Kernel Korner - The New Work Queue Interface in the 2.6 Kernel

Nov 01, 2003 By Robert Love (/user/800995)
in

Interrupt latency is a key factor in the performance of a system. Work queues are one of several tools available to the driver writer to avoid doing time-consuming work when interrupts are disabled.

(/issue/115)

From Issue #115 November 2003 (/issue/115)

For most UNIX systems, Linux included, device drivers typically divide the work of processing interrupts into two parts or halves. The first part, the top half, is the familiar interrupt handler, which the kernel invokes in response to an interrupt signal from the hardware device. Unfortunately, the interrupt handler is true to its name: it interrupts whatever code is executing when the hardware device issues the interrupt. That is, interrupt handlers (top halves) run *asynchronously* with respect to the currently executing code. Because interrupt handlers interrupt already executing code (whether it is other kernel code, a user-space process or even another interrupt handler), it is important that they run as quickly as possible.

Worse, some interrupt handlers (known in Linux as fast interrupt handlers) run with all interrupts on the local processor disabled. This is done to ensure that the interrupt handler runs without interruption, as quickly as possible. More so, *all* interrupt handlers run with their current interrupt line disabled on all processors. This ensures that two interrupt handlers for the same interrupt line do not run concurrently. It also prevents device driver writers from having to handle recursive interrupts, which complicate programming. If interrupts are disabled, however, other interrupt handlers cannot run. Interrupt latency (how long it takes the kernel to respond to a hardware device's interrupt request) is a key factor in the performance of the system. Again, the speed of interrupt handlers is crucial.

To facilitate small and fast interrupt handlers, the second part or bottom half of interrupt handling is used to defer as much of the work as possible away from the top half and until a later time. The bottom half runs with all interrupts enabled. Therefore, a running bottom half does not prevent other interrupts from running and does not contribute adversely to interrupt latency.

Nearly every device driver employs bottom halves in one form or another. The device driver uses the top half (the interrupt handler) to respond to the hardware and perform any requisite time-sensitive operations, such as resetting a device register or copying data from the device into main memory. The interrupt handler then marks the bottom half, instructing the kernel to run it as soon as possible, and exits.

In most cases, then, the real work takes place in the bottom half. At a later point—often as soon as the interrupt handler returns—the kernel executes the bottom half. Then the bottom half runs, performing all of the remaining work not carried out by the interrupt handler. The actual division of work between the top and bottom halves is a decision made by the device driver's author. Generally, device driver authors attempt to defer as much work as possible to the bottom half.

Confusingly, Linux offers many mechanisms for implementing bottom halves. Currently, the 2.6 kernel provides softirqs, tasklets and work queues as available types of bottom halves. In previous kernels, other forms of bottom halves were available; they included BHs and task queues. This article deals with the new work queue interface only, which was introduced during the 2.5 development series to replace the ailing keventd part of the task queue interface.

Introducing Work Queues

Work queues are interesting for two main reasons. First, they are the simplest to use of all the bottom-half mechanisms. Second, they are the only bottom-half mechanism that runs in process context; thus, work queues often are the only option device driver writers have when their bottom half must sleep. In addition, the work queue mechanism is brand new, and new is cool.

Let's discuss the fact that work queues run in process context. This is in contrast to the other bottom-half mechanisms, which all run in interrupt context. Code running in interrupt context is unable to sleep, or block, because interrupt context does not have a backing process with which to reschedule. Therefore, because interrupt handlers are not associated with a process, there is nothing for the scheduler to put to sleep and, more importantly, nothing for the scheduler to wake up. Consequently, interrupt context cannot perform certain actions that can result in the kernel putting the current context to sleep, such as downing a semaphore, copying to or from user-space memory or non-atomically allocating memory. Because work queues run in process context (they are executed by kernel threads, as we shall see), they are fully capable of sleeping. The kernel schedules bottom halves running in work queues, in fact, the same as any other process on the system. As with any other kernel thread, work queues can sleep, invoke the scheduler and so on.

