Yet Another Kernel Compile Guide

Why and How?

A few years ago I wrote a couple of HowTos on compiling and packaging a complete kernel+modules package. Recently, I decided to look at how the stock Slackware kernel packages are built. I put this guide together from what I picked up.

There are already a few kernel compile guides around, but I haven't seen one yet that looks at using Slackware's own kernel build scripts.

A “build-all-kernels.sh” script executes all the Slackbuilds, one after the other, passing all relevant variables to them, and creating a set of packages. As you'd expect, the script needs to be run as root.

At the end of this document I have an example of a custom “setup.sh” file that can be used to set various options and then run the main build script.

The point of using a setup file is:

- To save us some typing and, more importantly, to avoid typos.
- To make it easier to see and change the settings we want to use.
- Perhaps use different setup files to create different “recipes”.

(A recipe dictates the arch of the package set.)

The Slackware kernel build scripts can be found in the k/ source directory on any mirror of Slackware -current. This contains the 64 and 32 bit generic and huge configs, and a kernel source tarball. The 14.2 source directories have a different setup, but the scripts in -current can be used for 14.2.

There isn't a firmware build script in there so I won't be looking at making a package for that.

The build routine looks like this:

- Build a package using the newest version source found
- Move the package to $OUTPUT
- Install the package
- Loop to the next package…

At its simplest, you can mirror the k/ directory and then run the build script. This will build and install a set of kernel packages, using the newest version source archive found in the current directory. There is no need to manually set the kernel version.

We can change a few things in the build routine, such as:

- The local version (explanation below.)
- The number of make jobs.
- The build directory.
- The directory that packages are moved to on completion.
- Whether the packages are installed on the build system.

Some minor tweaking of the scripts is necessary if we change the default build directory, and we may

not want to clobber the existing symlinks in /boot if we are making a custom kernel.

For this example I will build packages for linux-5.0.8, with “jabberwok” as a local version, and /home/kernels/jabberwok as the output directory.

The resulting packages will be named:

kernel-generic-jabberwok-5.0.8_jabberwok-x86_64-1.txz
kernel-headers-5.0.8_jabberwok-x86-1.txz
kernel-huge-jabberwok-5.0.8_jabberwok-x86_64-1.txz
kernel-modules-jabberwok-5.0.8_jabberwok-x86_64-1.txz
kernel-source-5.0.8_jabberwok-noarch-1.txz

It's worth noting here that the kernels and modules have “jabberwok” in their names and versions, while headers and source only have it in their versions. This means that the kernels and modules are distinct and apart from the stock Slackware packages, and shouldn't affect them. I.E. a huge local version kernel will only upgrade another huge kernel with the same local version. However, the headers and source packages *will* upgrade the currently installed stock packages.

If you use slackpkg then the kernel-source and kernel-headers will need blacklisting to avoid being downgraded back to the stock versions.

If upgrading the stock kernels then you would also need to blacklist those for the same reason.

Assumptions:

- A Slackware 14.2 or -current install with all the build tools.
- Access to a slackware(64)-current/source/k/ directory.
- A source tarball for the linux kernel in tar.xz or tar.gz format.
- Root.

But:

This guide doesn't go into making kernel configs in any depth. Typing “make help” in the kernel source directory will show the various methods available. If you are going to stray far from the default Slackware configs then you ought to know what hardware you have, or enjoy testing random configurations and frequent reboots : -)

Build Variables

Some variables that can be used to change the behaviour of the build script:

OUTPUT:

Optional. Where the build script will move the completed packages to.
Default if unset: auto from $TMP, recipe and version. E.G.: ${TMP}/output-x86_64-${VERSION}

INSTALL_PACKAGES:

Optional. Whether to install the packages. Set this to anything other than “YES” or “” to not install the packages, which is useful when building packages to use on a different machine.
Default if not set: YES

NUMJOBS:

Optional. Number of make jobs. If make fails with this then it is run again with no jobs number set.
Default if not set: -j7

TMP:

Optional. Package build directory. If changing this then the modules Slackbuild must be adjusted (see below.)
Default if not set: /tmp

RECIPES:

Optional: The ARCH list to build for. Values can be any of x86_64, IA32_NO_SMP, IA32_SMP. Anything else will cause the build script to disown you.
Default if not set: automatic from `uname -m`

There are other variables used in the Slackbuilds but these are the most significant for our purpose.

Mirroring The Kernel Slackbuild Directory

There are a few ways of mirroring the directory - lftp, wget, or rsync if the mirror supports it.

With lftp:

```
lftp https://mirrors.slackware.com -e \
  "mirror -v /slackware/slackware64-current/source/k"
```

If you don't want to download the source tarball (e.g. because you already have one ready to use) then instead do:

```
lftp https://mirrors.slackware.com -e \
  "mirror -v -X *xz -X *sign \
  /slackware/slackware64-current/source/k"
```

Note that I am using the -current source folder here because it contains the newest configs, and they are closest to the target kernel version.

