UCSD Embedded Linux online
Week 8

Slide 1

This week we're going to drop down into kernel space and look at the process of writing device drivers for Linux. This will be a very brief introduction to a very extensive topic. The object is to get to the point where you can write a simple driver to access hardware on the target board. But we'll begin our investigation with a simple memory-only driver running on the workstation.

To dig deeper into the fascinating subject of device drivers, I recommend the book *Linux Device Drivers* by Alesandro Rubini, et al that's listed in the bibliography. It's a very readable and very thorough treatment of programming to the kernel APIs although it is getting a bit dated as it deals with kernel version 2.6.10. A more up to date reference is *Essential Linux Device Drivers* by Sreekrishnan Venkateswaran. This one is also quite thorough, but I find the organization a bit odd.

Slide 2

So what the heck is a "device driver" anyway? Fundamentally, it's simply a way to "abstract out" the hardware so the application programmer doesn't have to worry about it.

In simple systems a set of linkable functions may be sufficient. As systems get larger and more complex, it is often useful to structure a device driver as one or more tasks that communicate with application tasks through the operating system's communication mechanisms.

In the context of Linux, a device driver is an independently loaded module of code that conforms to a well-defined set of driver APIs. In a large, complex OS like Linux, writing a device driver is a non-trivial exercise because there are just a lot of details to be aware of.

You might also be asking the question, why do we want to write device drivers? We've already seen that it's fairly easy to access peripheral registers from User Space and in fact we even invented our own simple set of driver APIs. Certainly, for conventional peripherals like disks and keyboards, printers, and so on, the drivers already exist. Why not use them? But the advantage of using the standard driver APIs for your own, unique devices, is that they become accessible from all the standard Linux utilities.

Slide 3

This slide attempts to capture the high-level structure and organization of the Linux kernel. The services that the kernel provides to applications can be divided up into five major categories:

Process Management
Linux is organized around the concept of multiple processes running on a single processor. The kernel is responsible for creating and destroying processes, scheduling their use of the CPU, and managing communication among them.

Memory Management
Linux treats memory as a "scarce resource" and proper management of this resource is critical to the system's performance. And of course, all processes operate in their own private virtual address space.

File Systems
Files are an essential abstraction in Linux where just about everything can be represented as a file. Linux supports multiple file system types that in turn are supported by specific block device drivers.

Device Control

Almost every system operation eventually maps to some physical device. The role of a device driver then is to abstract out the peculiar features of the device it represents into some uniform API.

Networking
Most network operations are not specific to a process and so must be managed by the kernel. Process scheduling is dependent on incoming packets that arrive asynchronously. Also, the kernel has to deal with routing and address resolution issues.

Slide 4

Just to quickly review from last week, everything in Linux is a file, including physical hardware devices, which have their own entries in the file system, usually in the /dev directory.

Devices come in four "flavors": character, block, pipe, and network. The first three classes treat data as a simple stream of bytes. Block devices are for mass storage and, of necessity, have a file system imposed on them. Pipes are just unidirectional links between processes.

Network devices are different in that they handle "packets" of data for multiple protocol clients rather than a "stream" of data for a single client. This necessitates a different interface between the kernel and the device driver. Network devices are not nodes in the /dev directory.

Slide 5

Most of this slide is also from last week. This is the basic low-level user interface to devices and files. The open call creates a connection to a file or device and returns a *file descriptor*, an integer index, which is used as an argument to all other I/O calls to identify the specific connection.

Low-level I/O is relatively simple and straightforward, but it's also somewhat inefficient because data isn't buffered and so every call to one of these functions causes a switch from user space to kernel space, even if you're only transferring one byte. Switching to kernel space is expensive because there are hoops the system has to jump through to securely transition from User privilege level to Kernel privilege level and back again.