Normally, a default set of kernel threads handles work queues. One of these default kernel threads runs per processor, and these threads are named events/*n* where *n* is the processor number to which the thread is bound. For example, a uniprocessor machine would have only an events/0 kernel thread, whereas a dual-processor machine would have an events/1 thread as well.

It is possible, however, to run your work queues in your own kernel thread. Whenever your bottom half is activated, your unique kernel thread, instead of one of the default threads, wakes up and handles it. Having a unique work queue thread is useful only in certain performance-critical situations. For most bottom halves, using the default thread is the preferred solution. Nonetheless, we look at how to create new work queue threads later on.

The work queue threads execute your bottom half as a specific function, called a work queue handler. The work queue handler is where you implement your bottom half. Using the work queue interface is easy; the only hard part is writing the bottom half (that is, the work queue handler).

Comments

Comment viewing options

Threaded list - expanded ▼ | Date - newest first ▼ | 50 comments per page ▼ | Save settings

Select your preferred way to display the comments and click "Save settings" to activate your changes.

A Simple Introduction to Device Drivers under Linux (/article/6916#comment-359462)
Submitted by Lokesh Venkateshiah (/users/lokesh-venkateshiah) on Fri, 11/19/2010 - 01:09.


(/users/lokesh-venkateshiah)

Since the misty days of yore, the first step in learning a new programming language has been writing a program that prints "Hello, world!" (See the Hello World Collection for a list of more than 300 "Hello, world!" examples.) In this article, we will use the same approach to learn how to write simple Linux kernel modules and device drivers. We will learn how to print "Hello, world!" from a kernel module three different ways: printk(), a /proc file, and a device in /dev.

Preparation: Installing Kernel Module Compilation Requirements

For the purposes of this article, a kernel module is a piece of kernel code that can be dynamically loaded and unloaded from the running kernel. Because it runs as part of the kernel and needs to interact closely with it, a kernel module cannot be compiled in a vacuum. It needs, at minimum, the kernel headers and configuration for the kernel it will be loaded into. Compiling a module also requires a set of development tools, such as a compiler. For simplicity, we will briefly describe how to install the requirements to build a kernel module using Debian, Fedora, and the "vanilla" Linux kernel in tarball form. In all cases, you must compile your module against the source for the running kernel (the kernel executing on your system when you load the module into your kernel).

A note on kernel source location, permissions, and privileges: the kernel source customarily used to be located in /usr/src/linux and owned by root. Nowadays, it is recommended that the kernel source be located in a home directory and owned by a non-root user. The commands in this article are all run as a non-root user, using sudo to temporarily gain root privileges only when necessary. To setup sudo, see the sudo(8), visudo(8), and sudoers(5) main pages. Alternatively, become root, and run all the commands as root if desired. Either way, you will need root access to follow the instructions in this article.

Preparation for Compiling Kernel Modules Under Debian

The module-assistant package for Debian installs packages and configures the system to build out-of-kernel modules. Install it with:

$ sudo apt-get install module-assistant
That's it; you can now compile kernel modules. For further reading, the Debian Linux Kernel Handbook has an in-depth discussion on kernel-related tasks in Debian.

Fedora Kernel Source and Configuration

The kernel-devel package for Fedora has a package that includes all the necessary kernel headers and tools to build an out-of-kernel module for a Fedora-shipped kernel. Install it with:

$ sudo yum install kernel-devel
Again, that's all it takes; you can now compile kernel modules. Related documentation can be found in the Fedora release notes.

Vanilla Kernel Source and Configuration

If you choose to use the vanilla Linux kernel source, you must configure, compile, install, and reboot into your new vanilla kernel. This is definitely not the easy route and this article will only cover the very basics of working with vanilla kernel source.

