's kernel headers and glibc package setup. So, **my** advice is to ignore this old post by Linus and install your kernel sources into `/usr/src` if you want. The kernel building location is purely a matter of personal preference.

Now, get a Slackware kernel config file for a headstart during your own configuring. Pat's config files are pretty generic. By the time you read this, there might be a config for a newer 2.6 release available:

```
wget
http://slackware.mirrors.tds.net/pub/slackware/slackware-13.37/source/k/conf
ig-generic-smp-2.6.37.6-smp
cp config-generic-smp-2.6.37.6-smp /usr/src/linux/.config
```

Alternatively, you can grab the configuration for the kernel which is currently running:

```
zcat /proc/config.gz > /usr/src/linux/.config
```

Run *make oldconfig* in the kernel source directory so that the defaults are used from the .config file you just installed. Because your kernel sources are probably newer than the .config, there will be new options to choose from. You will only have to answer these (press ENTER for the default answers which are mostly fine, or M to build new drivers as modules).

```
cd /usr/src/linux
make oldconfig
```

You now have configured a pretty generic kernel (that is the reason why Pat calls them “kernel-generic” probably 🙂 but you will want to change some of the defaults to suit your needs. Run the X based configurator (if you do not run X but are at a text console, just run “`make menuconfig`” to get the curses-based dialog program instead)

```
make xconfig
```

Walk through the forest of options. What I usually change are things like:

- Build the ext3 (needs the jbd driver as well) and reiser/xfs/jfs fileystem drivers into the kernel instead of compiling them as modules - that way I do not need to create an additional “initrd“ (see under “Filesystems” in the configurator).
- Enable 64GB of RAM.
  (under “Processor type and features” > “High Memory Support (64GB)”). Use this if you have a

system with 4GB of RAM or more.

- Enable the “low-latency” kernel if you run a desktop/laptop computer - multimedia apps will run much smoother
(under “Processor type and features” > “Preemption model” > “Preemptible kernel”). If you run a desktop system with a lot of multimedia applications, then this is an useful option for you because it will keep your system more responsive even under high load.
- Set a 1000 Hz timer (under “Processor type and features” > “Timer Frequency” > “1000 Hz”). A higher tick count can be beneficial to multimedia 'desktop' systems.
- Or set a tickless timer (dynamic ticks - under “Processor type and features” > “Tickless System (Dynamic Ticks)”).
- If you are (re-)building a Slackware kernel, you should make sure that installing your new kernel will leave the original kernel modules intact. You do this by changing the *local-version* part of the kernel's release number to an unique string (under “General setup” > “Local version - append to kernel release”). This kernel option corresponds to **CONFIG_LOCALVERSION** in your `.config` file. Slackware sets that value to ”-smp” for a SMP kernel to give you an idea.
The resulting kernel release value (as returned by “`umake -r`”) for a kernel version “`2.6.37.6`” with a local-version of “`-alien`” would be “`2.6.37.6-alien`”
- … and more I can't think of right now. You can decide to disable a lot of the modules that the default config will build, to cut down on compilation time, if you don't have the hardware in your computer. You could also look at *software suspend* and *CPU frequency scaling* (under “Processor type and features”) if you own a laptop.

And finally save your configuration if you're satisfied.

Building your kernel

Now, start the build of kernel and modules, and install them to the proper places.

```
make bzImage modules            # compile the kernel and the modules
make modules_install            # installs the modules to
/lib/modules/<kernelversion>
cp arch/x86/boot/bzImage /boot/vmlinuz-custom-2.6.37.6  # copy the new
kernel file
cp System.map /boot/System.map-custom-2.6.37.6          # copy the
System.map (optional)
cp .config /boot/config-custom-2.6.37.6                 # backup copy of
your kernel config
cd /boot
rm System.map                                           # delete the old
link
ln -s System.map-custom-2.6.37.6 System.map              # create a new link
```

With 2.6.x kernels, running “`make`” or “`make all`” instead of “`make bzImage modules`” should be sufficient. This will build the default targets, being *vmlinux* (the uncompressed kernel), *bzImage* (the compressed kernel which we will be using) and *modules* (all of the kernel modules). Since we do not need the uncompressed kernel I usually stick to the “`make bzImage modules`” command.
If you want to know more about the available *make* targets, you can run “`make help`” and examine

the output. The default build targets are marked by an asterisk (*).

