1.105.2 Reconfigure, build, and install a custom kernel and kernel Weight 3

Linux Professional Institute Certification — 102

Nick Urbanik nicku@nicku.org

2005 September

Outline

Topic 105 Kernel [7]

Where we are up to

1.105.1 Manage/Query kernel and kernel modules at runtime [4]

1.105.2 **Reconfigure, build, and install a custom kernel and kernel [3]**

Description of Objective

1.105.1 Reconfigure, build, and install a custom kernel and kernel

Candidates should be able to *customize*, *build*, and *install* a kernel and kernel loadable modules from source. This objective includes customizing the current kernel configuration, building a new kernel, and building kernel modules as appropriate. It also includes installing the new kernel as well as any modules, and ensuring that the boot manager can locate the new kernel and associated files (generally located under /boot, see objective 1.102.2 for more details about boot manager configuration).

Key files, terms, and utilities include:

`/usr/src/linux/*` — Where we traditionally put the kernel source (though it is better to not compile everything as the `root` user)

`/usr/src/linux/.config` — The source configuration file built/edited by `make\ {x,old,menu,,}config`

`/lib/modules/kernel-version/*` —

`/boot/*` — where the BIOS loads the kernel from

`make` — program used to build software, including the kernel

make targets: `config`, `menuconfig`, `xconfig`, `oldconfig`, `modules`, `install`, `modules_install`, `dep` — all targets you can type after `make` when building a kernel

What is the kernel?

- The kernel consists of:
  - the kernel itself:
    - such as `/boot/vmlinuz-2.6.12-1.1447_FC4smp`
  - The kernel modules:
    - In `/lib/modules/$(uname -r)`

Kernel naming conventions

- In a name such as `vmlinuz-2.6.12-1.1447_FC4smp`, there are the following parts of the name that identify the kernel:
  - major number: here 2
    - In the Makefile, called **VERSION**
  - minor number: here 6
    - In the Makefile, called **PATCHLEVEL**
  - revision: here 12
    - In the Makefile, called **SUBLEVEL**
  - vendor string: here `-1.1447_FC4smp`
    - In the Makefile, called **EXTRAVERSION**
    - Always change this in the top level makefile if you already have an existing kernel with the same name whose modules you do not want to overwrite
- The *value that you choose for these variables* in the top level *Makefile* determines what you see when you run the program **`uname -r`**
- Consequently also determines the *name of the modules directory*.

Types of Kernel Files

- The main kernel file can be produced by **`make zImage`** or by **`make bzImage`**
  - a *zImage* kernel is limited to about 508 kB in size and is loaded into lower memory
  - zImage kernels are deprecated after 2.4.0-test3-pre3
    - See `Documentation/i386/boot.txt`
  - a *bzImage* kernel can be up to about 2.5 MB in size
  - *Both* are compressed using *gzip compression*
  - the 'b' in "bzImage" means "**big**" rather than indicating bzip2 compression
  - bzImage kernels are loaded into higher memory

Other Kernel Files in `/boot`

- **`System.map`** contains the addresses of kernel symbols
  - Used by tools to interpret kernel error messages or OOPSes, to translate kernel addresses into names that mean more to us humans
  - See `http://www.dirac.org/linux/system.map/`
- The **initrd file**, which is a compressed filesystem that is mounted as a disk
  - It contains the drivers (*kernel modules*) that the kernel *needs to access the hard disk*.
  - The memory used by the initial ram disk is freed up after the modules have been loaded into the kernel
- It is nice to have the kernel **`.config`** configuration file handy so that the administrator knows how the kernel was built

Kernel Modules

- Kernel modules usually provide one of the following:

  device driver: supporting a specific kind of hardware

  file system driver: supporting the ability to read/write different file systems

  system call extensions: most system calls are supported by the base kernel, but modules can extend or add system calls

  network driver: implement particular network protocols

  executable loader: support loading and executing additional executable file formats

Documentation

- The directory `Documentation` under the top level contains lots of documentation relating to many aspects of the kernel.
- The file `Documentation/Configure.help` provides help with configuration for pre-2.6 kernels.

