1 Device Drivers

- Device drivers take on a special role in the Linux kernel. They are distinct "black boxes" that make a particular piece of hardware respond to a well-defined internal programming interface; they hide completely the details of how the device works.
- User activities are performed by means of a set of standardized calls that are independent of the specific driver; mapping those calls to device-specific operations that act on real hardware is then the role of the device driver. This programming interface is such that drivers can be built separately from the rest of the kernel and "plugged in" at runtime when needed.
- Each driver is different; as a driver writer, you need to understand your specific device well. But most of the principles and basic techniques are the same for all drivers.

1.1 The Role of the Device Driver

- The role of a device driver is providing mechanism, not policy. The distinction between mechanism and policy is one of the best ideas behind the Unix design. Most programming problems can indeed be split into two parts:
  - what capabilities are to be provided (the mechanism)
  - how those capabilities can be used (the policy).

  If the two issues are addressed by different parts of the program, or even by different programs altogether, the software package is much easier to develop and to adapt to particular needs.
- For example, Unix management of the graphic display is split between
  - the X server, which knows the hardware and offers a unified interface to user programs,
  - and the window and session managers, which implement a particular policy without knowing anything about the hardware.

  People can use the same window manager on different hardware, and different users can run different configurations on the same workstation. Even completely different desktop environments, such as KDE and GNOME, can coexist on the same system.

- Another example is the layered structure of TCP/IP networking:
  - the operating system offers the socket abstraction, which implements no policy regarding the data to be transferred,
  - while different servers are in charge of the services (and their associated policies).
- When writing drivers, a programmer should pay particular attention to this fundamental concept: *write kernel code to access the hardware, but don't force particular policies on the user, since different users have different needs.*
  - The driver should deal with making the hardware available, leaving all the issues about how to use the hardware to the applications.
  - A driver, then, is flexible if it offers access to the hardware capabilities without adding constraints. Sometimes, however, some policy decisions must be made. For example, a digital I/O driver may only offer byte-wide access to the hardware in order to avoid the extra code needed to handle individual bits.
- Policy-free drivers have a number of typical characteristics.
  - These include support for both synchronous and asynchronous operation,
  - the ability to be opened multiple times,
  - the ability to exploit the full capabilities of the hardware,
  - and the lack of software layers to "simplify things" or provide policy-related operations.

  Drivers of this sort not only work better for their end users, but also turn out to be easier to write and maintain as well. Being policy-free is actually a common target for software designers.

1.2 Splitting the Kernel

- In a Unix system, several concurrent processes attend to different tasks. Each process asks for system resources, be it computing power, memory, network connectivity, or some other resource. The kernel is the big chunk of executable code in charge of handling all such requests. Although the distinction between the different kernel tasks isn't always

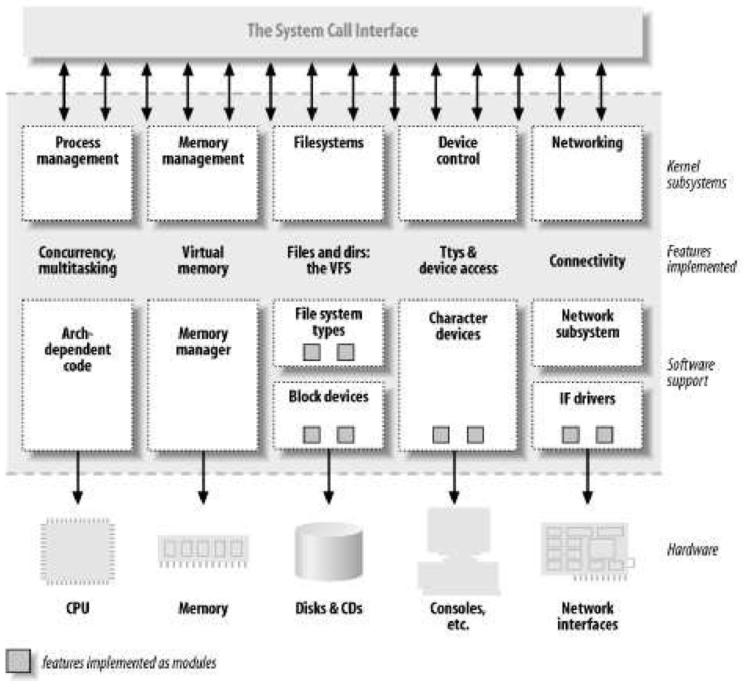

Figure 1: Split view of the kernel.

