UnixWorld Online: Tutorial Article No. 010

Write a Linux Hardware Device Driver

Learn to write and install a Linux device driver to control a hardware card. We detail what functions need to be written, outline the supporting kernel functions that are available, explain how to initialize the driver and how memory is requested and allocated in an efficient manner, and provide a ''real'' driver example.

By Tom Coffey and Andrew O'Shaughnessy

Introduction to Linux

Linux is a 32-bit multitasking, multimedia operating system with complete source code, developed by the free software community on the Internet. Linux is a clone of the Unix operating system that runs on Intel 80386/80486/Pentium computers. It supports a wide range of software, from TEX to the X Window System, to the GNU C/C++ compiler, to TCP/IP. The Linux system is mostly compatible at the source level with a number of Unix standards including IEEE POSIX.1, System V, and BSD. Linux also provides a complete Unix programming environment, including standard libraries, programming tools, compilers, and debuggers.

A device driver consists of a set of routines that control a peripheral device attached to a workstation. The operating system normally provides a uniform interface to all peripheral devices. Linux and Unix present peripheral devices at a sufficiently high level of abstraction by observing that a large proportion of I/O devices can be represented as a sequence of bytes. Linux and Unix use the file--which is a well understood data structure for handling byte sequences--to represent I/O devices.

Linux I/O Subsystem

Figure 1 shows the Linux architecture in the most general terms. Here, the kernel is shown wrapped around the hardware to depict that it is the software component that has direct access to--and control over--the system hardware, including the processor, primary memory, and I/O devices.

Figure 2 [Bach86] shows that user-level programs communicate with the kernel using system calls, for instance, `open()`, `read()`, `write()`, `ioctl()`, `close()`, and the like.

Linux System Calls

The kernel is not a separate task under Linux. It is as if each process has a copy of the kernel. When a user process executes a system call, it does not transfer control to another process, but changes its execution mode from user to kernel mode. In kernel mode, while executing the system call, the process has access to the kernel address space, and through supporting functions, it has access to the address space of the user executing the call.

Figure 3 depicts the I/O Subsystem. The Linux kernel implements a device-independent I/O system that serves all devices. A device driver provides the I/O system with a standard interface to the hardware, hiding the unique characteristics of the hardware device from the user to the greatest extent possible.

Listing 1 illustrates a user program that employs some basic system calls to read characters from a device into a buffer. When a system call is requested, the kernel transfers control to the appropriate device driver routine that executes on behalf of the calling user process (as shown previously with Figure 3).

All devices look like files on a Linux system. In fact, the user-level interface to a device is called a special file These special files (often called device nodes) reside in the `/dev` directory. For example, invoking the command `ls -l /dev/lp*` can be used to yield the following status information:

```
crw-rw-rw 1 root root 6, 0 April 23 1994 /dev/lp0
```

This example indicates that: lp0 is a character type device (the first letter of the file mode field is c), th major number is 6, and minor device number 0 is assigned to the device.

Major device numbers are used by the Linux system to map I/O requests to the driver code, thereby deciding which device driver to execute, when a user reads from or writes to the special file. The minor numbers are entirely under the control of the driver writer, and usually refer to sub-devices of the device. These sub-devices may be separate units attached to a controller. Thus, a disk device driver may, for example, communicate with a hardware controller (the device) which has several disk drives (sub-devices) attached.

Figure 4 outlines the flow of execution of a system call within the Linux operating system.

Device Drivers

A device driver is a collection of subroutines and data within the kernel that constitutes the software interface to an I/O device. When the kernel recognizes that a particular action is required from the device, it calls the appropriate driver routine, which passes control from the user process to the driver routine. Control is returned to the user process when the driver routine has completed. A device driver may be shared simultaneously by user applications and must be protected to ensure its own integrity.

Figure 5 shows the relationship between device driver and the Linux system.

A device driver provides the following features:

- A set of routines that communicate with a hardware device and provide a uniform interface to the operating system kernel.
- A self-contained component that can be added to, or removed from, the operating system dynamically.
- Management of data flow and control between user programs and a peripheral device.
- A user-defined section of the kernel that allows a program or a peripheral device to appear as a ``/dev'' device to the rest of the system's software.

Character and Block Device Drivers

Character and block device drivers are the two main types of peripheral drivers. A disk drive is an example of a block device, whereas, terminals and line printers are examples of character devices.

A block device driver is accessed by user programs through a system buffer that acts as a data cache. Specific allocation and memory management routines are not necessary as the system transfers the data to/from the device. Character device drivers communicate directly with the user program, as there is no buffering performed. Linux transfers control to the appropriate device driver when a user program requests a data transfer between a section of its memory and a device. The device driver is responsible for transferring the data. Within Linux, the source for character drivers is kept in the `/usr/src/linux/drivers/char` directory. This article only addresses the development of character device drivers.