Slide 6

The alternative user interface is called the Standard I/O Library, or stdio for short. It provides a versatile interface to the low-level I/O system calls and often has better performance because it buffers data and only invokes a kernel call when there's enough data to make it worthwhile. As shown here, the stdio API looks a lot like the low-level API.

While buffering generally improves performance by reducing the number of transitions to the kernel, it also means that, for example, on return from fwrite, the data has not necessarily been written to the device. This could be a problem in deterministic, real-time environments. The other problem is that there is no equivalent to ioctl in the stdio library. So any device-dependent control must still be handled by the low-level functions.

Slide 7

There are a couple of different ways that device drivers can be installed, or integrated with the kernel. A driver can be compiled and linked directly into the kernel image, becoming a part of the kernel itself. Drivers for the most common peripherals and those required to boot the system are built this way.

But it would be virtually impossible, not to mention relatively pointless, to build drivers for all possible devices into the kernel image. So, most device drivers are in fact built as *installable kernel modules* to be dynamically integrated into the kernel when their functionality is required.

Fundamentally, installable kernel modules offer a way to extend the functionality of the kernel without having to rebuild it. When a module's functionality is required, it is dynamically "installed", or integrated into the kernel. Later, if its functionality is no longer required, the module can be removed from the kernel.

Modules are also a very useful way to test kernel code before building it directly into the kernel image.

Being part of the kernel, modules run at kernel privilege level and are thus capable of bringing down the entire system.

Slide 8

A module is *installed* into the kernel by executing the insmod command. When that functionality is no longer needed, the module can be removed, and its memory space reclaimed, by executing rmmod. Note that insmod allows parameters to be passed to the module.

You must be root or Super User to invoke these commands. Remember that the shell command su lets you become root user temporarily. If you become root with su, you will probably have to preface the insmod and rmmod commands with /sbin/ because /sbin is not normally in a regular user's path.

The basic requirement for a kernel module (other than being thoroughly debugged, of course) is that it include at least two functions, one of which is called as part of insmod to initialize the module and the other is called as part of rmmod to "clean up" when the module is removed.

Slide 9

This slide illustrates the basic role of the module initialization function. Basically, its role is to make the module's presence known to the kernel and inform the kernel what it's capable of doing. It does this by calling one or more kernel services that *register* various capabilities such as device drivers. In effect, the module is saying "here I am and this is what I can do."

This also illustrates how parameters are passed into a module. Parameters can be strings or signed or unsigned integers of various lengths.

Note also the module_init() macro that identifies the function that initializes the module.

Slide 10

This slide illustrates graphically how a module becomes part of the kernel. As we noted in the last slide, the primary role of init_module is to "register" the module with the kernel, typically by calling some sort of "register capabilities" function and passing it a pointer to a table of function pointers in the module. This is what allows the kernel, and ultimately user space applications, to get access to the module's functionality.

The module's functions can in turn call functions in the kernel. Now think about this. The kernel and the module are independently loaded entities. From the module's point of view, the kernel functions are unresolved externals that need to be resolved through some form of runtime linking process. So how do those symbols get resolved?

The answer is that when the kernel is loaded, its symbol table is also loaded. So insmod resolves the module's external symbols against the kernel's symbol table. Furthermore, any global symbols defined in the module get added to the kernel symbol table so that subsequently loaded modules can make use of the facilities in this module. There are many instances of such hierarchical module relationships.

Slide 11

Our trivial module example is a variation on the standard Hello World program. Incidentally, all of the examples for this week's class are found under the directory drivers/ in your home directory. You can create a new Eclipse makefile project in drivers/hello or you can choose to view the files with another editor and build the project from a shell window.

In any case, take a look at hello.c. This illustrates a number of module concepts such as:

- What header files are required
- What symbols must be defined

- How parameters are declared and used

It also illustrates several "documentation" macros that can be used to provide additional information about the module. The utility modinfo is used to output this information.