Outline

Getting the sources

```
$ lftp ftp://ftp.au.kernel.org/pub/linux/kernel/v2.6/ ↩
cd ok, cwd=/pub/linux/kernel/v2.6
lftp ftp.au.kernel.org:/pub/linux/kernel/v2.6> ls
-rw-rw-r--    1 ftp      ftp         12777 Dec 18  2003 ChangeLog-2.6.0
-rw-rw-r--    1 ftp      ftp        193569 Jan 09  2004 ChangeLog-2.6.1
-rw-rw-r--    1 ftp      ftp       1552868 Dec 25  2004 ChangeLog-2.6.10
-rw-rw-r--    1 ftp      ftp       1495678 Mar 03  2005 ChangeLog-2.6.11
-rw-r--r--    1 ftp      ftp          1221 Mar 09  2005 ChangeLog-2.6.11.1
...
-rw-rw-r--    1 ftp      ftp       4191691 Oct 19  2004 patch-2.6.9.gz
-rw-rw-r--    1 ftp      ftp           248 Oct 19  2004 patch-2.6.9.gz.sign
-rw-rw-r--    1 ftp      ftp           248 Oct 19  2004 patch-2.6.9.sign
drwxrwsr-x    2 ftp      ftp          8192 Dec 19  2003 pre-releases
drwxrwsr-x    4 ftp      ftp         28672 Sep 13 03:05 snapshots
drwxrwsr-x    4 ftp      ftp         24576 Sep 13 13:53 testing
lftp ftp.au.kernel.org:/pub/linux/kernel/v2.6> mget linux-2.6.13.1.tar.bz2*
 38375702 bytes transferred in 746 seconds (50.2K/s)
Total 2 files transferred
lftp ftp.au.kernel.org:/pub/linux/kernel/v2.6> bye
$ tar xvjf linux-2.6.13.1.tar.bz2 ↩
drwxr-xr-x git/git           0 2005-09-10 12:42:58 linux-2.6.13.1/
-rw-r--r-- git/git       18691 2005-09-10 12:42:58 linux-2.6.13.1/COPYING
-rw-r--r-- git/git       89317 2005-09-10 12:42:58 linux-2.6.13.1/CREDITS
drwxr-xr-x git/git           0 2005-09-10 12:42:58 linux-2.6.13.1/Documentation/
...
```

Where to untar the source?

- Many people untar the source below `/usr/src`
- . . . but I prefer to untar it in a subdirectory below my home directory
- It is better to compile the code *as a normal user* rather than as root
  - It is a good principal to do *anything* with the least privelege required
- I will call the first directory appears when we untar the code as the *top level directory*
  - For example, if I did the untarring above in the directory `~/src`, then the top level directory is `~/src/linux-2.6.13.1`

Outline

editing `.config`

- We next need to edit/create a file `.config` in the top level directory
- Could edit by hand, but easy to make a mistake
- We call `make` with one of the four *targets*:

config this is a method I have not used for years. It does not allow you to go back: you can only move forward, answering questions

menuconfig this gives you a nice text curses-based screen that allows you to navigate through the choices as you wish

xconfig on 2.4 kernels, gives a nice Tk interface, and on 2.6 kernels gives a program called `qconf`, which on my system is linked to a `qt` library.

oldconfig this allows you to easily update an existing `.config` file, answering the configuration questions only for new options which are in the new source code, but which were not covered in the old `.config` file.

Answering the questions

- For each configuration option, we may be presented with the options
  - y yes: means compile this right into the base kernel
  - m module: means compile this as an external module that can be loaded into the kernel when it is needed
    - It doesn't hurt to compile lots of modules, even though you don't need them, except that:
      - it takes more time to compile,
      - the chance of finding an error in the source code is increased, and
      - the modules directory will take more hard disk space.
  - n no: means do not compile this capability at all.

Outline

make targets

- Here are the steps to compile the base kernel image:

  `make dep`: only needed in pre 2.6 kernels, not in 2.6 kernels

  `make clean`: removes old object files; important if the source has been compiled previously

  `make bzImage`: builds the kernel image file. You will find it in the location `arch/i386/boot/bzImage`

- There are alternatives that I suggest you avoid, such as:

  `make zImage` `Documentation/i386/boot.txt` says this is deprecated after 2.4.0. For a very small kernel, loaded into low memory.

  `make zlilo` attempts to install the kernel directly using `lilo`

  `make zdisk` to create a bootable floppy.

- It is simplest (to me) to use `make bzImage` and copy the kernel file to wherever you want it.

make targets for the modules

`make modules`: builds the kernel modules. Takes a while on a slow machine, especially if you have enabled many kernel modules

`sudo make modules_install`: install the modules under `/lib/modules/`⟨*kernel-name*⟩, where ⟨*kernel-name*⟩ is determined by how you edited the variables at the top of the main Makefile

Other make targets

`make mrproper`: Clean the kernel source tree completely, to almost pristine condition. This will also delete `.config`. (`make distclean` slightly cleaner).