Modifying lilo.conf

Edit `/etc/lilo.conf` and add a new section for your new kernel. Remember, your new kernel may not even boot if you made a mistake somewhere, so you will want to leave the existing sections for your current kernel(s) intact. Your current `/etc/lilo.conf` will have a section somewhat like this, near the bottom of the file:

```
image = /boot/vmlinuz
root = /dev/sda1
label = linux
read-only # Non-UMSDOS filesystems should be mounted read-only for checking
```

Add another section just below (adding it below will guarantee that your current - working - kernel will stay the default to boot):

```
image = /boot/vmlinuz-custom-2.6.37.6
root = /dev/sda1
label = newkernel
read-only # Non-UMSDOS filesystems should be mounted read-only for checking
```

After you have added a section for your new kernel to `/etc/lilo.conf` you should save the file, and then run `lilo` to activate your changes:

```
lilo
```

Now is the time for a reboot, and test that new kernel! When the lilo boot screen appears, select the "`newkernel`" option instead of the default "`linux`" option.
If your new kernel boots fine, you can make it the default kernel by adding the following line to the top of `/etc/lilo.conf` and re-run "lilo":

```
default = newkernel
```

Slackware kernel-headers package

You are going to build and use a new kernel. You may wonder what you need to do with the Slackware *kernel-headers* package.
The answer is: **do not remove this package!**

There are two places where you will find kernel headers; one place is inside the kernel source directory (in our case, the `/usr/src/linux-2.6.37.6` directory) and the other place is `/usr/include/linux`. The *kernel-headers* package usually contains the headers taken from the source of the default Slackware kernel. These particular headers are used when the *glibc* package is built. The fact that the `kernel-headers` package installs these files to `/usr/include/linux` makes them independent of the header files you find in the kernel source directory.

As long as you do not upgrade your *glibc* package, you should not upgrade or remove the `kernel-headers` package.

- How do the `kernel-headers` and `glibc` packages relate?
  At some point in time you will want to upgrade (recompile!) parts of your system's software. If that software links against glibc (as most core software does), it's successful compilation relies on the presence of the correct kernel headers in `/usr/include/linux`. It does not matter that you are propably running an entirely different kernel than Slackware's default kernel. The `kernel-headers` package reflects the state of the system at the time `glibc` was built. If you delete the `kernel-headers` package, your running system will in no way be affected, but you will not be able to (re-)compile most software.

- Do the kernel sources still serve a purpose after you have built your new kernel?
  In the previous bullet point I told you that the compilation of system software uses the headers located in `/usr/include/linux`. Likewise, the kernel source tree is required any time you want to compile a 3rd-party kernel module (madwifi, linux-uvc, ndiswrapper, … the list is endless). You are not limited to compiling a driver for your running kernel. You can build drivers for any kernel as long as its modules tree (below `/lib/modules`) and sources are present.
  Let's say you are going to build a module for a kernel whose version you have specified in an environment variable $KVER. For instance, by running

  ```
  export KVER=2.6.38.2
  ```

  During the driver's compilation you will need to have this particular kernel's header files available in the location `/lib/modules/$KVER/build/include/linux`. The symbolic link `/lib/modules/$KVER/build` is created at the time when you install your new kernel and modules.
  If you delete the kernel sources after you build a kernel, you will not be able to build any “out-of-kernel” (other word for 3rd-party) drivers afterwards.

Other packages that contain kernel modules

Most certainly you will have installed one or more packages containing kernel modules that are not part of the default kernel. Slackware installs the “svgalib-helper” package for instance, and if you installed any wireless driver, these are basically kernel modules too.
Be aware that by installing and booting into your new kernel, you will no longer have these out-of-kernel modules available. You have to recompile their sources so that the resulting kernel modules match the version of your new kernel.
You can get an overview of all packages that have installed a kernel module for your current kernel by running this command (*note that you must run this command while still running your old kernel*):

```
cd /var/log/packages
grep -l "lib/modules/$(uname -r)" *
```

All the listed packages will need a recompile if you want to have their modules available for your new kernel as well.

If you rebuild a package containing a kernel module, use **installpkg** rather than **upgradepkg** to install this new package without removing the original version.
If you used `upgradepkg` this would remove the old kernel module and you may still need this if you ever want to reboot into your old kernel. This trick works under the assumption that the version of the kernel is part of the *VERSION* of your package, like here: `svgalib_helper-1.9.25_`*`2.6.37.6`*`-i486-1.txz` (I know, lame example since this package no longer exists)

The method described above will leave you **unaware** of any kernel modules that you may have compiled manually instead of creating a package for them. Typically, proprietary graphics drivers such as those from *Nvidia* or *Ati* will cause a few anxious moments if you forget to recompile them for your new kernel before starting X Window... especially when your computer boots into the graphical *runlevel 4* by default.