Note the use of printk() instead of printf(). C library functions like printf() are not available to the kernel. printk() is the kernel equivalent that writes its string to the file /var/log/messages and also to the console if the message log level, the number in brackets, is of sufficiently high priority where a lower number is a higher priority. Unfortunately, this doesn't work from X Windows shells. So we use the tail command to see the last few lines written to the messages file. You can enter this command either as tail –f or tailf, one word, and it will continue to "follow" the messages file.

Now look at the Makefile. Starting with the 2.6 series kernels, modules can only be built from within the kernel source tree. And that, by the way, is the source tree of the kernel the module will run with. Modules are kernel version specific because kernel APIs can change from version to version and a module compiled for one kernel version may not run with a different version. It is possible to turn off version checking in kernel configuration, but unless you have a very good reason, it's not a good idea.

This makefile invokes make again with the –C option to change to the kernel source tree directory and pass in a variable M that is the directory where this makefile was invoked from. The kernel's makefile then includes this one for the purpose of getting that first line that starts "obj-m." That's the key that says build a loadable module from the file hello.o.

If you created an Eclipse project, the module should already be built. If not, execute make in a shell window. Even though modules really are just relocatable object files, they've been given a different extension, .ko, to make it clear that they really are kernel modules.

Now execute the lsmod command shown here to see a list of modules currently loaded in your system. There are a lot of them. They're listed in reverse order in which they were loaded. So the most recently loaded module is listed first.

Execute the tailf command in a different shell window and then, as root user, execute the insmod command. Execute lsmod again and you should see hello show up at the top of the list. Now look at the messages file. Two things happened. The init function output its message, but just before that is a rather ominous looking message about "tainting" the kernel. Ooooh, what does that mean?

A 'tainted" kernel is one that has one or more binary-only modules installed. That is, the source code for the module is not available. The kernel maintainers were getting tired of dealing with bug reports that involved proprietary, binary only modules. So they added a "module license feature" in 2.6 whereby you can declare that your module is open source. There's a set of valid license strings defined in module.h that you can specify as the argument to the MODULE_LICENSE() macro.

Loading a binary only module sets a flag in the kernel that remains set until you reboot. If the kernel should subsequently crash, that flag is included in the crash dump and that's how the kernel maintainers know that they are free to ignore that one.

Go ahead and execute the rmmod command and note that the exit function's message shows up. Execute lsmod again and hello is no longer in the list.

Slide 12

Linux, as we all know, is released under the Gnu General Public License (GPL). This has implications for how you integrate device drivers into the system.

Everyone agrees that applications that run in User Space and only make use of published kernel APIs are not "derivative works" as defined by the GPL and thus can remain proprietary. On the other hand, any code that is linked statically into the kernel image is, by definition, a derivative work and must itself be released under the GPL with source code available.

Dynamically loaded kernel modules are a middle ground. Most people accept that modules need not be open source, although as we saw in the last slide, the kernel will at least slap your wrist if you run proprietary module.

Nevertheless, there are kernel developers who would like to ban proprietary, binary-only modules and there was talk at one point of not allowing such modules to load. Linus said absolutely not, that such a move would destroy the commercial viability of Linux. Many hardware vendors have a legitimate business interest in maintaining their drivers as proprietary. If they couldn't do that they simply wouldn't support Linux.

So the point is, if you want to maintain your driver as proprietary, don't build it into the kernel image; only distribute it as a module. Granted, this is semantic hair splitting here, but it does serve a useful purpose.

Slide 13

OK, with that background, let's move on to device drivers. The very first thing a driver needs to do is allocate one or more *device numbers* for the devices it manages. This would usually be done in the module init function. Device numbers are 32 bits—the high 12 bits represent the major number and the low 20 bits the minor number. Of course, good programming practice says you should never rely on a specific bit allocation of something like a device number, so the kernel provides a set of macros for creating and dissecting them.

There are two ways to get a range of device numbers for your driver. If you need a specific range of numbers, you use the function register_chrdev_region(), which may return an error if the requested region is not available.