The canonical Linux source code is hosted at http://kernel.org (http://kernel.org). The most recent stable release is linked to from the front page. Download the full source release, not the patch. For example, the current stable release is located at http://kernel.org/pub/linux/kernel/v2.6/linux-2.6.21.5.tar.bz2 (http://kernel.org/pub/linux/kernel/v2.6/linux-2.6.21.5.tar.bz2). For faster download, find the closest mirror from the list at http://kernel.org/mirrors/ (http://kernel.org/mirrors/), and download from there. The easiest way to get the source is using wget in continue mode. HTTP is rarely blocked, and if your download is interrupted, it will continue where it left off.

$ wget -c "http://kernel.org/pub/linux/kernel/v2.6/linux-.tar.bz2"
Unpack the kernel source:

$ tar xjvf linux-.tar.bz2
Now your kernel is located in linux-/. Change directory into your kernel and configure it:

$ cd linux-
$ make menuconfig
A number of really nice make targets exist to automatically build and install a kernel in many forms: Debian package, RPM package, gzipped tar, etc. Ask the make system for help to list them all:

$ make help
A target that will work on almost every distro is:

$ make tar-pkg
When finished building, install your new kernel with:

$ sudo tar -C / -xvf linux-.tar
Then create a symbolic link to the source tree in the standard location:

$ sudo ln -s /lib/modules/'uname -r'/build
Now the kernel source is ready for compiling external modules. Reboot into your new kernel before loading modules compiled against this source tree.

"Hello, World!" Using printk()

For our first module, we'll start with a module that uses the kernel message facility, printk(), to print "Hello, world!". printk() is basically printf() for the kernel. The output of printk() is printed to the kernel message buffer and copied to /var/log/messages (with minor variations depending on how syslogd is configured).

Download the hello_printk module tarball and extract it:

$ tar xzvf hello_printk.tar.gz
This contains two files: Makefile, which contains instructions for building the module, and hello_printk.c, the module source file. First, we'll briefly review the Makefile.

obj-m := hello_printk.o
obj-m is a list of what kernel modules to build. The .o and other objects will be automatically built from the corresponding .c file (no need to list the source files explicitly).

KDIR := /lib/modules/$(shell uname -r)/build
KDIR is the location of the kernel source. The current standard is to link to the associated source tree from the directory containing the compiled modules.

PWD := $(shell pwd)
PWD is the current working directory and the location of our module source files.

default:
$(MAKE) -C $(KDIR) M=$(PWD) modules
default is the default make target; that is, make will execute the rules for this target unless it is told to build another target instead. The rule here says to run make with a working directory of the directory containing the kernel source and compile only the modules in the $(PWD) (local) directory. This allows us to use all the rules for compiling modules defined in the main kernel source tree.

Now, let's run through the code in hello_printk.c.

#include
#include
This includes the header files provided by the kernel that are required for all modules. They include things like the definition of the module_init() macro, which we will see later on.

```
static int __init
hello_init(void)
{
printk("Hello, world!\n");
return 0;
}
```
This is the module initialization function, which is run when the module is first loaded. The __init keyword tells the kernel that this code will only be run once, when the module is loaded. The printk() line writes the string "Hello, world!" to the kernel message buffer. The format of printk() arguments is, in most cases, identical to that of printf(3).

module_init(hello_init);
The module_init() macro tells the kernel which function to run when the module first starts up. Everything else that happens inside a kernel module is a consequence of what is set up in the module initialization function.

```
static void __exit
hello_exit(void)
{
printk("Goodbye, world!\n");
}
```

module_exit(hello_exit);
Similarly, the exit function is run once, upon module unloading, and the module_exit() macro identifies the exit function. The __exit keyword tells the kernel that this code will only be executed once, on module unloading.

```
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Valerie Henson ");
MODULE_DESCRIPTION("\"Hello, world!\" minimal module");
MODULE_VERSION("printk");
```
MODULE_LICENSE() informs the kernel what license the module source code is under, which affects which symbols (functions, variables, etc.) it may access in the main kernel. A GPLv2 licensed module (like this

one) can access all the symbols. Certain module licenses will taint the kernel, indicating that non-open or untrusted code has been loaded. Modules without a MODULE_LICENSE() tag are assumed to be non-GPLv2 and will result in tainting the kernel. Most kernel developers will ignore bug reports from tainted kernels because they do not have access to all the source code, which makes debugging much more difficult. The rest of the MODULE_*() macros provide useful identifying information about the module in a standard format.

