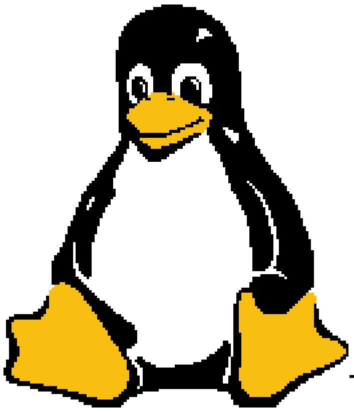

The Linux Kernel

David A Rusling
david.rusling@digital.com

REVIEW, Version 0.8-3

January 25, 1999

This book is for Linux enthusiasts who want to know how the Linux kernel works. It is not an internals manual. Rather it describes the principles and mechanisms that Linux uses; how and why the Linux kernel works the way that it does. Linux is a moving target; this book is based upon the current, stable, 2.0.33 sources as those are what most individuals and companies are now using.

For Gill, Esther and Stephen

Legal Notice

3 Foxglove Close, Wokingham, Berkshire RG41 3NF, UK
david.rusling@arm.com

Preface

Linux is a phenomenon of the Internet. Born out of the hobby project of a student it has grown to become more popular than any other freely available operating system. To many Linux is an enigma. How can something that is free be worthwhile? In a world dominated by a handful of large software corporations, how can something that has been written by a bunch of "hackers" (sic) hope to compete? How can software contributed to by many different people in many different countries around the world have a hope of being stable and effective? Yet stable and effective it is and compete it does. Many Universities and research establishments use it for their everyday computing needs. People are running it on their home PCs and I would wager that most companies are using it somewhere even if they do not always realize that they do. Linux is used to browse the web, host web sites, write theses, send electronic mail and, as always with computers, to play games. Linux is emphatically not a toy; it is a fully developed and professionally written operating system used by enthusiasts all over the world.

The roots of Linux can be traced back to the origins of Unix™ . In 1969, Ken Thompson of the Research Group at Bell Laboratories began experimenting on a multi-user, multi-tasking operating system using an otherwise idle PDP-7. He was soon joined by Dennis Richie and the two of them, along with other members of the Research Group produced the early versions of Unix™. Richie was strongly influenced by an earlier project, MULTICS and the name Unix™ is itself a pun on the name MULTICS. Early versions were written in assembly code, but the third version was rewritten in a new programming language, C. C was designed and written by Richie expressly as a programming language for writing operating systems. This rewrite allowed Unix™ to move onto the more powerful PDP-11/45 and 11/70 computers then being produced by DIGITAL. The rest, as they say, is history. Unix™ moved out of the laboratory and into mainstream computing and soon most major computer manufacturers were producing their own versions.

Linux was the solution to a simple need. The only software that Linus Torvalds, Linux's author and principle maintainer was able to afford was `Minix`. `Minix` is a simple, Unix™ like, operating system widely used as a teaching aid. Linus was less than impressed with its features, his solution was to write his own software. He took Unix™ as his model as that was an operating system that he was familiar with in his day to day student life. He started with an Intel 386 based PC and started to write. Progress was rapid and, excited by this, Linus offered his efforts to other students via the emerging world wide computer networks, then mainly used by the academic community. Others saw the software and started contributing. Much of this new software was itself the solution to a problem that one of the contributors had. Before long, Linux had become an operating system. It is important to note that Linux contains no Unix™ code, it is a rewrite based on published POSIX standards. Linux is built with and uses a lot of the GNU (GNU's Not Unix™) software produced by the Free Software Foundation in Cambridge, Massachusetts.

Most people use Linux as a simple tool, often just installing one of the many good CD ROM-based distributions. A lot of Linux users use it to write applications or to run applications written by others. Many Linux users read the HOWTOs[1] avidly and feel both the thrill of success when some part of the system has been correctly configured and the frustration of failure when it has not. A minority are bold enough to write device drivers and offer kernel patches to Linus Torvalds, the creator and maintainer of the Linux kernel. Linus accepts additions and modifications to the kernel sources from anyone, anywhere. This might sound like a recipe for anarchy but Linus exercises strict quality control and merges all new code into the kernel himself. At any one time though, there are only a handful of people contributing sources to the Linux kernel.

