

Hacking the Linux 2.6 kernel, Part 1: Getting ready

Materials you'll need, and tips for configuring and compiling the kernel

Skill Level: Introductory

[Lina Mårtensson \(linam@tyst.nu\)](mailto:linam@tyst.nu)
Freelance writer

[Valerie Henson \(val@nmt.edu\)](mailto:val@nmt.edu)
Software Engineer
IBM

20 Jul 2005

In this first of a two-part series, learn about system and environment requirements, the best ways to acquire Linux source code, how to configure and boot your new kernel, and how to use the `printk` function to print messages during bootup.

Section 1. Before you start

Learn what these tutorials can teach you, and what you need to run the examples in them.

About this series

The capability of being modified is perhaps one of Linux's greatest strengths, and anyone who has dabbled with the source code has at least stood at the gates of the kingdom, if not opened them up and walked inside.

These two tutorials are intended to get you started. They are for anyone who knows a little bit of programming and who wants to contribute to the development of Linux,

who feels that something is missing in the kernel and wants to fix that, or who just wants to find out how a real operating system works.

About this tutorial

This tutorial takes a basic approach to kernel hacking, and starts by explaining what materials are needed and offering tips for configuring and compiling the kernel. It lays the groundwork for [Hacking the Linux 2.6 kernel, Part 2: Making your first hack](#), which presents an overview of the kernel source, provides an understanding of system calls, teaches you how to make your own modules, and, finally, how to create, apply, and submit patches.

Prerequisites

To run the examples in this tutorial, you need a Linux box, root access on the Linux box (or a sympathetic admin), the ability to reboot the box several times a day, an installed compilation environment, and a way to get the kernel source.

Get more advice on choosing the correct materials for your needs in the next section, [Requirement details](#).

Section 2. Requirement details

This section covers

What you'll need

- A Linux box
- Root access on your Linux box (or the sympathetic ear of an admin)
- The ability to reboot the box several times a day
- An installed compilation environment
- Some way to get the kernel source

System requirements and tips

Your Linux box needs to have at least 300 MB of free disk space. Find out how much free disk space you have by using the command `df -h`.

Remember that you can compile on one Linux box, and run the kernel you compile on another Linux box. You should compile on the fastest Linux box possible. A 500 MHz Pentium III with 256 MB RAM will take about 40 minutes to compile a Linux kernel; a 1200 MHz Pentium III with 640 MB RAM will take about 10 minutes; a Duron 800 Mhz with 256 MB RAM will take around 5 minutes.

Root access

You don't actually *need* root access as long as you have someone available with root access to set you up.

What you *do* need is the ability to boot your new kernel -- this may include the ability to:

- Run LILO
- Copy kernels to places outside your home directory
- Reboot the machine

You can compile in your normal user directory, and make the directories you need to write to be owned by your user. You can also give the user reboot a password so you can reboot the machine without `su`-ing to root. Talk to your sysadmin to find out what you need to do.

Rebooting and safety

Ideally, your Linux box is something only you need access to. If other people need it during the day but not at night, you might try using it then.

If you are concerned about making the box unusable, there are many ways to test a new kernel and then reboot back into your old kernel.

Disk corruption is not as common as you might think, but if you do not want to risk the data on your hard disk, you can use a ramdisk with your new kernel. You can also create a new partition on your hard disk and use that as your root file system

when you are experimenting with new kernels.

Compilation environment

Hopefully, most people installed their Linux box with gcc and other compilation tools included. You can test to see if you have a compilation environment installed by copying the following into a file named `hello.c`:

```
#include <stdio.h>

int main (void) {
    printf ("Hello, world!\n");
    return 0;
}
```

Then, compile and run it with:

```
$ gcc hello.c -o hello
$ ./hello
```

If this doesn't work, you need to install a compilation environment. The easiest way to do this might just be to re-install your machine.

Getting the kernel source

The better methods of getting the kernel source involve having a reasonably high bandwidth Internet connection. Here are the suggested ways, starting with the most preferred way:

- Cogito
- FTP
- Distribution CD

Choose one of these options, and remember that while it's easiest to just install the source from your distribution CD, you'll also want to stay up to date with the latest version of the Linux source if you want to participate in kernel development (which we recommend).

There are various other ways to get the kernel source that are not covered here, involving CVS or rsync.