Now, to compile and run the code. Change into the directory and build the module:

```
$ cd hello_printk
$ make
```

Then, load the module, using insmod, and check that it printed its message, using dmesg, a program that prints out the kernel message buffer:

```
$ sudo insmod ./hello_printk.ko
$ dmesg | tail
```

You should see "Hello, world!" in the output from dmesg. Now unload the module, using rmmod, and check for the exit message:

```
$ sudo rmmod hello_printk
$ dmesg | tail
```

You have successfully compiled and installed a kernel module!

Hello, World! Using /proc

One of the easiest and most popular ways to communicate between the kernel and user programs is via a file in the /proc file system. /proc is a pseudo-file system, where reads from files return data manufactured by the kernel, and data written to files is read and handled by the kernel. Before /proc, all user-kernel communication had to happen through a system call. Using a system call meant choosing between finding a system call that already behaved the way you needed (often not possible), creating a new system call (requiring global changes to the kernel, using up a system call number, and generally frowned upon), or using the catch-all ioctl() system call, which requires the creation of a special file that the ioctl() operates on (complex and frequently buggy, and very much frowned upon). /proc provides a simple, predefined way to pass data between the kernel and userspace with just enough framework to be useful, but still enough freedom that kernel modules can do what they need.

For our purposes, we want a file in /proc that will return "Hello, world!" when read. We'll use /proc/hello_world. Download and extract the hello_proc module tarball. We'll run through the code in hello_proc.c.

```
#include
#include
#include
```

This time, we add the header file for procfs, which includes support for registering with the /proc file system.

The next function will be called when a process calls read() on the /proc file we will create. It is simpler than a completely generic read() system call implementation because we only allow the "Hello, world!" string to be read all at once.

```
static int
hello_read_proc(char *buffer, char **start, off_t offset, int size, int *eof,
void *data)
{
```

The arguments to this function deserve an explicit explanation. buffer is a pointer to a kernel buffer where we write the output of the read(). start is used for more complex /proc files; we ignore it here. offset tells us where to begin reading inside the "file"; we only allow an offset of 0 for simplicity. size is the size of the buffer in bytes; we must check that we don't write past the end of the buffer accidentally. eof is a short cut for indicating EOF (end of file) rather than the usual method of calling read() again and getting 0 bytes back. data is again for more complex /proc files and ignored here.

Now, for the body of the function:

```
char *hello_str = "Hello, world!\n";
int len = strlen(hello_str); /* Don't include the null byte. */
/*
* We only support reading the whole string at once.
*/
if");
MODULE_DESCRIPTION("\"Hello, world!\" minimal module");
MODULE_VERSION("proc");
```

Then, we're ready to compile and load the module:

```
$ cd hello_proc
$ make
$ sudo insmod ./hello_proc.ko
```

Now, there is a file named /proc/hello_world that will produce "Hello, world!" when read:

```
$ cat /proc/hello_world
Hello, world!
```

You can create many more /proc files from the same driver, add routines to allow writing to /proc files, create directories full of /proc files, and more. For anything more complicated than this driver, it is easier and safer to use the seq_file helper routines when writing /proc interface routines. For further reading, see Driver porting: The seq_file interface.