When the mirror is created, untar the kernel source archive in k/ and cd into the source directory:

```
cd k
tar xf linux-5.0.8.tar.xz
cd linux-5.0.8
```

Creating Kernel Configs

I will make both a “generic” and “huge” config, because the build script uses those by default. If you wish to build only one of those, comment out or remove the section for the unwanted config from the main build script.

The config files have the following naming convention:

config-<type>-<version>[suffix]

Where:

<type> is “generic” or “huge”.
<version> matches the kernel source version.
[suffix] is either “.x64” or nothing.

E.G. a generic 64 bit kernel config for linux-5.0.8 would be:

config-generic-5.0.8.x64

Copy two existing configs:

```
cp ../kernel-configs/config-generic-4.19.36.x64 \
  ../kernel-configs/config-generic-5.0.8.x64

cp ../kernel-configs/config-huge-4.19.36.x64 \
  ../kernel-configs/config-huge-5.0.8.x64
```

Edit the new files and customise CONFIG_LOCALVERSION. E.G.:

CONFIG_LOCALVERSION=“-jabberwok”

Note that the preceding '-' is needed.

As noted above, setting a local version will ensure that the new kernel packages won't upgrade the currently installed stock packages, unless of course you *want* them to, in which case leave it empty. The kernel-headers and source packages *will* upgrade those though, regardless of what we do here, but if the new kernel is expected to be the main kernel then this shouldn't be a problem.

The local version will also show in `uname` output:

```
Linux thing 5.0.8-jabberwok #1 SMP Sun Apr 21 20:38:51 BST 2019 x86_64
Intel(R) Core(TM) i5-2520M CPU @ 2.50GHz GenuineIntel GNU/Linux
```

The configs need to be in a state so that they will build without prompting when the build is started. The easiest way to do this is by copying the two new configs one at a time into the kernel source directory as .config, run “make olddefconfig”, and then copy them back:

Generic:

```
cat ../kernel-configs/config-generic-5.0.8.x64 > .config
make olddefconfig
cat .config > ../kernel-configs/config-generic-5.0.8.x64
```

Huge:

```
cat ../kernel-configs/config-huge-5.0.8.x64 > .config
make olddefconfig
cat .config > ../kernel-configs/config-huge-5.0.8.x64
```

The output of both of these make commands should be:

```
scripts/kconfig/conf  --olddefconfig Kconfig
#
# configuration written to .config
#
```

If you get different output then something is probably wrong.

The two configs now have all of Slackware's kernel settings, and with any new settings set to their default values.

You can customise them as much as you like of course. Just use your favourite make command on them, set them up however you like, and then copy them back as I did above.

The Simplest Build

Caveat Maybe?

There is a bit of a problem in that the generic and huge kernel packages will have doinst.sh files that overwrite the symlinks in /boot, which normally point to the stock kernel, system map, and config.

If you want to keep the stock kernels untouched and usable in lilo, then the build script needs so be edited to not make those symlinks. Leaving the original symlinks in place means the stock kernel is there as a fallback in lilo if it's needed.

This will fix it:

```
sed -i "/ln -sf/d" kernel-generic.SlackBuild
```

(The generic script is also used to make the huge kernel.)

Setting The Variables

After placing the relevant configs in k/kernel-configs/ and the kernel source tarball in k/, it is only necessary to set any variables we want in the environment and then run the build script. You will want to set INSTALL_PACKAGES=NO if you don't want to install the packages as they are being built.

Examples to export $OUTPUT in the environment:

In bourne type shells (sh, bash, ksh, zsh, ash etc):

```
export OUTPUT=/home/kernels/jabberwok
```

Or alternatively:

```
OUTPUT=/home/kernels/jabberwok
export OUTPUT
```

In csh:

```
setenv OUTPUT /home/kernels/jabberwok
```

If you don't want to install the packages, use one of the above methods to export INSTALL_PACKAGES as anything other than empty or YES. E.G.:

```
export INSTALL_PACKAGES=NO
```

The Build

Then run the build script:

```
sh build-all-kernels.sh
```

Depending on the recipes, at the start you will see something like:

```
************************************************
* Building kernels for recipe x86_64...
************************************************
```

(Insert appropriate knuckle biting and cheek clenching here.)

And at the end:

```
x86_64 kernel packages done!
```

If something went awry, then a build log may be useful. It may be large, probably over 100,000 lines for a single recipe, but could be useful for debugging failed builds if there isn't anything immediately obvious in the terminal:

```
sh build-all-kernels.sh 2>&1 | tee build.log
```

Viewing large build logs in 'less' ought to be reasonably faster than, say, a text editor with syntax highlighting, like vim, for example.

Not Only But Also

In addition to the packages being created, we also have the kernels, system maps, and configs copied to $KERNEL_OUTPUT_DIRECTORY which is set to:

$OUTPUT/kernels/<type>$(echo ${LOCALVERSION} | tr -d -).s

Which in this example produced for generic:

```
ls -l /home/kernels/jabberwok/kernels/generic.s/
total 6664
-rw-r--r-- 1 root root  725466 Apr 22 04:50 System.map.gz
-rw-r--r-- 1 root root 5902624 Apr 22 04:50 bzImage
-rw-r--r-- 1 root root  186961 Apr 22 04:49 config
```

$LOCALVERSION is unset before creating that directory, in case you were wondering.

