CrashCourse

Published on *Crash Course* (http://crashcourse.ca)

Home > INTERMISSION: Let's talk about header files ... (FREE LESSON)

INTERMISSION: Let's talk about header files ... (FREE LESSON)

By *rpjday*
Created *2010-06-30 06:07*

More free stuff? Cool.

As with the previous "lesson," this isn't an official course lesson but it's something you might find useful so I'll throw it in as bonus content, and we'll pick up with subscriber-only courseware in the next installment. Don't thank me -- it's what I live for.

So what's to know about header files? I'm glad you asked.

Anyone who has even a smattering of C language development knows that, in your typical C program, you'll include a number of header files that contain, say, macros you're about to use or declarations of routines you're about to call. These header files represent what you use in *user space* programming and normally match the functions that you're going to invoke from the standard C library. These header files normally reside under the `/usr/include/` directory and, on my current Ubuntu 10.04 system, are part of the `libc6-dev` package. None of this should come as a major revelation -- it's just a standard part of C language programming.

When you get into Linux kernel programming, however, you quickly learn that there is a whole new set of header files you're going to be using. Because the kernel code and modules you write are going to be running exclusively in kernel space, you will have absolutely no use for all that user space stuff like the standard C library and the associated header files.

Instead, your kernel code is going to work exclusively with the header files you can find in your kernel source tree, many of which can be found in the top-level `include/` directory you can see. (I'm assuming that you have a kernel source tree to peruse; if you don't, it's your job to run out and get one. Course students should already have a git clone of the current development tree, and I explain how to get one back in Lesson 1.)

So, to recap, if you're programming in user space, you have all of the standard C library and header files at your disposal. If you're writing kernel code, you are working exclusively with the header files that come with the kernel source tree, and there is no mixing of the two. It sounds simple enough. But wait. As always, there's more.

Where else can you get those kernel header files?

As I've mentioned earlier in this course, if you're going to be writing loadable modules or any other code that runs in kernel space, you absolutely *must* have the kernel header files against which to compile your code. In a simple case, you could have a full kernel source tree -- that would certainly work.

On the other hand, you don't actually need the entire tree -- all you really need is that portion of it that contains the header files and a few other things, and most distributions provide a package that gives you exactly that. Under Ubuntu, this would be a package with a name resembling `linux-headers-2.6.32.22.23-generic` (or whatever matches your current kernel):

```
$ apt-cache show linux-headers-2.6.32-22-generic
... snip ...
Description: Linux kernel headers for version 2.6.32 on x86/x86_64
 This package provides kernel header files for version 2.6.32 on
 x86/x86_64.
 .
 This is for sites that want the latest kernel headers.  Please read
 /usr/share/doc/linux-headers-2.6.32-22/debian.README.gz for details.
```

You can see where these kernel space header files would be installed with:

```
$ dpkg -L linux-headers-2.6.32-22-generic
/.
/usr
/usr/src
/usr/src/linux-headers-2.6.32-22-generic
/usr/src/linux-headers-2.6.32-22-generic/.config
/usr/src/linux-headers-2.6.32-22-generic/scripts
/usr/src/linux-headers-2.6.32-22-generic/scripts/basic
... snip ...
```

In short, if you want to do kernel programming, there is a package corresponding to each running kernel that you can install that provides the *kernel space* header files against which you can compile your loadable modules so that you don't even need a full kernel source tree. So far, so good. But, as always, there's more to the story.

Header files for both spaces

There are times when you're programming for user space but you need header files that define kernel space structures since you're going to be defining a structure that you want to pass into kernel space, almost certainly via a system call, and you need to get a declaration for that structure *somewhere*, which leads us to introduce a third type of header file -- the kind that are relevant for *both* kernel and user space.

Such header files are carefully selected from the header files in the kernel source tree, they're "cleaned" (using a process that will be explained shortly), and they're bundled into yet another package that you'll see in a minute. So ... where do these header files come from? At the top of your kernel source tree, simply run:

```
$ make distclean       [optional]
$ make headers_install
```

at which point a carefully selected subset of the kernel header files scattered around the tree are collected, sanitized and placed carefully under the kernel source tree directory `usr/include/`, where you can examine them with:

```
$ find usr/include | less
usr/include
usr/include/linux
usr/include/linux/virtio_9p.h
usr/include/linux/in_route.h
usr/include/linux/auxvec.h
usr/include/linux/sockios.h
usr/include/linux/joystick.h
usr/include/linux/netfilter_bridge
usr/include/linux/netfilter_bridge/ebt_802_3.h
... etc etc ...
```

What you're looking at in the output above is the collection of *kernel* header files that are also deemed to be appropriate for *user* space programmers who want to, perhaps, define structures that they will be passing to kernel code. More to the point, these header files have already been packaged for you and are almost certainly already on your system. In the case of Ubuntu 10.04, this would be the `linux-libc-dev` package:

```
$ dpkg -L linux-libc-dev
/.
/usr
/usr/include
/usr/include/asm-generic
/usr/include/asm-generic/errno-base.h
/usr/include/asm-generic/auxvec.h
/usr/include/asm-generic/bitsperlong.h
/usr/include/asm-generic/errno.h
/usr/include/asm-generic/fcntl.h
/usr/include/asm-generic/int-l64.h
... and on and on ...
```

And make sure you understand what you're looking at above -- this is a package of header files that are available for inclusion in your *user* space programs, but are meant only for defining *kernel* space structures and other information so that your user code and kernel code can share the same declarations and definitions.