In that case, reboot into *runlevel 3*, download the latest version of your graphics driver and compile/install the driver. This will enable you to reboot into your graphical logon. For those who forget, booting into another than the default runlevel is easy: when the LILO screen appears, type the label of your kernel (which was `newkernel` in our example above) and after that, type the runlevel number: Newkernel Space3Enter

Creating an initrd

In case your kernel does not include the driver for your root filesystem, or a driver for your SATA bus, or other stuff that is only built as modules, your kernel will panic if it boots and can not access the necessary disks, partitions and/or files. Typically, this looks like

```
VFS: Cannot open root device "802" or unknown-block (8,2)
Please append a correct "root=" boot option
Kernel Panic-not syncing: VFS: unable to mount root fs on unknown block(8,2)
```

and this means you will have to build an *initrd* or “Initial Ram Disk” containing the required modules. The location of the initrd is then added in the appropriate section of `/etc/lilo.conf` so that the kernel can find it when it boots, and is able to load the drivers it needs to access your disks. Creating an initrd is quite simple, and I will show two cases here, one in care you have a Reiser filesystem on your root partition, and the second for the case you have an ext3 filesystem. I assume a 2.6.37.6 kernel in these example commands, if your new kernel is different, change the version number as appropriate.

- Change into the /boot directory:

```
cd /boot
```

- Run “mkinitrd” to create the `/boot/initrd` file containing a compressed filesystem with the modules you tell it to add on the commandline:

```
mkinitrd -c -k 2.6.37.6 -m reiserfs
```

for a Reiser filesystem, or

```
mkinitrd -c -k 2.6.37.6 -m ext3
```

in case you installed an ext3 filesystem on your root partition..

- Add the line “initrd = /boot/initrd.gz” to the newkernel's section in the file `/etc/lilo.conf`, save your changes and then re-run lilo; I will use the `lilo.conf` example section I already used in a previous paragraph:

```
image = /boot/vmlinuz-custom-2.6.37.6
  root = /dev/sda1
  initrd = /boot/initrd.gz
  label = newkernel
  read-only # Non-UMSDOS filesystems should be mounted read-only for
checking
```

then run

```
lilo
```

On next boot, your new kernel will not panic.

- If you are already using an initrd image with your current kernel, you can choose between two options:
  - Create a second initrd image using the command above but with an explicit name for the resulting initrd file (which should be different from the default in order not to overwrite the old one:

    ```
    mkinitrd -c -k 2.6.37.6 -m ext3 -o /boot/initrd-custom-2.6.37.6.gz
    ```

    and then change the `lilo.conf` section to look like this:

    ```
    image = /boot/vmlinuz-custom-2.6.37.6
      root = /dev/sda1
      initrd = /boot/initrd-custom-2.6.37.6.gz
      label = newkernel
      read-only # Non-UMSDOS filesystems should be mounted read-only
    for checking
    ```

  - Add the kernel modules for your new kernel to the existing initrd file. That way, you have a single initrd image containing modules for multiple kernels. All you need to do is leave out the option “`-c`” which is the option to wipe the directory `/boot/initrd-tree` and start from scratch:

    ```
    mkinitrd -k 2.6.37.6 -m ext3
    ```

I have written a shell script (mkinitrd_command_generator.sh) which examines your running Slackware system and shows you an example `mkinitrd` command. If you run that `mkinitrd` command, it will produce an initrd image that contains all the kernel modules and support libraries so that your system can boot with the Slackware *generic* kernel.
Here is an example of how to run the command with it's output shown as well:

```
/usr/share/mkinitrd/mkinitrd_command_generator.sh /boot/vmlinuz-generic-2.6.37.6
#
# mkinitrd_command_generator.sh revision 1.45
#
# This script will now make a recommendation about the command to use
# in case you require an initrd image to boot a kernel that does not
# have support for your storage or root filesystem built in
# (such as the Slackware 'generic' kernels').
# A suitable 'mkinitrd' command will be:

mkinitrd -c -k 2.6.37.6 -f ext3 -r cryptslack -m mbcache:jbd:ext3 -C /dev/sda8 -u -o /boot/initrd.gz
# An entry in 'etc/lilo.conf' for kernel '/boot/vmlinuz-generic-2.6.37.6' would look like this:
# Linux bootable partition config begins
# initrd created with 'mkinitrd -c -k 2.6.37.6 -f ext3 -r cryptslack -m mbcache:jbd:ext3 -C /dev/sda8 -u -o /boot/initrd.gz'
image = /boot/vmlinuz-generic-2.6.37.6
  initrd = /boot/initrd.gz
  root = /dev/mapper/cryptslack
  label = 2.6.37.6
  read-only
# Linux bootable partition config ends
```

You may notice that it detect my LUKS-encrypted root partition.
This script is included in the *mkinitrd* package of the Slackware releases after 12.2.