It is better not to have a `/usr/src/linux` symbolic link to your kernel source tree. However, the `/usr/src/linux` link will sometimes be used as a default by some modules that compile outside the kernel. If you need it, make sure it points to the correct kernel tree. If it doesn't, modules that compile outside the kernel may use the wrong header files.

Cogito

Cogito, a new source code management tool created by Linus Torvalds and maintained by kernel hackers, can be used to clone a Linux source repository. This option requires 500 MB of free disk space. See [Resources](#) for a link to the Cogito Web site.

Follow the instructions in the README file to install Cogito. Then, clone the main Linux tree using Cogito, choosing as necessary the directory you would like to work in, such as:

```
$ cd /usr/src
```

```
$ cg-clone rsync://rsync.kernel.org/pub/scm/linux/kernel/git/torvalds/linux-2.6.git
```

This will create a new directory and clone the Linux tree into it.

And now you have the kernel source! See [Resources](#) for links to more information on how to use Cogito.

FTP

You can FTP the kernel source from `ftp.kernel.org`. This is a bandwidth-intensive operation, so don't use it if you have a slow link.

You should FTP to your local kernel mirror, which is named `ftp.<country code>.kernel.org`. For example, if you live in the US, the command is `ftp.us.kernel.org`.

Log in (if necessary) with username `ftp` or `anonymous`. Change the directory to `pub/linux/kernel/v2.6` and download the latest kernel. For example, if the latest version is 2.6.9, download the file named `linux-2.6.9.tar.gz`.

Usually there is a file named `LATEST-IS-<version>` to tell you what the latest version is. If your bandwidth is low, you can choose to download the bzip2 format (file name ending in `.bz2`) instead of the gzipped format (file name ending in `.gz`), but bzip2 will take a long time to uncompress.

Untar and uncompress the file in the directory where you are planning to work.

You now have your Linux kernel source.

Install source from a distribution CD

If you already have a directory named `/usr/src/linux` and it contains more than one directory, you already have the source installed. If you don't, read on.

Mount your installation CD ROM. On a Red Hat-based system, the source RPM is usually in `<DistributionName>/RPMS/` and is named `kernel-source-<version>.<arch>.rpm`.

One way to find the kernel source package is to run the command `find /mnt/cdrom -name *kernel-*` (assuming your CD is mounted at `/mnt/cdrom`).

Install the RPM using `rpm -iv <pathname>/kernel-source-<version>.<arch>.rpm`. The `v` switch will tell you if it fails or not.

There are many systems that are not RPM-based and they will have other ways of installing the kernel source from the distribution CD. There are too many options to cover them all here.

Which source code source is best?

Generally, we recommend Cogito over FTP, and FTP over a vendor source.

Cogito handles creating patches for you. (A patch contains the differences between one source tree and another source tree.) Cogito also applies the latest changes for you. When you want to update your tree you just type `cg-update` and, most of the time, it works. The only time you will have to do work is if you have written code that

conflicts with the new code downloaded from the parent tree.

When you want to make a patch in Cogito to send to somebody, just type `cg-diff > patchname`. When you want to undo your latest changes, you type `cg-cancel`.

The vanilla kernel source from `ftp.kernel.org` is usually preferred over the vendor source, because you never know what changes the vendor has made to the source. Most of the time, the vendor has improved the tree, but often they make changes of dubious value to a kernel hacker. For example, Red Hat 6.2 shipped a kernel that compiled differently depending on whether you were running an SMP kernel at the time of compilation. This makes sense if you are just recompiling a kernel for that machine, but it was useless if you were trying to compile a kernel for a different machine.

The other reason to use vanilla source instead of vendor source is if you want to create and send patches to other kernel developers. If you create a patch and send it to the Linux kernel mailing list for inclusion in the main kernel tree, it had better be against the latest vanilla source tree. If you create a patch on a vendor source tree, it's unlikely to apply to a vanilla source tree.

The one place where vendor source is crucial is for non-x86 architectures. The vanilla source almost never builds and boots on an architecture other than x86. Each architecture usually has some quirky way of getting the latest source in addition to the vendor-supplied source.