Kernel Programming Environment

A Linux user process executes in a space isolated from critical system data and other user processes. This protected environment provides security to protect the process from mistakes in other processes. By contrast, a device driver executes in kernel mode, which places few limits on its freedom of action. The driver is assumed to be correct and responsible. A driver has to be part of the kernel in order to service interrupts and access device hardware. A driver should process interrupts efficiently to preserve the scheduler s ability to balance the demands on the system. It should also use system buffers responsibly to avoid degrading system performance.

A device driver contains both interrupt and synchronous sections. The interrupt section deals with real-time events and is driven by interrupts from devices. The synchronous section, which comprises the remainder of the driver, only executes when the process which it serves is also active. When a device requests some software service, it generates an ‘‘interrupt.’’ The interrupt handler must determine the cause of the interrupt and take appropriate action.

A Linux process might have to wait for an event to occur before it can proceed. For example, a process might wait for requested information to be written to a hardware device before continuing. One way that processes can coordinate their actions with events is through `sleep()` and `wakeup()` system calls. When a process goes to sleep, it specifies an event that must occur, that is, wakeup, before it can continue its task. For example: `interruptible_sleep_on(&dev_wait_queue)` causes the process to sleep and adds the process number to the list of processes sleeping on `dev_wait_queue`. When the device is ready, it posts an interrupt, causing the interrupt service routine in the driver to be activated. The routine services the device and issue a corresponding wakeup call, for example, `wake_up_interruptible(&dev_wait_queue)`, which wakes up the process sleeping on `dev_wait_queue`.

Special care must be taken if two or more processes, such as the synchronous and interrupt portions of a device driver, share common data. The shared data area must be treated as a critical section. The critical section is protected by ensuring that processes only have mutually exclusive access to the shared data. Mutually exclusive access to a critical section can be implemented by using the Linux kernel routines `cli()` and `sti()`. Interrupts are disabled by `cli()` while the process is operating in the critical section and re-enabled by `sti()` upon exit from the critical section, as in:

```
cli()
Critical Section Operations
sti()
```

Virtual File system Switch (VFS)

The principal interface between a device driver and the rest of the Linux kernel comprises a set of standard entry points and driver-specific data structures (see Figure 6).

Listing 2 illustrates how the entry points are registered with the Virtual File system Switch using the `file_operations` structure. This structure, which is defined in `/usr/include/linux/fs.h`, constitutes a list of the functions written for the driver. The initialization routine, `xxx_init()` registers the `file_operations` structure with the VFS and allocates a major number for the device.

Device Driver Development Supporting Functions

The table below contains most of the common supporting functions available for writing device drivers. See also the Kernel Hackers' Guide [John93] for a more detailed explanation:

`add_timer()`
: Causes a function to be executed when a given amount of time has passed

`cli()`
: Prevents interrupts from being acknowledged

`end_request()`
: Called when a request has been satisfied or aborted

`free_irq()`
: Frees an IRQ previously acquired with `request_irq()` or `irqaction()`

`get_fs*()`
: Allows a driver to access data in user space, a memory area distinct from the kernel

`inb(), inb_p()`
: Reads a byte from a port. Here, `inb()` goes as fast as it can, while `inb_p()` pauses before returning.

`irqaction()`
: Registers an interrupt like a signal.

`IS_*(inode)`
: Tests if inode is on a file system mounted with the corresponding flag.

`kfree*()`
: Frees memory previously allocated with `kmalloc()`

`kmalloc()`
: Allocates a chunk of memory no larger than 4096 bytes.

`MAJOR()`
: Reports the major device number for a device.

`MINOR()`
: Reports the minor device number for a device.

`memcpy_*fs()`
: Copies chunks of memory between user space and kernel space

`outb(), outb_p()`
: Writes a byte to a port. Here, `outb()` goes as fast as it can, while `outb_p()` pauses before returning.

`printk()`
: A version of `printf()` for the kernel.

put_fs*()
: Allows a driver to write data in user space.

register_*dev()
: Registers a device with the kernel.

request_irq()
: Requests an IRQ from the kernel, and, if successful, installs an IRQ interrupt handler.

select_wait()
: Adds a process to the proper select_wait queue.

*sleep_on()
: Sleeps on an event, puts a wait_queue entry in the list so that the process can be awakened on that event.

sti()
: Allows interrupts to be acknowledged.

sys_get*()
: System calls used to get information regarding the process, user, or group.

wake_up*()
: Wakes up a process that has been put to sleep by the matching *sleep_on() function.

Name space

The name of the driver should be a short string. Throughout this article we have used "xxx" as our device name. For instance, the parallel (printer) device is the ‘‘lp’’ device, the floppies are the ‘‘fd’’ devices, and the SCSI disks are the ‘‘sd’’ devices. To avoid name space confusion, the entry point names are formed by concatenating this unique driver prefix with a generic name that describes the routine. For instance, xxx_open() is the ‘‘open’’ routine for the ‘‘xxx’’ driver.

Accessing Hardware Memory

A Linux user process can not access physical memory directly. The memory management scheme--which is a demand paged virtual memory system--means that each process has its own address space (user virtual address space) that begins at virtual location zero. The kernel has its own distinct address space known as the system virtual address space.