Hello, World! Using /dev/hello_world

Now we will implement "Hello, world!" using a device file in /dev, /dev/hello_world. Back in the old days, a device file was a special file created by running a crufty old shell script named MAKEDEV which called the mknod command to create every possible file in /dev, regardless of whether the associated device driver would ever run on that system. The next iteration, devfs, created /dev files when they were first accessed,

which led to many interesting locking problems and wasteful attempts to open device files to see if the associated device existed. The current version of /dev support is called udev, since it creates /dev links with a userspace program. When kernel modules register devices, they appear in the sysfs file system, mounted on /sys. A userspace program, udev, notices changes in /sys and dynamically creates /dev entries according to a set of rules usually located in /etc/udev/.

Download the hello world module tarball. We'll go through hello_dev.c.

```
#include
#include
#include <
#include

#include
```

As we can see from looking at the necessary header files, creating a device requires quite a bit more kernel support than our previous methods. fs.h includes the definitions for a file operations structure, which we must fill out and attach to our /dev file. miscdevice.h includes support for registering a miscellaneous device file. asm/uaccess.h includes functions for testing whether we can read or write to userspace memory without violating permissions.

hello_read() is the function called when a process calls read() on /dev/hello. It writes "Hello, world!" to the buffer passed in the read() call.

```
static ssize_t hello_read(struct file * file, char * buf,
size_t count, loff_t *ppos)
{
char *hello_str = "Hello, world!\n";
int len = strlen(hello_str); /* Don't include the null byte. */
/*
* We only support reading the whole string at once.
*/
if (count < len)
return -EINVAL;
/*
* If file position is non-zero, then assume the string has
* been read and indicate there is no more data to be read.
*/
if (*ppos != 0)
return 0;
/*
* Besides copying the string to the user provided buffer,
* this function also checks that the user has permission to
* write to the buffer, that it is mapped, etc.
*/
if (copy_to_user(buf, hello_str, len))
return -EINVAL;
/*
* Tell the user how much data we wrote.
*/
*ppos = len;

return len;
}
```

Next, we create the file operations struct defining what actions to take when the file is accessed. The only file operation we care about is read.

```
static const struct file_operations hello_fops = {
.owner = THIS_MODULE,
.read = hello_read,
};
```

Now, create the structure containing the information needed to register a miscellaneous device with the kernel.

```
static struct miscdevice hello_dev = {
/*
* We don't care what minor number we end up with, so tell the
* kernel to just pick one.
*/
MISC_DYNAMIC_MINOR,
/*
* Name ourselves /dev/hello.
*/
"hello",
/*
* What functions to call when a program performs file
* operations on the device.
*/
&hello_fops
};
```

As usual, we register the device in the module's initialization function.

```
static int __init
hello_init(void)
{
int ret;

/*
* Create the "hello" device in the /sys/class/misc directory.
```

```
* Udev will automatically create the /dev/hello device using
* the default rules.
*/
ret = misc_register(&hello_dev);
if (ret)
printk(KERN_ERR
"Unable to register \"Hello, world!\" misc device\n");

return ret;
}

module_init(hello_init);
```

And remember to unregister the device in the exit function.

```
static void __exit
hello_exit(void)
{
misc_deregister(&hello_dev);
}

module_exit(hello_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Valerie Henson ");
MODULE_DESCRIPTION("\"Hello, world!\" minimal module");
MODULE_VERSION("dev");
```

Compile and load the module:

```
$ cd hello_dev
$ make
$ sudo insmod ./hello_dev.ko
```

Now there is a device named /dev/hello that will produce "Hello, world!" when read by root:

```
$ sudo cat /dev/hello
Hello, world!
```

But we can't read it as a regular user:

```
$ cat /dev/hello
cat: /dev/hello: Permission denied
$ ls -l /dev/hello
crw-rw---- 1 root root 10, 61 2007-06-20 14:31 /dev/hello
```

This is what happens with the default udev rule, which says that when a miscellaneous device appears, create a file named /dev/ and give it permissions 0660 (owner and group have read-write access, everyone else has no access). We would really like instead for the device be readable by regular users and have a link to it named /dev/hello_world. In order to do this, we'll write a udev rule.