The preferred mechanism is to ask the kernel to dynamically allocate the required range using the function alloc_chrdev_region(). This will put a starting device number in the output parameter dev.

Slide 14

With a device number in hand, we can now connect our driver to the kernel. This requires a cdev structure, which can be allocated dynamically through the function cdev_alloc(), or statically, often as part of some device-specific structure as we'll see in just a bit.

Before we can add the driver to the kernel, there are two fields that must be initialized. The ops field must point to a file operations structure that we'll see in the next slide and the owner field normally points to THIS_MODULE.

The function cdev_add() makes the driver known to the kernel. If it succeeds (it almost always does), the driver is "alive" and ready to be accessed.

Slide 15

The most important field in the cdev data structure is the *File Operations Table*. With the exception of the first field, it is a list of pointers to functions in the driver that implement the driver APIs.

Slide 16

The file operations table has grown considerably over the years (it used to fit on a single slide). Many of the additional functions relate to asynchronous I/O and operations on files that devices rarely need.

A driver only implements the API functions that it actually needs. All other entries in the file operations table can be left NULL and the kernel will either return an error or perform a sensible default action.

Slide 17

Another structure useful to device drivers is the file struct that gets passed to every driver function. The most interesting field here, and the only one that a driver should ever set, is private_data. This allows a driver to allocate whatever it may need on a per connection basis and pass that data around to all of the driver functions. Private data is usually allocated by the open function.

Slide 18

Our first device driver is called, appropriately enough, "simple_char." It doesn't do much beyond illustrating many of the basic device driver mechanisms that we've looked at. simple_char acts on a memory area as if it were a device.

Again, it's your choice whether you want to do this in Eclipse or with some other editor. Open simple_char.c and start at the top. A number of operational parameters can be set either at compile time or at load time through the module_param() mechanism. By default the major number is set to 0, which means assign it dynamically.

simple_char creates four devices with minor numbers 0 to 3. Take a look at simple_init_module() down near the bottom of the file. It allocates a range of device numbers, then allocates memory for device structures, including a *mutex* for each device. More on that later.

Look at the file operations structure a little farther up. This uses the "tagged" initialization format where structure fields are identified by name. Fields not included in the initialization are set to NULL. This is a very useful C feature that avoids compatibility issues when the structure changes.

Go back up and find simple_open(). It sets up a private data structure, which is just a pointer to the dev structure and sets the device's data area to zero length if it's being opened as write only. Note that we must protect against multiple accesses to the dev structure using a mutex.

Slide 19

The macro container_of() deserves some explanation because it shows up frequently in kernel programming. Each of the simple_char devices has a struct simple_dev_t associated with it that will become the private_data entry for the connection. How does open() find the address of the correct struct simple_dev_t?

Remember that every device has an associated struct cdev, which in this case is declared in struct simple_dev_t and pointed to by struct inode passed into open(). container_of() returns the address of a struct simple_dev_t that contains the cdev structure passed as its first argument, inode->i_cdev. container_of() is defined in linux/kernel.h.

Slide 20

Run make to build the driver, unless you're using Eclipse in which case it should already be built. The next step is to install the newly created driver module. But we won't use insmod this time because there's one final step to take before we can actually use simple_char. We have to create a set of inodes in /dev representing the devices that simple_char manages.

That's what the shell script simple_load does. It first installs the module with insmod, then uses awk on the file /proc/devices to find simple_char's major number, which was dynamically assigned by the kernel.

Try it out. Execute the simple_load script. Then use cp to write something to simple0. Then use cat to read it back.

Slide 21

Let's take a look at how simple_char does reading and writing. Data transfer begins at the current file position, f_pos, passed into the read and write methods. For our purposes, f_pos is just an index into the data array.

A pointer to the dev structure, containing a pointer to the data array, is passed as the private_data member of the filp file structure.