clearly marked, the kernel's role can be split (as shown in Fig. 1) into the following parts:

1. Process management
   - The kernel is in charge of creating and destroying processes and handling their connection to the outside world (input and output).
   - Communication among different processes (through signals, pipes, or interprocess communication primitives) is basic to the overall system functionality and is also handled by the kernel.
   - In addition, the scheduler, which controls how processes share the CPU, is part of process management. More generally, the kernel's process management activity implements the abstraction of several processes on top of a single CPU or a few of them.
2. Memory management

   - The computer's memory is a major resource, and the policy used to deal with it is a critical one for system performance.
   - The kernel builds up a virtual addressing space for any and all processes on top of the limited available resources.
   - The different parts of the kernel interact with the memory-management subsystem through a set of function calls, ranging from the simple malloc/free pair to much more complex functionalities.

3. Filesystems
   - Unix is heavily based on the filesystem concept; almost everything in Unix can be treated as a file.
   - The kernel builds a structured filesystem on top of unstructured hardware, and the resulting file abstraction is heavily used throughout the whole system.
   - In addition, Linux supports multiple filesystem types, that is, different ways of organizing data on the physical medium. For example, disks may be formatted with the Linux-standard ext3 filesystem, the commonly used FAT filesystem or several others.
4. Device control
   - Almost every system operation eventually maps to a physical device. With the exception of the processor, memory, and a very few other entities, any and all device control operations are performed by code that is specific to the device being addressed.
   - That code is called a *device driver*.
   - The kernel must have embedded in it a device driver for every peripheral present on a system, from the hard drive to the keyboard and the tape drive.
5. Networking
   - Networking must be managed by the operating system, because most network operations are not specific to a process: incoming packets are asynchronous events.
   - The packets must be collected, identified, and dispatched before a process takes care of them.
   - The system is in charge of delivering data packets across program and network interfaces, and it must control the execution of programs according to their network activity. Ad-

ditionally, all the routing and address resolution issues are implemented within the kernel.

1.2.1 Loadable Modules

- You can add functionality to the kernel (and remove functionality as well) while the system is up and running. Each piece of code that can be added to the kernel at runtime is called a module .
- The Linux kernel offers support for quite a few different types (or classes) of modules, including, but not limited to, device drivers. Each module is made up of object code (not linked into a complete executable) that can be dynamically linked to the running kernel by the *insmod* program and can be unlinked by the *rmmod* program.
- Figure 1 identifies different classes of modules in charge of specific tasks (a module is said to belong to a specific class according to the functionality it offers). The placement of modules in Figure 1.5 covers the most important classes, but is far from complete because more and more functionality in Linux is being modularized.

1.3 Classes of Devices and Modules

- The Linux way of looking at devices distinguishes between three fundamental device types. Each module usually implements one of these types, and thus is classifiable as a *char module*, a *block module*, or a *network module*.
- This division of modules into different types, or classes, is not a rigid one; the programmer can choose to build huge modules implementing different drivers in a single chunk of code. Good programmers, nonetheless, usually create a different module for each new functionality they implement, because decomposition is a key element of scalability and extendability.
- There are other ways of classifying driver modules that are orthogonal to the above device types.
  - In general, some types of drivers work with additional layers of kernel support functions for a given type of device. For example, one can talk of universal serial bus (USB) modules, serial modules, SCSI modules, and so on.

  - Every USB device is driven by a USB module that works with the USB subsystem, but the device itself shows up in the system as a char device (a USB serial port, say), a block device (a USB memory card reader), or a network device (a USB Ethernet interface).

- Other classes of device drivers have been added to the kernel in recent times, including FireWire drivers and I2O drivers. In the same way that they handled USB and SCSI drivers, kernel developers collected class-wide features and exported them to driver implementers to avoid duplicating work and bugs, thus simplifying and strengthening the process of writing such drivers.