Loading modules on boot

Prior to Slackware 11.0, modules for your kernel were loaded either by the hotplug subsystem, or by explicit `modprobe` commands in the file `/etc/rc.d/rc.modules`. Having the same `rc.modules` file for 2.4.x and 2.6.x kernels was not an optimal situation.
In Slackware 12.0 and newer the 2.6 kernel the only kernel that is available. The loading of kernel modules is handled by udev and by explicit `modprobe` commands: the modules that are not loaded by udev can still be put in a `rc.modules` file. Only, there can now be more than just one file.
Slackware will look for the existence of the following (executable) files in this order:

- If `/etc/rc.d/rc.modules.local` exists, it will be run
- Else, if `/etc/rc.d/rc.modules-$(uname -r)` exists, it will be run
- Else, if `/etc/rc.d/rc.modules` exists, it will be run

The **$(uname -r)** is the current kernel release. If your kernel version is `2.6.37.6-smp`, then Slackware will look for a file `/etc/rc.d/rc.modules-2.6.37.6-smp` to run. This way, specific *rc* files for different kernels can be present, allowing an optimal tuning for your system.

The Slackware 13.37 package `/slackware/a/kernel-modules-smp-2.6.37.6_smp-i686-1.txz` will install the file `/etc/rc.d/rc.modules-2.6.37.6-smp`. You can use that as an example if you want to build your own kernel and need explicit `modprobe` commands for specific kernel modules.

If you decide to build your own 2.6 kernel from source, you might get bitten by the fact that there will not be a file called `/etc/rc.d/rc.modules-$(uname -r)` - you will have to create it yourself. The `rc.modules` usually is a symlink to the `rc.modules-2.6.37.6-smp`. A typical result from the absence of a rc.modules file for your specific kernel is that your mouse will not be working. Take that behaviour as a hint to create the `rc.modules` file! You can take a full copy of any existing `rc.modules-2.6.xx` file. If your system does not have any rc file for a 2.6 kernel you can take the one on the Slackware CD as an example: `/source/k/kernel-modules-smp/rc.modules.new`.
Here's an example in case you would have built a new kernel with version `2.6.38.2.alien` and you already had installed a Slackware kernel `2.6.37.6-smp`:

```
cp -a /etc/rc.d/rc.modules-2.6.37.6-smp /etc/rc.d/rc.modules-2.6.38.2.alien
```

The file `/etc/rc.d/rc.modules-2.6.38.2.alien` will then be used when your new kernel `2.6.38.2.alien` boots.

GPG signature

The Linux kernel source archives are signed with the OpenPGP “Linux Kernel Archives Verification Key”. This is a means to verify that the source code you downloaded is the original archive and has not been tampered with. The steps for this validation are outlined in this chapter.

- First, import the OpenPGP key into your GnuPG keyring; either by copying the key from the signature page or by importing it from a keyserver. The kernel key ID is `0x517D0F0E`. An example goes like this:

  ```
  gpg --keyserver wwwkeys.pgp.net --recv-keys 0x517D0F0E
  ```

  The resulting output will be somewhat like this:

  ```
  gpg: key 517D0F0E: public key "Linux Kernel Archives Verification Key <ftpadmin@kernel.org>" imported
  ```

```
gpg: Total number processed: 1
gpg:               imported: 1
```

- Next, get the signature file for the kernel archive you've downloaded:

```
wget
http://www.us.kernel.org/pub/linux/kernel/v2.6/linux-2.6.37.6.tar.bz2.s
ign
```

  and make sure it is in the same directory as the kernel archive itself.

- The final step is to run gpg on this signature file, and check what it has to report:

```
gpg --verify linux-2.6.37.6.tar.bz2.sign linux-2.6.37.6.tar.bz2
```

  The output will be like this:

```
gpg: Signature made Mon Mar 28 01:08:08 2011 CEST using DSA key ID
517D0F0E
gpg: Good signature from "Linux Kernel Archives Verification Key
<ftpadmin@kernel.org>"
gpg: WARNING: This key is not certified with a trusted signature!
gpg:          There is no indication that the signature belongs to the
owner.
Primary key fingerprint: C75D C40A 11D7 AF88 9981  ED5B C86B A06A 517D
0F0E
```

  If you would have told gnupg to trust this key, the last part would have looked different. To me, adding a trustlevel to this key makes no real sense, except when you've met one of the kernel developers who had the key with him and could present thrustworthy credentials.

Nevertheless, the outcome is that the source code archive was indeed signed with the key that you just imported. So, that is good news.

Translations

- This article has been translated into Spanish by Jorge Courbis: http://coredump.cl/wiki/doku.php?id=compilaci%C3%B3n_de_kernel