This slide graphically illustrates the process of writing data to a device through a device driver. The file descriptor argument to write contains the device's major device number, which becomes an index into the chrdevs table. This in turn selects the file operations structure for the selected device. The application-level library function write makes a system call, which is translated by the syscall table to select the write function pointer from fops.

The data to be written must be moved from user space, which is swappable, to kernel space, which is not. Conversely, for a read operation, data must be moved from kernel space to user space. There's a lot going on here.

And yet conceptually, it's quite simple. Remember back in slide 2 I suggested that a device driver may be nothing more than a set of functions linked to the application that implements a specific set of APIs. Well, our Linux driver is really that – a set of functions that user space processes invoke. It's just that the process of calling these functions is rather convoluted because of the need to transition between user space and kernel space.

Slide 22

Device drivers are subject to being called by multiple processes "simultaneously". For example, two processes, A and B, could have the same simple_char device open for writing. Process A starts a write operation, but is then preempted by process B, which also starts a write operation. Unless we do something to assure each process exclusive access to the data area, the result will be wrong.

The kernel provides *mutexes* as synchronization objects. So just as we used a mutex to synchronize access to common data in our thermostat, we'll use a kernel mutex that allows only one process at a time to access the device data buffer. Other processes attempting to access it must wait until the current mutex holder is done.

The mutex API is shown here. Before using a mutex, it must be initialized.

A process acquires the mutex by calling mutex_lock() or mutex_lock_interruptible(). If the mutex is free, these functions lock the mutex and return immediately. If the mutex is already locked by another process, the current process is blocked. mutex_lock() does not return until the mutex becomes available. mutex_lock_interruptible(), on the other hand, returns with a non-zero error code if the process is interrupted by a signal. In this case, the calling function should return –ERESTARTSYS, telling the kernel to either retry the operation or return –EINTR to the application.

A mutex is released by calling mutex_unlock(). If one or more processes are waiting on it, mutex_lock() makes one of the processes ready.

Slide 23

Just for the record, here is the old, pre 2.6 device registration protocol. It's worth knowing this because there may still be a few legacy drivers out there that haven't been upgraded to the new APIs.

This mechanism uses one function to register a char device and another to register a block device. In both cases, the arguments are: a major device number, a device name, and a file operations structure. The return value is either the major device number or an error code. If the device number passed in is zero, the function returns a dynamically assigned major device number.

Slide 24

Kernel level code presents its own set of problems when it comes to debugging. There's a fundamental problem with debugging kernel code using gdb. gdb is an application that relies on kernel services. If you stop the kernel at a breakpoint, gdb effectively stops.

There are some tools that allow GDB to be used on kernel code. Probably the most useful is kgdb, a stub much like gdbserver that is linked into the kernel you want to debug. You then run that kernel on a separate machine in the same way that we ran GDB with our target board. We don't have time to get into kgdb here, but the web resources document has a useful link if you want to pursue it.

The fact is, most kernel hackers use printk for debugging. Personally I've never been a big fan of the printf debugging strategy. On occasion it's proven useful but more often than not the print statement just isn't in the right place. So you have to move it, rebuild the code and try again.

We saw printk earlier in our simple module example. This slide shows in more detail how it actually works. Rather than writing directly to a console terminal, printk writes to a circular buffer in memory and then wakes up any process waiting on the *syslog* system call.

To make a long story short, the process waiting on the syslog call is klogd and its default behavior is to write anything in the circular buffer to the log file /var/log/messages and, if the *loglevel* is high enough, print it to the current console.

Incidentally, one consequence of this strategy is that printk can be invoked from within interrupt service routines whereas most I/O operations can't because of their potential for blocking.

Slide 25

The main problem with printk is that each invocation causes a disk write. That's because syslogd's objective is to write log data as soon as possible so that in case of a crash, there's something to look at.