- In addition to device drivers, other functionalities, both hardware and software, are modularized in the kernel. One common example is filesystems. A filesystem type determines how information is organized on a block device in order to represent a tree of directories and files. Such an entity is not a device driver, in that there's no explicit device associated with the way the information is laid down; the filesystem type is instead a software driver, because it maps the low-level data structures to high-level data structures.

- Kernel programmers should be aware that the development process changed with 2.6. The 2.6 series is now accepting changes that previously would have been considered too large for a "stable" kernel. Among other things, that means that internal kernel programming interfaces can change, some modules don't compile under earlier versions.

Character devices

- A character (char) device is one that can be accessed as a stream of bytes (like a file); a char driver is in charge of implementing this behavior. Such a driver usually implements at least the open, close, read, and write system calls.

- The text console (/dev/console) and the serial ports (/dev/ttyS0 and friends) are examples of char devices, as they are well represented by the stream abstraction. Char devices are accessed by means of filesystem nodes, such as /dev/tty1 and /dev/lp0.

- The only relevant difference between a char device and a regular file is that you can always move back and forth in the regular file, whereas most char devices are just data channels, which you can only access sequentially.

Block devices

- Like char devices, block devices are accessed by filesystem nodes in the /dev directory. A block device is a device (e.g., a disk) that can host a filesystem.
- In most Unix systems, a block device can only handle I/O operations that transfer one or more whole blocks, which are usually 512 bytes (or a larger power of two) bytes in length. Linux, instead, allows the application to read and write a block device like a char device (it permits the transfer of any number of bytes at a time).
- As a result, block and char devices differ only in the way data is managed internally by the kernel, and thus in the kernel/driver software interface. Like a char device, each block device is accessed through a filesystem node, and the difference between them is transparent to the user.

Network interfaces

- A network interface is in charge of sending and receiving data packets, driven by the network subsystem of the kernel, without knowing how individual transactions map to the actual packets being transmitted. Many network connections (especially those using TCP) are stream-oriented, but network devices are, usually, designed around the transmission and receipt of packets. A network driver knows nothing about individual connections; it only handles packets.
- Not being a stream-oriented device, a network interface isn't easily mapped to a node in the filesystem, as /dev/tty1 is. The Unix way to provide access to interfaces is still by assigning a unique name to them (such as eth0), but that name doesn't have a corresponding entry in the filesystem. Communication between the kernel and a network device driver is completely different from that used with char and block drivers. Instead of read and write, the kernel calls functions related to packet transmission.

1.3.1 Building and Running Modules

- Introducing all the essential concepts about modules and kernel programming. We build and run a complete (if relatively useless) module, and look at some of the basic code shared by all modules (only about modules, without referring to any specific device class).

1.4 The Hello World Module

The following code is a complete "hello world" module:

```
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("Dual BSD/GPL");

static int hello_init(void)
{
    printk(KERN_ALERT "Hello, world\n");
    return 0;
}
static void hello_exit(void)
{
    printk(KERN_ALERT "Goodbye, cruel world\n");
}
module_init(hello_init);
module_exit(hello_exit);
```

- This module defines two functions,
  - one to be invoked when the module is loaded into the kernel (*hello_init*)
  - and one for when the module is removed (*hello_exit*).
- The *module_init* and *module_exit* lines use special kernel macros to indicate the role of these two functions. Another special macro (*MODULE_LICENSE*) is used to tell the kernel that this module bears a free license; without such a declaration, the kernel complains when the module is loaded.
- The *printk* function is defined in the Linux kernel and made available to modules; it behaves similarly to the standard C library function *printf*. The kernel needs its own printing function because it runs by itself, without the help of the C library.