Exercise for the student: The obvious exercise -- take a few minutes and compare some of the the header files that were generated in your kernel source tree and verify that they exist under `/usr/include/` on your system. Any sane Linux development system should have some installed package that corresponds to exactly those headers.

And who decides which kernel header files are exported?

When you run:

```
$ make headers_install
```

from the top of your kernel source tree, who or what decides precisely which kernel header files will get bundled up and stashed under the kernel source directory `usr/include` for later "exporting" to user space? That's easy.

The header files to be exported are defined by the `Kbuild` files scattered throughout the kernel source tree. The one at the very top level is the engine, while elsewhere throughout the tree, you'll find `Kbuild` files like, say, this one:

```
$ cat include/Kbuild
header-y += asm-generic/
header-y += linux/
header-y += sound/
header-y += mtd/
header-y += rdma/
header-y += video/
header-y += drm/
header-y += xen/
header-y += scsi/
$
```

That file simply defines that the export process should recursively continue into those subdirectories and keep checking for more `Kbuild` files.

If we check further, we'll start to see `Kbuild` files like:

```
$ cat include/linux/Kbuild
header-y += byteorder/
header-y += can/
header-y += dvb/
header-y += hdlc/
header-y += isdn/
... snip ...
header-y += affs_hardblocks.h
header-y += aio_abi.h
header-y += arcfb.h
header-y += atmapi.h
header-y += atmarp.h
... snip ...
```

which clearly represents a combination of more recursive directories, plus immediate header files. Quite simply, all kernel `Kbuild` files have that general structure and, collectively (throughout the entire kernel source tree), they define all of the kernel header files to be exported to user space.

But there's one more detail ...

What does it mean to "sanitize" one of those header files?

In many cases, the header files to be exported contain some content that is meaningful *only* in kernel space, and it's only a *subset* of the header file that needs to be exported. Kernel-only code is normally surrounded by a preprocessor conditional that checks the value of the `__KERNEL__` macro, and part of the the job of the export process (when you run `make headers_install`) is to examine each file that is being exported, identify the code that is relevant only in kernel space, and remove it. Quite simple, really.

That's why (for example) the *kernel* version of the header file `include/video/edid.h` looks like this:

```
#ifndef __linux_video_edid_h__
#define __linux_video_edid_h__

#if !defined(__KERNEL__) || defined(CONFIG_X86)

struct edid_info {
        unsigned char dummy[128];
};

#ifdef __KERNEL__
extern struct edid_info edid_info;
#endif /* __KERNEL__ */

#endif

#endif /* __linux_video_edid_h__ */
```

but by the time it ends up in user space and is placed at `/usr/include/video/edid.h`, it looks like this:

```
#ifndef __linux_video_edid_h__
#define __linux_video_edid_h__


struct edid_info {
        unsigned char dummy[128];
};

#endif /* __linux_video_edid_h__ */
```

Technically, there's no actual harm in leaving in that kernel-only content since, when you're compiling in user space, you're guaranteed that the preprocessor macro `__KERNEL__` will never be set, but it's cleaner to just strip out that irrelevant content during the export process.

Aside: If you look carefully, you'll notice that many of the `Kbuild` files contain both the variables `header-y` and `unifdef-y` to identify the header files to be sanitized and exported. The latter is now deprecated and `Kbuild` files should now contain only the first form, but the older form is still supported.

Exercise for the student: You can list the files in a Ubuntu package with the `dpkg -L` command. So list the exported header files in the `linux-libc-dev` package with:

```
$ dpkg -L linux-libc-dev
```

and compare that output with what was generated by:

```
$ make headers_install
```

And that should do it. Next lesson: Back to subscriber-only content and debugging the kernel with `gdb`. Or something like that.

OBLIGATORY MARKETING SPIEL: If you were just passing by and stopped in to read this tutorial, make sure you check out the kernel programming course itself.

Source URL: http://crashcourse.ca/introduction-linux-kernel-programming/intermission-lets-talk-about-header-files-free-lesson

Links:
[1] http://crashcourse.ca/introduction-linux-kernel-programming/lesson-1-building-and-running-new-linux-kernel
[2] http://crashcourse.ca/introduction-linux-kernel-programming/introduction-linux-kernel-programming