This means there's a performance hit whether or not you need the information from a particular printk call. This is probably OK while you're debugging, but you certainly don't want to leave the printk statements in the production version of the driver. You don't want your customer puzzling over all these weird messages in the messages file. Of course as soon as you take the printks out, someone will discover a bug, or you'll add a new feature and you'll need to put them back in again.

An easy way to manage this problem is to encapsulate the printk's in a macro as shown here. Note however that this macro relies on a gcc extension to the ANSI C preprocessor that supports macros with a variable number of arguments. But since we're dealing with kernel level code, which itself relies heavily on gcc extensions, this shouldn't be a serious portability issue.

simple_char defines just such a macro controlled by the symbol DEBUG declared in the makefile. There are already a couple of PDEBUG statements in simple_char. Access the device and watch what comes out on the messages file. Think about where else you might want to put debug statements.

Slide 26

The /proc file system is another way to gain visibility into what's happening in kernel code. It acts just like an ordinary file system. You can list the files in the /proc directory, you can read and write the files, but they don't really exist. The information in a /proc file is generated on the fly when the file is read. The kernel module that registered a given /proc file contains the functions that generate read data and accept write data.

/proc files provide dynamic information about the state of a kernel module in a way that is easily accessible to user-level tasks and the shell. In effect, they provide a window from user space into kernel space. In the abbreviated directory listing shown here, the directories with number labels represent processes. Each process gets a directory under /proc with several subdirectories and files describing the state of the process.

To prove that the information is dynamic, do the following:

cd to the /proc directory
execute cat interrupts

This lists interrupt numbers, the devices they service and the number of times each interrupt has fired off since the system booted. Execute the same command again. Several of the numbers have gone up, in particular the timer tick interrupt.

Slide 27

/proc files can be a useful mechanism for gaining visibility into what's happening in your device driver. Unlike printk(), which always prints if it's compiled in, a /proc file only returns information when you ask for it.

In its simplest form, a /proc file is nothing more than a read method that is registered as a /proc file. This method looks identical to the read method for a device driver. When a User space process reads the file, the registered method is called. A file operations structure is required with only two entries: the owner and read fields.

The create_proc_read_entry() function specifies the name of the file, where in the /proc hierarchy it goes, and a pointer to the file operations structure. If the parent argument is NULL, the file is created in the root of /proc.

Of course when a module is removed, any /proc files it created should be removed also. Otherwise, attempting to read that file would most likely crash the system.

simple_char has a /proc file implementation controlled by the compile-time symbol DEBUG. Starting at line 66 in simple_char.c is a short section bracketed by #ifdef DEBUG. The DEBUG flag is turned on in the Makefile, so the /proc file implementation is built in.

Slide 28

The ioctl method was mentioned briefly when we looked at the low-level I/O system calls. This is another way to get information out of a driver as well as set information into it. From the User Space side, ioctl() has a variable argument list. In reality, it's just a single optional argument, traditionally identified as char *argp, because we can't transfer an arbitrary number of arguments into the kernel across the call gate.

Generally, transferring small amounts of information using ioctl() is easier to code than a /proc file. The downside is you need a User Space program specifically for your driver's ioctl() parameters.

Also, a /proc file is always potentially visible to the user and so good practice dictates removing a debug-oriented /proc file from production code. ioctl() commands, on the other hand, are not really visible if they don't appear in user documentation. So it's usually safe to leave debug-oriented ioctl commands in the driver at the cost of a little extra code space.

Slide 29

It used to be that assigning ioctl() commands was a pretty ad hoc affair. In the really old days, a driver writer would just start at 1 and go up as far as he needed. But that makes it all too easy to send the wrong command to a device—trying to set the baud rate on a printer, for example.

The new method breaks the command into four bit fields as shown here. Pick a magic number for your driver after consulting the file ioctl-number.txt in the kernel documentation directory. There is a set of macros that help set up command numbers. These are illustrated in the project file simple_ioctl.h.

This approach allows the driver to do some sanity checking on the commands passed to it.