- Some people say the name means something highly and deeply technical (**m**aintainer proper)
- . . . but Linus says it's a cleaning fluid (German version of Mr Clean):

`http://www.alphalinux.org/archives/axp-list/1996/October1996/1237.html`

On Tue, 22 Oct 1996, Marc Singer wrote:
>
> > > What is mrproper? I've been wondering this for a long time.
> >
> > mrproper clears out all the config preferences.
>
> Yes, but what does it represent? Mr. Proper?

There was a silly cleaning cluid commercial over here in Finland a few years ago with a particularly annoying jingle. "Mr Proper" is/was the name of the cleaning fluid.

Sorry about that,

Linus

Outline

Installing into /boot

- All these files should have a name containing the version that you set in the Makefile
  - In the following, replace $VERSION by the value of VERSION in the Makefile, $PATCHLEVEL by the value of PATCHLEVEL in the Makefile, ...
- Manually copy it from the file arch/i386/boot/bzImage (relative to the top level of the source tree) to /boot
  - Copy it to the name /boot/vmlinuz-$VERSION.$PATCHLEVEL.$SUBLEVEL$EXTRAVERSION
- Copy the System.map file into /boot
  - Call it /boot/System.map-$VERSION.$PATCHLEVEL.$SUBLEVEL$EXTRAVERSION
- Copy .config to /boot as /boot/config-$VERSION.$PATCHLEVEL.$SUBLEVEL$EXTRAVERSION

Outline

Make the initial RAM disk file

- If you did not compile all the modules that your kernel needs to access the hard disk right into the kernel (not as modules), then you need an initial ram disk file
- Let us represent the value of the kernel version — `$VERSION.$PATCHLEVEL.$SUBLEVEL$EXTRAVERSION` as ⟨*kernel version*⟩.
- On Red Hat systems you create this with a command like this:

```
$ sudo mkinitrd -v /boot/initrd-⟨kernel version⟩.img ⟨kernel version⟩
$ mkinitrd --help
usage: mkinitrd [--version] [-v] [-f] [--preload <module>]
       [--omit-scsi-modules] [--omit-raid-modules] [--omit-lvm-modules]
       [--with=<module>] [--image-version] [--fstab=<fstab>] [--nocompress]
       [--builtin=<module>] [--nopivot] <initrd-image> <kernel-version>

       (ex: mkinitrd /boot/initrd-2.2.5-15.img 2.2.5-15)
```

Outline

Having Grub start the kernel

- edit GRUB's configuration file `/boot/grub/menu.lst` or `/boot/grub/grub.conf`
- Add a new section for your kernel, telling GRUB about the initrd file if you need one:

```
title Latest kernel (2.6.13.2)
        root (hd0,0)
        kernel /boot/vmlinuz-2.6.13.2 ro root=/dev/hda1
        initrd /boot/initrd-2.6.13.2.img
```

Test it

- Do not remove your old kernel from `/boot/grub/menu.lst` before you have tested your new kernel
- Boot the new kernel on a test system and give it a good try out before you install it on your production systems

Building an external module

- You may need to compile a module to support special hardware,
  - for example: a WinModem, or the LabJack data acquisition system
- You get the source code for the module; you don't want to have to install all the source code for your kernel.
- Much easier with 2.6 kernels than with 2.4

Outline

Building an external module on Fedora/Red Hat

- Install the appropriate `kernel-⟨type-⟩devel` software package; for example, do
  - `$` **`yum -y install kernel-devel`** ↩ for an ordinary kernel, `$` **`yum -y install kernel-smp-devel`** ↩ for a multiprocessor kernel.
- In the directory where you have the source code `foo.c` for the module `foo.ko`, you need a `Makefile` containing

```
obj-m := foo.o

KDIR  := /lib/modules/$(shell uname -r)/build
PWD   := $(shell pwd)

default:
    $(MAKE) -C $(KDIR) M=$(PWD) modules
```

- Then type `$` **`make`** ↩ where your `foo.c` is
- Then load it with `$` **`sudo insmod foo.ko`** ↩

Resources for Building Modules

- Excellent instructions are provided in the Release Notes
- See also `Documentation/kbuild/modules.txt` in the kernel source code.

Outline

Building External Modules on Debian/Ubuntu

- install the package `linux-headers-$(uname -r)`:

```
$ sudo apt-get update ↩
$ sudo apt-get install linux-headers-$(uname -r) ↩
```

- Then follow the instructions in `Documentation/kbuild/modules.txt`
- Note:
  - I was not successful with Breezy and `labjack.ko`
  - any suggestions welcome.

Topics Covered

Topics Covered

Topics Covered

Topics Covered

Topics Covered

Topics Covered

Topics Covered

Topics Covered

Topics Covered

Topics Covered

Topics Covered

Topics Covered

Topics Covered

Topics Covered

Topics Covered

License covering this document