Section 3. Configuring your kernel

Configuration tools

The Linux kernel comes with several configuration tools. Run each one by typing `make <something>config` in the top-level kernel source directory. (Run all `make` commands from the top-level kernel directory.)

make config is the barebones configuration tool. It asks each and every configuration question in order. The Linux kernel has a *lot* of configuration questions. Don't use it unless you are a masochist.

make oldconfig takes the config file named `.config` and only asks the configuration questions that are not already answered in that file. It's most useful for

upgrading to a more recent version of the kernel.

make menuconfig is the kernel hacker mainstay. This command pops up a text-based menu-style configurator using the ncurses library. You can descend into the menus that interest you and change only the configuration options you care about.

make xconfig, make gconfig: If you're running XWindows, these are prettier, clickable versions of menuconfig. They are more sensitive to buggy configuration input files than menuconfig and thus tend to work less often than menuconfig (more people use menuconfig). xconfig uses the qt libraries; gconfig uses the gtk libraries.

Files

Each of the configuration programs produces these end products:

- A file named `.config` in the top-level directory containing all your configuration choices
- A file named `autoconf.h` in the `include/linux/` directory defining (or not defining) the `CONFIG_*` symbols so that the C preprocessor knows whether or not they are turned on

If you have a working `.config` file for your machine, just copy it into your kernel source directory and run `make oldconfig`. You should double check the configuration with `make menuconfig`.

If you don't already have a `.config` file, you can create one by visiting each submenu and turning on or off the options you need. menuconfig and xconfig have a "Help" option that shows you the `Configure.help` entry for that option, which may help you decide whether or not it should be turned on.

Red Hat Linux and other distributions often include sample `.config` files with their distribution-specific kernel source in a subdirectory named `configs` in the top-level source directory.

If you are compiling for PowerPC, you have a variety of default configs to choose from in `arch/ppc/configs`. `make defconfig` will copy the default ppc config file to `.config`.

Tips for configuring a kernel

Keep these tips in mind when configuring a kernel:

- Always turn on "Prompt for development... drivers."
- From kernel 2.6.8 on, you can add your own string (such as your initials) at "Local version - append to kernel release" to personalize your kernel version string (for older kernel versions, you have to edit the `EXTRAVERSION` line in the `Makefile`).
- Always turn off module versioning, but always turn on kernel module loader, kernel modules, and module unloading.
- In the kernel hacking section, turn on all options except "Use 4Kb for kernel stacks instead of 8Kb." If you have a slow machine, don't turn on "Debug memory allocations" either.
- Turn off features you don't need.
- Use only the modules that you have a good reason to use. For example, if you are working on a driver, you'll want to load a new version of it without rebooting.
- Be extra sure you selected the right processor type. Find out what processor you have now with `cat /proc/cpuinfo`.
- Find out what PCI devices you have installed with `lspci -v`.
- Find out what your current kernel has compiled in with `dmesg | less`. Most device drivers print out messages when they detect a device.

If all else fails, copy a `.config` file from someone else.

Section 4. Compiling your kernel

Compile the kernel

At this point, you have your kernel source, and you've run one of the configuration programs and (very important) saved your new configuration file. Don't laugh -- this has been forgotten many times before.

First, build the kernel:

On x86:

```
# make -j<number of jobs> bzImage
```

On PowerPC:

```
# make -j<number of jobs> zImage
```

Where `<number of jobs>` is two times the number of CPUs in your system. So, if you have a single-CPU Pentium III, you'd do this:

```
# make -j2 bzImage
```

The `-j` argument tells the `make` program how many jobs (commands in the `Makefile`) to run at once. `make` knows which jobs can be run at the same time and which jobs need to wait for other jobs to finish before they can run. Kernel compilation jobs spend enough time waiting for I/O (for example, reading the source file from disk) that running two of the jobs per processor results in the shortest compilation time. **Note:** If you get a compile error, run `make` with only one job so that it's easy to see where the error occurred. Otherwise, the error message will be hidden in the output from the other `make` jobs.

If you have enabled modules, you'll need to compile them with the command:

```
# make modules
```

If you are planning on loading these modules on the same machine they are compiled on, then run the automatic module install command. But *first*. Save your old modules!

```
# mv /lib/modules/`uname -r` /lib/modules/`uname -r`.bak  
# make modules_install
```

This will put all the modules in subdirectories of the `/lib/modules/<version>` directory. You can find out what `<version>` is by looking in

```
include/linux/version.h.
```

Recompile the kernel

So, you've compiled the kernel once. Now you want to change part of the kernel and recompile it. What do you need to do?