- The module can call *printk* because, after *insmod* has loaded it, the module is linked to the kernel and can access the kernel's public symbols (functions and variables).
- The string *KERN_ALERT* is the priority of the message. We've specified a high priority in this module, because a message with the default priority might not show up anywhere useful, depending on the kernel version you are running, the version of the *klogd* daemon, and your configuration.
- You can test the module with the *insmod* and *rmmod* utilities, as shown below. Note that only the superuser can load and unload a module.

```
[root@ozdogan week7]# make
make -C /lib/modules/2.6.19.2/build M=/home/ozdogan/week7 modules
make[1]: Entering directory '/usr/src/kernels/linux-2.6.19.2'
  CC [M]  /home/ozdogan/week7/hello.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/ozdogan/week7/hello.mod.o
  LD [M]  /home/ozdogan/week7/hello.ko
make[1]: Leaving directory '/usr/src/kernels/linux-2.6.19.2'
% su
root# insmod ./hello.ko
Hello, world
root# rmmod hello
Goodbye cruel world
root#
```

- For the above sequence of commands to work, you must have a properly configured and built kernel tree in a place where the makefile is able to find it (/usr/src/linux-2.6.10 in the example shown). According to the mechanism your system uses to deliver the message lines, your output may be different. In particular, the previous screen dump was taken from a text console; if you are running *insmod* and *rmmod* from a terminal emulator running under the window system, you won't see anything on your screen. The message goes to one of the system log files, such as /var/log/messages (the name of the actual file varies between Linux distributions).

1.5 Kernel Modules Versus Applications

- While most small and medium-sized applications perform a single task from beginning to end, every kernel module just registers itself in order to serve future requests, and its initialization function terminates immediately. In other words, the task of the module's initialization function is to prepare for later invocation of the module's functions; it's as though the module were saying, "Here I am, and this is what I can do."

- The module's exit function (*hello_exit* in the example) gets invoked just before the module is unloaded. It should tell the kernel, "I'm not there anymore; don't ask me to do anything else." This kind of approach to programming is similar to event-driven programming, but while not all applications are event-driven, each and every kernel module is.

- Another major difference between event-driven applications and kernel code is in the exit function: whereas an application that terminates can be lazy in releasing resources or avoids clean up altogether, the exit function of a module must carefully undo everything the init function built up, or the pieces remain around until the system is rebooted.

- As a programmer, you know that an application can call functions it doesn't define: the linking stage resolves external references using the appropriate library of functions. $printf$ is one of those callable functions and is defined in *libc*.

- A module, on the other hand, is linked only to the kernel, and the only functions it can call are the ones exported by the kernel; there are no libraries to link to. The *printk* function used in *hello.c* earlier, for example, is the version of $printf$ defined within the kernel and exported to modules. It behaves similarly to the original function, with a few minor differences, the main one being lack of floating-point support.

- Figure 1.5 shows how function calls and function pointers are used in a module to add new functionality to a running kernel.

- Because no library is linked to modules, source files should never include the usual header files, $< stdarg.h >$ and very special situations being the only exceptions. Only functions that are actually part of the kernel itself may be used in kernel modules. Anything related to the kernel is declared in headers found in the kernel source tree you have set up and configured; most of the relevant headers live in include/linux and

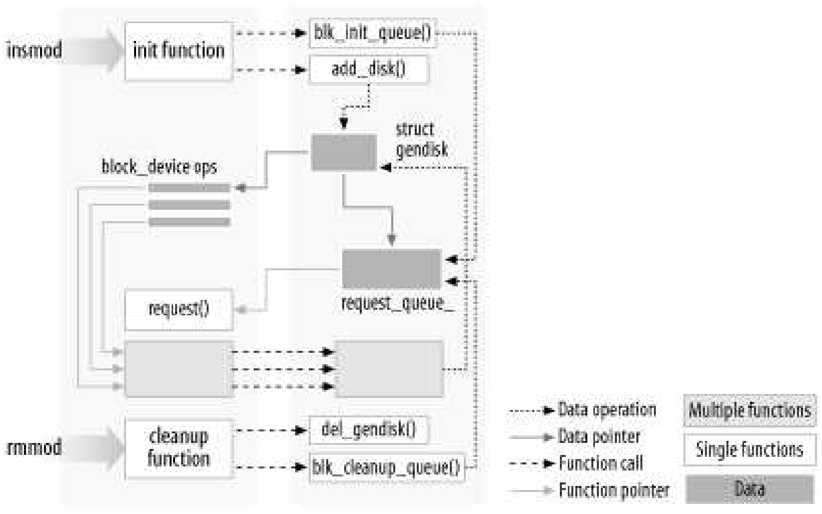

Figure 2: Linking a module to the kernel.

include/asm, but other subdirectories of include have been added to host material associated to specific kernel subsystems.