The device driver copies data between the kernel’ s address space and the user program’ s address space whenever the user makes a read() or write() system call. Several Linux routines--such as, memcpy_*fs() and put_fs*()--enable device drivers to transfer data across the user-system boundary. Data may be transferred in bytes, words, or in buffers of arbitrary sizes. For example, memcpy_fromfs() transfers an arbitrary number of bytes of data from user space to the device, while get_fs_byte() transfers a byte of data from user space. Similarly, memcpy_tofs() and put_fs_byte() write data to user space memory.

The transfer of data between the memory accessible to the kernel and the device itself is machine-dependent. Some machines require that the CPU execute special I/O instructions to move data between a device register and addressable memory--often called direct memory access (DMA). Another scheme, known as memory mapped I/O, implements the device interface as one or more locations in the memory address space. The most common method uses I/O instructions, provided by the system to allow drivers access the data in a general way. Linux provides inb() to read a single byte from an I/O address (port) and outb() to write a single byte to an I/O address. The calling syntax is shown here:

```
unsigned char inb(int port)
outb(char data, int port)
```

Writing a Character Device Driver

Listing 3 shows a sample `xxx_write()` routine where the device driver would, typically, poll the hardware to determine if it is ready to transfer data. The `xxx_write()` routine transfers a character string of *count* bytes from the user-space memory to the device. Using interrupts, the hardware is able to interrupt when it is ready to transfer data and so there is no waiting. Listing 4 outlines an alternative `xxx_write()` routine for an interrupt-driven driver.

Here, `xxx_table[]` is an array of structures, each of which have several members. Some of the members include `xxx_wait_queue` and `bytes_xfered`, which are used for both reading and writing. The interrupt-handling code can use either `request_irq()` or `irqaction()` in the `xxx_open()` routine to call `xxx_interrupt()`.

Listing 5 presents an example of a complete device driver (for the bus mouse). The source listing contains the code for a typical bus mouse driver, such as the Logitec bus mouse or the Microsoft bus mouse.

Device Driver Initialization

In order that the device driver is correctly initialized when the operating system is booted, the `xxx_init()` routine must be executed. To ensure this happens, add the following line to the end of the `chr_drv_init()` function in the `/usr/src/linux/driver/char/mem.c` file:

```
mem_start = xxx_init(mem_start);
```

and resave the file back to disk.

Installing the Driver in the Kernel

A character device driver has to be archived into the `/usr/src/linux/drivers/char/char.a` library. The following steps are required to link the driver to the kernel:

- Put a copy of the source file (say `xxx_drv.c`) in the `/usr/src/linux/drivers/char` directory.
- Edit `Makefile` in the same directory so it will compile the source for the driver--add `xxx_drv.o` to the `OBJS` list, which causes the `make` utility to automatically compile `xxx_drv.c` and add the object code to the `char.a` library archive.
- The last step step is the recompilation of the kernel.

The following steps are required to recompile the Linux kernel:

1. Log in as root
2. Change to the `/root/linux` directory
3. Carry out the following series of commands
    - **`make clean ; make config`** to configures the basic kernel
    - **`make dep`** to set-up the dependencies correctly

- **make** to create the new kernel

4. Wait for the kernel to compile and go to the `/usr/src/linux` directory.
5. In order to boot the new kernel, copy the new kernel image (`/usr/src/linux/zImage`) into the place where the regular bootable kernel is found.

Device File Creation

In order to access the device using system calls, a special file is created. The driver files are normally stored in the `/dev` directory of the system. The following commands create the special device file:

`mknod /dev/xxx c 22 0`
: Creates a special character file named `xxx` and gives it major number 22 and minor number 0.

`chmod 0666 /dev/xxx`
: Ensures that every user in the system has read/write access to the device.

Conclusions

In this article, we have detailed how to write a hardware character device driver for the Linux operating system. We have outlined how to access hardware memory. We have also presented the kernel programming environment, as well as the supporting functions available to write a device driver. A number of worked examples were also presented to aid the programmer in developing his/her own device driver(s).

Bibliography

[Bach86] Bach, M; *The Design of the Unix Operating System*; Englewood Cliffs, NJ: Prentice Hall, 1986.

[John93] Michael K. Johnson; *LINUX Kernel Hackers' Guide*; 201 Howell Street, Apt. 1C, Chapel Hill, North Carolina 27514-4818; 1993.

[Swit93] Robert Switzer, University of Gottingen, Germany; *Operating Systems, A Practical Approach*; Prentice Hall 1993.

[YCI94] The Linux Bible, *The GNU Testament-2nd Edition*; Yggdrasil Computing Incorporated, Version 2.1.1, 10 July 1994.

Edited by Becca Thomas / Online Editor / UnixWorld Online / editor@unixworld.com

Technical review and some commentary kindly provided by Stephen Degler.
Last Modified: Thursday, 25-Jan-96 08:10:16 PST