In most cases, simply running `make -j2 bzImage` (or whatever your kernel compile command is) again will do the trick. If you've altered a module's source file, then do `make modules` and `make modules_install` (if appropriate).

Sometimes, you'll change things so much that `make` can't figure out how to recompile the files correctly. `make clean` will remove all the object and kernel object files (ending in `.o` and `.ko`) and a few other things. `make mrproper` will do everything `make clean` does, plus remove your config file, the dependency files, and everything else that `make config` creates. Be sure to save your config file in another file before running `make mrproper`. Afterwards, copy the config file back to `.config` and start over, beginning at `make menuconfig`. A `make mrproper` will often fix strange kernel crashes and compilation errors that make no sense.

Here are some tips for recompilation when you are working on just one or two files. Say you are changing the file `drivers/char/foo.c` and you are having trouble getting it to compile at all. You can just type (from the `drivers/char/` directory) `make -C <top directory of your kernel source> SUBDIRS=$PWD modules`, and `make` will immediately attempt to compile the modules in that directory instead of descending through all the subdirectories, in order, looking for files that need recompilation. If `drivers/char/foo.c` is a module, you can then `insmod` the `drivers/char/foo.ko` file when it actually does compile, without going through the full-blown `make modules` command. If `drivers/char/foo.c` is not a module, you'll then have to run `make -j2 bzImage` to link it with all the other parts of the kernel.

The `&&` construct in bash is very useful for kernel compilation. The bash command line:

```
# thing1 && thing2
```

says, "do `thing1`, and if it doesn't return an error, do `thing2`." This is useful for doing something on the condition that a compile command succeeds.

When working on a module, the following command is useful:

```
# rmmod foo.ko && make -C <top directory of your kernel source> \  
SUBDIRS=$PWD modules && insmod foo.ko
```

This says, "Remove the module `foo.ko`, and if that succeeds, compile the modules in `drivers/char/`, and if that succeeds, load the kernel module object file." That way, if you get a compile error, it doesn't load the old module file, and it's obvious that you need to fix your code.

Don't let yourself get too frustrated! If you just can't figure out what's going on, try the `make mrproper` command.

Section 5. Booting your new kernel

Before you boot the kernel

Now that you've successfully compiled your new kernel, you need to boot it. We're going to cover booting a new kernel on three different architectures: x86, PowerPC, and Alpha. Some similarities exist among all three, so we'll start with the similarities.

Rule number one: *Never, ever delete your current working kernel or bootloader configuration files!* Never copy your new kernel over your original kernel. You don't need to keep every kernel you ever compile, but do pick one or more "safe" kernels and keep them and their configuration files intact.

Copy your new kernel to a safe location

First, copy the new kernel you just compiled to a safe location. We suggest using `/boot/mynewkernel` as the destination. If you're on x86, you want to copy the `bzImage`:

```
# cp arch/i386/boot/bzImage /boot/mynewkernel
```

If you're on a PowerPC or Alpha, copy the `vmlinux`:

```
# cp vmlinux /boot/mynewkernel
```

Booting your new kernel with LILO on an x86

LILO is the most common bootloader for x86 machines. You are running LILO if you see "LILO:" when you boot. We'll just go over the bare minimum of information you need to boot a new kernel in LILO. (For more information on LILO, type `man lilo` and `man lilo.conf`).

You will need to be root to edit `/etc/lilo.conf` and run LILO.

The LILO configuration file is in `/etc/lilo.conf`. The basic idea here is to copy the chunk of your configuration file for your current kernel and then change the name of the kernel file to boot. Remember, you are copying the information for your current kernel and changing the copy, *not* editing the original.

Say you have something like this in your `/etc/lilo.conf`:

```
image=/boot/vmlinuz-2.2.14-5.0
label=linux
initrd=/boot/initrd-2.2.14-5.0.img
read-only
root=/dev/sda1
```

- The `image` field tells LILO where to find the file containing your new kernel.
- The `label` field is what you type at the LILO prompt to boot that kernel. It's a good idea to make the label something short and easy to type.
- The `initrd` field is optional and specifies an initial ramdisk to load before mounting the root file system.
- The `read-only` field says to initially mount the root file system read-only rather than read-write.
- The `root` field tells LILO which device contains the root file system (your root file system is what is mounted at `/`).