Slide 30

If the argument to an ioctl command is just an integer, it's straightforward. You just use the integer directly. It's equally straightforward if your ioctl command returns something as the function value. But what if the argument is a pointer?

The problem is, that pointer is pointing into user space. So just like we used copy_to_user() and copy_from_user() to move data between user space and kernel space, here we use put_user() and get_user(). These macros are optimized for the short data items typical of ioctl commands.

access_ok() checks that the pointer really is pointing into user space. The put_user() and get_user() forms without the underscores invoke access_ok(). The ones with the underscores don't and thus are a little faster.

There may be cases where we want to restrict ioctl() access to privileged users. In Unix, privilege is typically an all-or-nothing proposition. Either you're the superuser or you aren't. The newer Linux kernels break privilege down into a set of "capabilities". Different users and different processes can have access to different capabilities. The capable() function returns 1 if the calling process has the specified capability.

With that background, take a look at the ioctl() implementation in simple_char.c starting at line 185. You should also open simple_ioctl.h. The commands use different methods to get and set the used field in the simple_dev_t structure. The user space application, ioctl, was built along with the simple_char module.

Slide 31

In simple_char, if data is not available when read is called, it simply returns with a count of zero. But in most real-world cases, if data is not available, what you really want to do is have the process wait until data becomes available. Returning a count of zero when we're not really at the end-of-file is generally not a useful thing to do.

Likewise, the write method may not be able to write any data because the output buffer is full and the device is not able to accept data. Again, returning a count of zero is not particularly useful.

In these cases, we want the process to "sleep" until a data transfer is possible and then have it "wake up" somehow. What we mean by sleep is that the process is blocked and another ready process gets to execute. In Linux, the mechanism for putting a process to sleep is a wait queue, now known as a wait queue "head".

Slide 32

Here is the wait queue API. A wait queue can be declared and initialized statically with one macro invocation. Or it can be dynamically declared and initialized with a function call.

As with the mutex, we have the option of waiting with signals blocked or not. Usually you want to use the "interruptible" version, which returns a non-zero value if the call was interrupted by a signal. There's also an option to wait for a maximum time, expressed in jiffies. We'll talk more about jiffies next week.

condition is any Boolean expression. Two things must happen before a process can wake up. Another process must execute a wake_up() on the corresponding wait queue AND the condition must be TRUE (non-zero). The wait_event() calls are actually macros that expand to a while loop that tests the condition on a wake up.

wake_up() wakes up all processes waiting on the queue. wake_up_interruptible() only wakes up those that went to sleep interruptible. In general, the two are indistinguishable if you're using interruptible sleeps. In practice, the convention is to use wake_up() if you're using wait_event() and wake_up_interruptible() if you use wait_event_interruptible().

The sleep API is more extensive than this, but this is sufficient for our purposes.

Slide 33

Here's a trivial example of process sleeping. Think of these as the read and write methods of a device driver. Let's say that Process A calls sleepy_read(). The flag is zero, so wait_event_interruptible() causes Process A to sleep. Then Process B calls sleepy_write(), setting the flag to 1. Then it calls wake_up_interruptible() on the same queue where Process A went to sleep. The condition is now true, so Process A wakes up and continues..

Note incidentally that the wait queue is passed *by value* to the wait_event() macros.

Slide 34

One more item before we look at some real code. While the normal behavior of read and write is to block if data transfer is not possible, there are occasions where the appropriate response is to return immediately with an error code. That's the purpose of the O_NONBLOCK flag. O_NONBLOCK is set to zero by default. When it's non-zero, a function should immediately return the appropriate error code if no data transfer is possible.

The device for experimenting with blocking operation is called blocking, located in the directory blocking_char. It implements a FIFO queue.