After the packages are built they can be installed using installpkg or upgraded using upgradepkg. A useful command to remember which deals with anything is:

```
upgradepkg --install-new --reinstall <package>
```

After installing or upgrading the generic kernel, don't forget to recreate your initrd. Lilo must also be run after upgrading, unless using grub.

A Setup Script

An example setup.sh:

```
#!/bin/sh

# setup script for build-all-kernels.sh

# The is an example kernel package build script.

########### SETTINGS ##########

LOCALVERSION=jabberwok
CONFIG_SUFFIX=.x64
export OUTPUT=/home/kernels/$LOCALVERSION
export INSTALL_PACKAGES=NO
#export TMP=/mnt/tmpfs
#export NUMJOBS=-j1

####### END OF SETTINGS ######

set -e
```

```
# From build-all-kernels.sh.
VERSION=$(/bin/ls -1 linux-*.tar.?z | sort -V | tail -n 1 \
    | rev | cut -f 3- -d . | cut -f 1 -d - | rev)
OLD_CONFIG=$(/bin/ls -1 kernel-configs/config-generic-*${CONFIG_SUFFIX} \
    | sort -V | tail -n 1 | rev | cut -f 2- -d . | cut -f 1 -d - | rev)

# Bail out if no VERSION found.
if [ -z "$VERSION" ]; then
    printf "%s\n" "Error: \$VERSION is not set."
    exit 1
fi

GENERIC_CONFIG=config-generic-${VERSION}${CONFIG_SUFFIX}
HUGE_CONFIG=config-huge-${VERSION}${CONFIG_SUFFIX}

# copy configs if they don't exist already for the version
if ! [ -e kernel-configs/$GENERIC_CONFIG ]; then
    printf "\n%s\n" "Copying generic kernel config:"
    cp -v kernel-configs/config-generic-${OLD_CONFIG}${CONFIG_SUFFIX} \
        kernel-configs/$GENERIC_CONFIG
fi

if ! [ -e kernel-configs/$HUGE_CONFIG ]; then
    printf "\n%s\n" "Copying huge kernel config:"
    cp -v kernel-configs/config-huge-${OLD_CONFIG}${CONFIG_SUFFIX} \
        kernel-configs/$HUGE_CONFIG
fi

# Use olddefconfig instead of oldconfig so make doesn't prompt us.
sed -i "s,make oldconfig,make olddefconfig,g" *SlackBuild

# Set local version.
sed -i 's/CONFIG_LOCALVERSION=.*/CONFIG_LOCALVERSION="-'$LOCALVERSION'"/' \
    kernel-configs/$GENERIC_CONFIG kernel-configs/$HUGE_CONFIG

# Don't clobber the symlinks in /boot.
sed -i "/ln -sf/d" kernel-generic.SlackBuild

# ONLY if we use a custom $TMP.
# sed -i "s,/tmp/package-kernel-source/,$TMP/package-kernel-source/,g" \
# kernel-modules.SlackBuild

# Speed up find - optional tweak.
# sed -i "s,-exec chmod 644 {} \\\\;,-exec chmod 644 {} \\\\+," \
# kernel-source.SlackBuild

# Run the build script.
sh build-all-kernels.sh
```

In a nutshell:

- Set the variables at the top of the script.
- sh setup.sh

Tweaks in a nutshell:

- “oldconfig” is changed to “olddefconfig”.
- The local version is set in each config.
- The symlinks are removed from the kernel Slackbuild.
- The hard-coded /tmp path is replaced. Only needed if setting $TMP.

$VERSION is found automatically from the highest version source archive. The configs are found in a similar way. They will be created if none are found for $VERSION. If the versions aren't what you expect, you may need to remove the other files, leaving the ones you need.

If $VERSION is unset it will bail out with an error.

Lilo

When using a local version the image name would be formatted like:

/boot/vmlinuz-<type>-<local version>-<version>-<local version>

Without a local version, e.g. an upgrade to a stock kernel, it would be:

/boot/vmlinuz-<type>-<version>

However, if the /boot symlinks are created then only:

/boot/vmlinuz

This saves having to re-edit /etc/lilo.conf at every upgrade, although **lilo still needs running afterwards.**

Example using a local version:

```
image = /boot/vmlinuz-generic-jabberwok-5.0.8-jabberwok
  initrd = /boot/initrd-5.0.8-jabberwok.gz
  root = /dev/sda2
  label = 5.0.8-jabberwok
  read-only
```

Substitute the initrd file name for your own. I tend to name them after the specific kernel they were made for, which is helpful with more than one kernel installed.

Final Thoughts

Once a setup file is made, it is a very simple and straightforward way to build kernels in a reproducable manner. No doubt my example setup file can be improved, but it shows enough to get

started.

This is the way that the kernel packages are made in Slackware -current, albeit with a few tweaks added, so it has does have a history of reliability behind it.

Link to some example scripts:

http://tty1.uk/scripts/kernel/

Consider these as works in progress, so they may be updated from time to time.

Sources

* Originally written by dive

howtos, kernel