Copy this information and change the following fields:

```
image=<file name of your new kernel>
label=<choose whatever name you'd like here>
```

Remove the `initrd` field if it exists:

```
initrd=/boot/initrd-2.2.14-5.0.img
```

The reason you should remove the `initrd` field is because the `initrd` file does not contain anything that would be useful to your new kernel. In a normal Linux distribution, the `initrd` contains a lot of kernel modules which can only be loaded by the distribution kernel.

The new lines you just added to your configuration file should look something like this when you're done:

```
image=/boot/mynewkernel
label=new
read-only
root=/dev/sda1
```

LILO does not know that you changed anything in `/etc/lilo.conf` until you run the `lilo` command. You will probably do this more than once: compile a new kernel, copy it to the new place, forget to run LILO, and reboot, only to find that LILO doesn't know about the new kernel. So, after you change the `/etc/lilo.conf` file, always run LILO:

```
# lilo -v
```

The `-v` says to be verbose about what it's doing. LILO will complain if it can't find the files you told it to use.

Now, reboot, and you'll see the LILO prompt. Type the label you chose (the label you entered in the `label` field) for your new kernel. LILO will then try to boot that new kernel. To avoid typing a command line into LILO, run this just before you reboot:

```
# lilo -v -R "<LILO command line>"
```

This tells LILO to use that command line the next time you reboot. It then erases that command line and goes back to normal behavior the next time you reboot. This is really useful for testing new kernels since you can boot your new kernel, see it crash, reset the machine, and automatically boot your working kernel without ever typing anything.

If you are having trouble with LILO, see [Resources](#) for more help.

Booting your new kernel with GRUB on an x86

GRUB is becoming a common bootloader for x86 machines. You are running GRUB if you see a pretty graphical menu which lets you select which image to boot with the arrow keys when you boot. We'll just go over the bare minimum of information you need to boot a new kernel in GRUB. For more information on GRUB, type `info grub`. You will need to be root to edit `/etc/grub.conf`.

The GRUB configuration file is in `/etc/grub.conf`. The basic idea here is to copy the chunk of your configuration file for your current kernel and then change the name of the kernel file to boot. Remember, you are copying the information for your current kernel and changing the copy, *not* editing the original. Say you have something like this in your `/etc/grub.conf`:

```
title Linux (2.4.9-31)
  root (hd0,0)
  kernel /vmlinuz-2.4.9-31 ro root=/dev/hda3
  initrd /initrd-2.4.9-31.img
```

- The `title` field is the title in the GRUB menu that corresponds to your new kernel.
- The `root` field tells GRUB where your root file system is (your root file system is what is mounted at `/`).
- The `kernel` field tells GRUB where to find the file containing your kernel and, after the kernel file name, it gives some options to pass the kernel when it boots. **Note:** The file name of the kernel is given relative to the boot partition, which usually means that you have to remove the `/boot` part from the file name.
- The `initrd` field is optional and specifies an initial ramdisk to load before mounting the root file system.

Copy this information and change the following fields:

```
title <choose whatever name you'd like here>
kernel <file name of your new kernel> ro root=/dev/hda3
```

Remember, you should almost certainly remove the `/boot` part from the file name of the kernel unless you know what you're doing.

Remove the `initrd` field if it exists:

```
initrd /initrd-2.4.9-31.img
```

The reason you should remove the `initrd` field is because the `initrd` file does not contain anything that would be useful to your new kernel. In a normal Linux distribution, the `initrd` contains a lot of kernel modules which can only be loaded by the distribution kernel.

The new lines you just added to your configuration file should look something like this when you're done:

```
title new
root (hd0,0)
kernel /mynewkernel ro root=/dev/hda3
```

Note again, that in this case our new kernel is in `/boot/mynewkernel`, but since we have a separate partition mounted on `/boot` and since GRUB will only look inside that partition when we boot, we need to take off the mountpoint part of the name, which is `/boot`. If we had everything, including `/boot`, on one big partition, then we would tell GRUB that our kernel was named `/boot/mynewkernel`.