Open blocking.c with an editor and go down to the pipe_init() function at line 401. For each device it sets up a pipe_dev_t data structure with two wait queues, a mutex and the appropriate fields to manage a circular buffer.

blocking uses a newer, more efficient method of creating device nodes for the driver. Recent kernels have a concept of *device classes* that group together devices with similar characteristics. We don't have time to get into the details here, but note that at line 429 we create a class.

Then in the initialization loop at line 453 we create devices within that class. One of the things create_device() does is to create the corresponding node in the /dev directory. The upshot is we don't need a load script. We can just insmod the module.

Go back up to the pipe_open() function at line 140. It allocates a data buffer if necessary. blocking is designed to handle any number of simultaneous readers and writers. So there is a pair of fields to track how many readers and writers are currently connected. When both counts reach zero in release(), the buffer can be freed. Note that access to the data structure is protected by the mutex. pipe_open() ends with a call to nonseekable_open(), which tells the kernel that this driver does not support the seek operation.

pipe_read() at line 199 handles both blocking and non-blocking I/O. It first acquires the mutex and tests for the presence of data. If data is available read() returns it to the user and we're done. If the buffer is empty, and the O_NONBLOCK flag is not set, then we release the mutex and go to sleep.

Upon waking up, there are two possibilities: the process received a signal, in which case the kernel deals with it, or the condition in the wait_event_interruptible() call was satisfied. But that's not the end of the story because wakeup() wakes up <u>all</u> processes waiting on the queue. Very likely, the first one of those multiple processes that gets to run is going to empty out the buffer again. So we have to reacquire the mutex and test for data present. Finally read() wakes up any writers who may be waiting for space to become available in the buffer. Incidentally, the reason for waking up everyone on the queue is to let the scheduler decide who gets to run. So processes wake up in priority order rather than simple FIFO order.

pipe_write() illustrates another, lower level, approach to sleeping. Look at the function pipe_getwritespace() starting at line 242. DEFINE_WAIT() creates a wait queue entry that is placed on the queue with prepare_to_wait(), which also sets the process state to TASK_INTERRUPTIBLE. Then we invoke the scheduler to run another process. When this process wakes up again, finish_wait() cleans up the wait queue and we again test to see if space is available.

Play around with blocking by opening multiple processes to read and write it. Monitor the driver's behavior with the PDEBUG() statements. Note incidentally that there's a subtle bug in the code. If you open the device for reading first, then write to it, things work as expected. But if you open it for writing first, fill the pipe, then read, both sides hang. Use PDEBUG() statements to find the bug, then fix it.

Slide 35

One last item before we wrap up this week. blocking illustrates an alternate approach to implementing a /proc file. The original /proc paradigm is not particularly useful for outputting large amounts of data as, for example, if you want to iterate over an array or linked list of data structures. The seq_file interface (I guess the abbreviation is meant to connote "sequence") was invented to address that issue. Your driver supplies implementations of the four function prototypes shown in this slide and registers the file with seq_open().

The kernel calls seq_start() to begin reading the file. seq_start() returns a private pointer to the first item to be output. seq_next() is called to iterate to the next item in the list. When seq_next() returns NULL, seq_stop() is called to finish up. After the seq_start() call and each call to seq_next(), seq_show() is called to actually write the data using a set of functions specifically for seq files.

Have a look at the seq_file implementation in blocking starting around line 70 of blocking.c.

Slide 36

Review time. This week we dipped our toes into the waters of kernel programming. We began with a brief review of the user space I/O APIs and the four classes of devices in Linux. Then we looked at loadable kernel modules as a way of extending kernel functionality dynamically at run time. As part of that, we noted that kernel modules have a "special" relationship to the GPL.

Next we introduced a simple character driver that shows how the driver interacts with the rest of the kernel. We looked at several common techniques for debugging kernel code. Finally we looked at blocking I/O whereby if data transfer is not possible, the calling process sleeps until transfer is possible.

Well, this week's episode has been largely preparation for what we'll tackle next week, which is using device drivers to manipulate hardware on the target board. Trust me, that'll be fun.