The udev rule has to do two things: create a symbolic link and change the permissions on device to make world readable. The rule that accomplishes this is:

```
KERNEL=="hello", SYMLINK+="hello_world", MODE="0444"
```

We'll break the rule down into parts and explain each part.

KERNEL=="hello" says to execute the rest of the rule when a device with a name the same as this string (the == operator means "comparison") appears in /sys. The hello device appeared when we called misc_register() with a structure containing the device name "hello". See the result for yourself in /sys:

```
$ ls -d /sys/class/misc/hello/
/sys/class/misc/hello/
```

SYMLINK+="hello_world" says to add (the += operator means append) hello_world to the list of symbolic links that should be created when the device appears. In our case, we know this is the only symbolic link in the list, but other devices may have multiple udev rules that create multiple different symbolic links, so it is good practice add to the list instead of assigning to it.

MODE="0444" says to set the permissions of the original device file to the 0444 mode, which allows owner, group, and world all to read the file.

In general, it is very important to use the correct operator (==, +=, or =), or unexpected things will happen.

Now that we understand what the rule does, let's install it in the /etc/udev directory. Udev rules files are arranged in much the same manner as the System V init scripts in /etc/init.d/. Udev executes every script the udev rules directory, /etc/udev/rules.d, in alphabetical/numerical order. Like System V init scripts, the files in the /etc/udev/rules.d directory are usually symbolic links to the real rules files, with the symbolic links named so that the rules will be executed in the correct order.

Copy the hello.rules file from the hello_dev directory into the /etc/udev/ directory and create a link to it that will be executed before any other rules file:

```
$ sudo cp hello.rules /etc/udev/
$ sudo ln -s ../hello.rules /etc/udev/rules.d/010_hello.rules
```

Now, reload the hello world driver and look at the new /dev entries:

```
$ sudo rmmod hello_dev
$ sudo insmod ./hello_dev.ko
$ ls -l /dev/hello*
cr--r--r-- 1 root root 10, 61 2007-06-19 21:21 /dev/hello
lrwxrwxrwx 1 root root 5 2007-06-19 21:21 /dev/hello_world -> hello
```

Now we have /dev/hello_world! Finally, check that you can read the "Hello, world!" devices as a normal user:

```
$ cat /dev/hello_world
Hello, world!
```

```
$ cat /dev/hello
Hello, world!
```

Worker Thread Execution in SMP System (/article/6916#comment-341775)
Submitted by Anil123 (not verified) on Thu, 08/20/2009 - 23:30.

Hi,

In SMP system if we create work_queue thread is created on every processor. Can these excute in parallel or at a atime at most one of the worker
thread will be scheduled in SMP system.

Regards,
Anil

Could this be updated for the 2.6.20 kernel? (/article/6916#comment-332810)
Submitted by chuck (http://www.linuxjournal.com) (not verified) on Fri, 02/06/2009 - 11:46.

This is a nice introduction. It would be nice if it was updated for the 2.6.20 kernel.

Thanks for the wonderful information (/article/6916#comment-329663)
Submitted by kasiviswanathan (not verified) on Wed, 12/03/2008 - 06:06.

Love has done a great job here by explaining almost the whole workqueue concept here.

Thanks very much! I've been (/article/6916#comment-328682)
Submitted by Anonymous (not verified) on Mon, 11/17/2008 - 08:49.

Thanks very much! I've been looking for an ability to read registers of some I2C in ISR() or in some bottom half!
Firstly I've been surprised to see my ISR()/tasklet corrupt the kernel, but then I've saw that I2C functions performs SLEEP! :)
And I thought that I need some bottom half that able to perform I2C bus read.
Finally by the help of this article, I've solved my problems.

Thanks.
Yan.

Priority of custom threads? (/article/6916#comment-126444)
Submitted by Anonymous (not verified) on Thu, 10/20/2005 - 07:18.

Robert,
when we create workqueues using custom threads, how can we make them run at a higher priority. Doesnt the workqueue have an interface to increase the priority of the worker thread(s) ?

Thanks
BD