Now, reboot, and you'll see the GRUB menu. Select the title you chose earlier with the arrow keys and hit Enter. GRUB will now try to boot your new kernel. For more information on GRUB, see the [Resources](#) section.

Booting your new kernel with yaboot and ybin on a PowerPC

Once again, we'll only cover the bare minimum to boot a new kernel under yaboot. For more information on yaboot and ybin, try `man yaboot`, `man yaboot.conf`, and `man ybin`. You will need to be root to edit `/etc/yaboot.conf` and run yaboot and ybin.

The yaboot configuration file is in `/etc/yaboot.conf` and is quite similar to the LILO configuration file. The basic idea here is to copy the chunk of your configuration file for your current kernel and then change the name of the kernel file to boot. Remember, you are copying the information for your current kernel and changing the copy, *not* editing the original. Say you have something like this in your `/etc/yaboot.conf`:

```
image=/boot/vmlinux-2.2.15-2.7.0
label=linux
root=/dev/hda10
novideo
```

- The `image` field tells yaboot where to find the file containing your new kernel.
- The `label` field is what you type at the yaboot prompt to boot that kernel. It's a good idea to make the label something short and easy to type.
- The `root` field tells yaboot which device contains the root file system.
- The `novideo` field works around a bug on some PowerMacs and may or may not be necessary on your system.

Just copy the yaboot entry that works for your particular machine. After you copy this information, change the following fields:

```
image=<file name of your new kernel>
label=<choose whatever name you'd like here>
```

The new lines you just added to your configuration file should look something like this when you're done:

```
image=/boot/mynewkernel
label=new
root=/dev/hda10
novideo
```

Next, run `ybin` if you have it. `Ybin` is a helper program that takes care of copying all the necessary files to the bootable partition, running yaboot, and all the little details that Apple's Open Firmware requires for booting a kernel. If you don't have `ybin`, consult the yaboot man page, `man yaboot`, for information about what you need to

do.

Troubleshooting booting

At this point, if your kernel doesn't boot, reboot into your old kernel (this is why you never overwrite your old kernel or configuration) and try to figure out what's gone wrong.

A good way to troubleshoot is to start out by copying the lines for your current kernel from your bootloader's configuration file and only changing the label or title of the entry. See if you can boot that kernel (and `initrd`, if it has one). Once you're sure that's working, then try substituting the name of your new kernel (and removing the `initrd` entry, if there is one). At this point, you can be fairly sure that it's your kernel that doesn't boot rather than a problem in your bootloader configuration.

Section 6. Writing kernel messages

Your first printk

Here's where you prove that kernel hacking is not magic. At this point, you've compiled and booted your new Linux kernel. Time for your first kernel hack. You're going to add a `printk`, a function that prints out a message during bootup.

First, bring up the file `init/calibrate.c` in your favorite editor. If you are using source control, be sure to check out the file first. In Cogito, use the command `cg-update`. In CVS, use `cvs co init/calibrate.c`. Find the `calibrate_delay(void)` function in the file.

The first lines are declarations of a few variables and will look something like this:

```
unsigned long ticks, loopbit;
int lps_precision = LPS_PREC;
```

Now, type in the following line just below them:

```
printk("*** I am a kernel hacker! ***\n");
```

Now recompile using `make bzImage` or `make zImage`, run LILO or `ybin` or whatever you need, and boot your new kernel. You don't need to run `make clean`. Watch the screen when you boot up -- you should see your new message. If it scrolls off the screen too quickly for you to read it, login and type `dmesg | less`. You should see your message near the beginning of the output.

Congratulations! You have just hacked the kernel. `printk`'s are one of the main tools of the kernel hacker. You will need to use them constantly.

More printk tricks

`printk` is in many ways the kernel equivalent of the C standard function `printf`. The formats are mostly the same. You will use certain formats far more often in kernel code than you do in user code. The `%x` or `%p` formats are especially useful.

`%x` says to print out the value in hexadecimal, which is base 16 and is usually far more useful information than the value in decimal, base 10. This is because the logical unit of information in a computer is a byte (8 bits), and a byte fits into two digits in a hexadecimal representation. So, the hexadecimal value `0x12345678` represents these four bytes (ignoring endian-ness):

```
0x12
0x34
0x56
0x78
```

As you can see, it's easy to separate out a hexadecimal value into individual byte values -- just read it two digits at a time. The "0x" part just says that the following numbers are in base 16, not base 10. (Note: Only obsolete and irrelevant machines have bytes which are not 8 bits. We don't care about bytes which are not 8 bits long.)