- Another important difference between kernel programming and application programming is in how each environment handles faults: whereas a segmentation fault is harmless during application development and a debugger can always be used to trace the error to the problem in the source code, a kernel fault kills the current process at least, if not the whole system.

1.5.1 Concurrency in the Kernel

- One way in which kernel programming differs greatly from conventional application programming is the issue of concurrency. Most applications, with the notable exception of multithreading applications, typically run sequentially, from the beginning to the end, without any need to worry about what else might be happening to change their environment.

- Kernel code does not run in such a simple world, and even the simplest kernel modules must be written with the idea that many things can be happening at once.

- There are a few sources of concurrency in kernel programming.
  - Naturally, Linux systems run multiple processes, more than one of which can be trying to use your driver at the same time.
  - Most devices are capable of interrupting the processor; interrupt handlers run asynchronously and can be invoked at the same time that your driver is trying to do something else.
  - Several software abstractions (such as kernel timers) run asynchronously as well.
  - Moreover, of course, Linux can run on symmetric multiprocessor (SMP) systems, with the result that your driver could be executing concurrently on more than one CPU.
  - Finally, in 2.6, kernel code has been made preemptible; this change causes even uniprocessor systems to have many of the same concurrency issues as multiprocessor systems.
- As a result, Linux kernel code, including driver code, must be *reentrant* (it must be capable of running in more than one context at the same time). Data structures must be carefully designed to keep multiple threads of execution separate, and the code must take care to access shared data in ways that prevent corruption of the data.
- Writing code that handles concurrency and avoids race conditions requires thought and can be tricky.

1.5.2 A Few Other Details

- Applications are laid out in virtual memory with a very large stack area. The stack, of course, is used to hold the function call history and all automatic variables created by currently active functions.
- The kernel, instead, has a very small stack; it can be as small as a single, 4096-byte page. Your functions must share that stack with the entire kernel-space call chain. Thus, it is never a good idea to declare large automatic variables; if you need larger structures, you should allocate them dynamically at call time.
- Often, as you look at the kernel API, you will encounter function names starting with a double underscore (__). Functions so marked are generally a low-level component of the interface and should be used with caution. Essentially, the double underscore says to the programmer: "If you call this function, be sure you know what you are doing."

- Kernel code cannot do floating point arithmetic. Enabling floating point would require that the kernel save and restore the floating point processor's state on each entry to, and exit from, kernel space (at least, on some architectures). Given that there really is no need for floating point in kernel code, the extra overhead is not worthwhile.

1.6 Compiling and Loading

Compiling Modules

- If you do not have a kernel tree handy, or have not yet configured and built that kernel, now is the time to go do it. You cannot build loadable modules for a 2.6 kernel without this tree on your filesystem. It is also helpful (though not required) to be actually running the kernel that you are building for. Once you have everything set up, creating a makefile for your module is straightforward. In fact, for the "hello world" example shown earlier in this chapter, a single line will suffice:

  ```
  obj-m := hello.o
  ```

  The assignment above states that there is one module to be built from the object file hello.o. The resulting module is named hello.kO after being built from the object file. If, instead, you have a module called module.ko that is generated from two source files (called, say, file1.c and file2.c), the correct incantation would be:

  ```
  obj-m := module.o
  module-objs := file1.o file2.o
  ```

  Makefile:

  ```
  # If KERNELRELEASE is defined, we've been invoked from the
  # kernel build system and can use its language.
          obj-m := hello.o

  ifneq ($(KERNELRELEASE),)
          obj-m := hello.o

  # Otherwise we were called directly from the command
  # line; invoke the kernel build system.
  else
  ```

```
        KERNELDIR ?= /lib/modules/$(shell uname -r)/build
        PWD  := $(shell pwd)
default:
        $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
endif
```

1.6.1 Loading and Unloading Modules

- After the module is built, the next step is loading it into the kernel. The "insmod" program loads the module code and data into the kernel, which, in turn, performs a function similar to that of *ld*, in that it links any unresolved symbol in the module to the symbol table of the kernel. Unlike the linker, however, the kernel doesn't modify the module's disk file, but rather an in-memory copy.

- How the kernel supports insmod: it relies on a system call defined in kernel/module.c. The function *sys_init_module* allocates kernel memory to hold a module (this memory is allocated with vmalloc); it then copies the module text into that memory region, resolves kernel references in the module via the kernel symbol table, and calls the module's initialization function to get everything going.

- If you actually look in the kernel source, you'll find that the names of the system calls are prefixed with *sys_*. This is true for all system calls and no other functions; it's useful to keep this in mind when grepping for the system calls in the sources.

- The "modprobe" utility is worth a quick mention. *modprobe*, like *insmod*, loads a module into the kernel. It differs in that it will look at the module to be loaded to see whether it references any symbols that are not currently defined in the kernel. If any such references are found, *modprobe* looks for other modules in the current module search path that define the relevant symbols. When *modprobe* finds those modules (which are needed by the module being loaded), it loads them into the kernel as well. If you use *insmod* in this situation instead, the command fails with an "unresolved symbols" message left in the system logfile.

- Modules may be removed from the kernel with the "rmmod" utility. Note that module removal fails if the kernel believes that the module is still in use (e.g., a program still has an open file for a device exported by the modules), or if the kernel has been configured to disallow module removal. It is possible to configure the kernel to allow "forced" removal of modules, even when they appear to be busy.

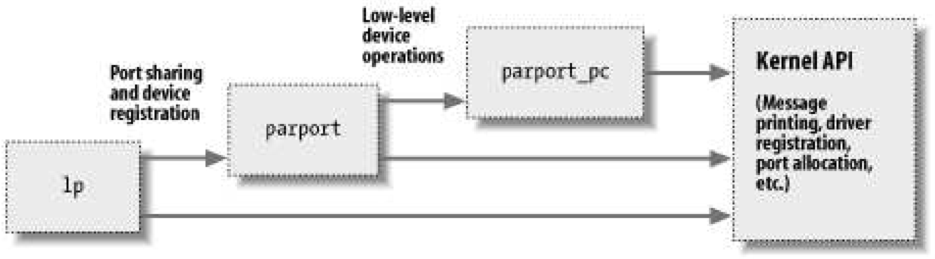
Low-level device operations
Port sharing and device registration
parport_pc
parport
lp
Kernel API
(Message printing, driver registration, port allocation, etc.)

Figure 3: Stacking of parallel port driver modules.

- The “lsmod” program produces a list of the modules currently loaded in the kernel. Some other information, such as any other modules making use of a specific module, is also provided. *lsmod* works by reading the /proc/modules virtual file. Information on currently loaded modules can also be found in the *sysfs* virtual filesystem under /sys/module.

1.7 The Kernel Symbol Table

- We’ve seen how *insmod* resolves undefined symbols against the table of public kernel symbols. The table contains the addresses of global kernel items (functions and variables) that are needed to implement modularized drivers.

- When a module is loaded, any symbol exported by the module becomes part of the kernel symbol table. In the usual case, a module implements its own functionality without the need to export any symbols at all. You need to export symbols, however, whenever other modules may benefit from using them.

- New modules can use symbols exported by your module, and you can stack new modules on top of other modules. Module stacking is implemented in the mainstream kernel sources as well: the *msdos* filesystem relies on symbols exported by the *fat* module, and each input USB device module stacks on the *usbcore* and *input* modules.

- Stacking in the parallel port subsystem is shown in Figure 1.7; the arrows show the communications between the modules and with the kernel programming interface.

- The Linux kernel header files provide a convenient way to manage the visibility of your symbols, thus reducing namespace pollution (filling the namespace with names that may conflict with those defined elsewhere in the kernel) and promoting proper information hiding. If your module needs to export symbols for other modules to use, the following macros should be used.

  ```
  EXPORT_SYMBOL(name);
  EXPORT_SYMBOL_GPL(name);
  ```