The `%p` format says to print out the value as a pointer or an address in memory. This format will depend on the machine, but is always in hexadecimal. It prints leading 0s, too.

Printing out the address of a variable

Your next assignment is to print out the address of a variable. Below your first `printk`, add this `printk`:

```
printk("The address of loops_per_jiffy is %p\n", &loops_per_jiffy);
```

Recompile and boot. Now you know at what virtual address in memory the variable `loops_per_jiffy` is stored. You can also find this out from the file `System.map` in your top-level Linux source directory:

```
$ grep loops_per_jiffy System.map
```

Loglevels

You can specify each `printk` to have a certain level of importance. Depending on the current *loglevel*, some messages will be printed to the console (your video monitor, usually) and some won't. Add this below your other `printks`:

```
printk(KERN_DEBUG "**** This is a debug message only. ****\n");
```

If your loglevel is configured normally, you won't see this message printed out during bootup. Once you've finished booting, log in and look at the kernel messages again: `dmesg | less`. Below your other messages, you should see `**** This is a debug message only. ****`.

This is convenient, since you only have to look at the output when you want to instead of having it printed on the screen (possibly while you're typing). The definition of all the different `KERN_*` loglevels are in `include/linux/kernel.h`. Usually, you will only need `KERN_INFO` and `KERN_DEBUG`.

Whenever you want to find out what's going on inside the kernel, a good way to start is by putting `printks` in strategic places. Be careful that you don't print out too much information -- it can slow down your kernel to the point of unusability. In certain places, adding a `printk` can cause a crash, cause a bug, or fix a bug. To find out exactly what `printk` does, start reading in the file `kernel/printk.c`.

Have fun with your `printk`'s!

Section 7. Summary

So far, you've learned about different ways to get the kernel source, how to configure your own kernel, and how to boot it using a variety of bootloaders. You've proved that kernel hacking is not magic by printing out messages during bootup.

In the second part of this tutorial on hacking the Linux kernel, you'll get an overview of the kernel source, learn more about system calls, and learn how to write your own kernel module.

Resources

Learn

- Part 2 of this series, "[Hacking the Linux 2.6 kernel, Part 2: Making your first hack](#)," (developerWorks, August 2005) reveals the organization of the kernel source, builds your understanding of system calls, and shows you how to craft your own kernel modules and patches.
- To learn more about how to use Cogito, take a look at the [README](#) file.
- The [LILO mini-HOWTO](#) and the [GRUB manual](#) are available online.
- "[Inside the Linux kernel debugger](#)" (developerWorks, June 2003) details KDB, the built-in kernel debugger in Linux, which allows you to trace the kernel execution and examine its memory and data structures.
- "[Magic sys request](#)" (developerWorks, April 2000) shows you how to recover from kernel meltdown.
- Find more resources for Linux developers in the [developerWorks Linux zone](#).

Get products and technologies

- Get [Cogito](#) and the Linux kernel source at [The Linux Kernel Archives](#).
- [Order the no-charge SEK for Linux](#), a two-DVD set containing the latest IBM trial software for Linux from DB2, Lotus, Rational, Tivoli, and WebSphere.
- Build your next development project on Linux with [IBM trial software](#), available for download directly from developerWorks.

Discuss

- [KernelNewbies.org](#) has lots of resources for people who are new to hacking the kernel: an FAQ, an IRC channel, a mailing list, and a wiki.
 - [KernelTrap](#) is a Web community devoted to sharing the latest in kernel development news.
 - At [Kernel Traffic](#) you can find a newsletter that covers some of the discussion on the Linux kernel mailing list.
 - Get involved in the developerWorks community by participating in [developerWorks blogs](#).
-

About the authors

Lina Mårtensson

Lina Mårtensson is pursuing a M.Sc. in Computer Science and Engineering at Chalmers University of Technology, Sweden. Contact Lina at linam@tyst.nu.

Valerie Henson

Val Henson works for the Linux Technology Center at IBM. She has more than five years experience working on the Linux and Solaris operating systems, including a year as a maintainer of part of the PowerPC Linux kernel tree. Contact Val at val@nmt.edu.