The majority of Linux users do not look at how the operating system works, how it fits together. This is a shame because looking at Linux is a very good way to learn more about how an operating system functions. Not only is it well written, all the sources are freely available for you to look at. This is because although the authors retain the copyrights to their software, they allow the sources to be freely redistributable under the Free Software Foundation's GNU Public License. At first glance though, the sources can be confusing; you will see directories called `kernel`, `mm` and `net` but what do they contain and how does that code work? What is needed is a broader understanding of the overall structure and aims of Linux. This, in short, is the aim of this book: to promote a clear understanding of how Linux, the operating system, works. To provide a mind model that allows you to picture what is happening within the system as you copy a file from one place to another or read electronic mail. I well remember the excitement that I felt when I first realized just how an operating system actually worked. It is that excitement that I want to pass on to the readers of this book.

My involvement with Linux started late in 1994 when I visited Jim Paradis who was working on a port of Linux to the Alpha AXP processor based systems. I had worked for Digital Equipment Co. Limited since 1984, mostly in networks and communications and in 1992 I started working for the newly formed Digital Semiconductor division. This division's goal was to enter fully into the merchant chip vendor market and sell chips, and in particular the Alpha AXP range of microprocessors but also Alpha AXP system boards outside of Digital. When I first heard about Linux I immediately saw an opportunity to have fun. Jim's enthusiasm was catching and I started to help on the port. As I worked on this, I began more and more to appreciate not only the operating system but also the community of engineers that produces it.

However, Alpha AXP is only one of the many hardware platforms that Linux runs on. Most Linux kernels are running on Intel processor based systems but a growing number of non-Intel Linux systems are becoming more commonly available. Amongst these are Alpha AXP, ARM, MIPS, Sparc and PowerPC. I could have written this book using any one of those platforms but my background and technical experiences with Linux are with Linux on the Alpha AXP and, to a lesser extent on the ARM. This is why this book sometimes uses non-Intel hardware as an example to illustrate

[1] A HOWTO is just what it sounds like, a document describing how to do something. Many have been written for Linux and all are very useful.

some key point. It must be noted that around 95% of the Linux kernel sources are common to all of the hardware platforms that it runs on. Likewise, around 95% of this book is about the machine independent parts of the Linux kernel.

Reader Profile

This book does not make any assumptions about the knowledge or experience of the reader. I believe that interest in the subject matter will encourage a process of self education where neccessary. That said, a degree of familiarity with computers, preferably the PC will help the reader derive real benefit from the material, as will some knowledge of the C programming language.

Organisation of this Book

This book is *not* intended to be used as an internals manual for Linux. Instead it is an introduction to operating systems in general and to Linux in particular. The chapters each follow my rule of "working from the general to the particular". They first give an overview of the kernel subsystem that they are describing before launching into its gory details.

I have deliberately not described the kernel's algorithms, its methods of doing things, in terms of `routine_X()` calls `routine_Y()` which increments the `foo` field of the `bar` data structure. You can read the code to find these things out. Whenever I need to understand a piece of code or describe it to someone else I often start with drawing its data structures on the white-board. So, I have described many of the relevant kernel data structures and their interrelationships in a fair amount of detail.

Each chapter is fairly independent, like the Linux kernel subsystem that they each describe. Sometimes, though, there are linkages; for example you cannot describe a process without understanding how virtual memory works.

The Hardware Basics chapter (Chapter 1) gives a brief introduction to the modern PC. An operating system has to work closely with the hardware system that acts as its foundations. The operating system needs certain services that can only be provided by the hardware. In order to fully understand the Linux operating system, you need to understand the basics of the underlying hardware.

The Software Basics chapter (Chapter 2) introduces basic software principles and looks at assembly and C programing languages. It looks at the tools that are used to build an operating system like Linux and it gives an overview of the aims and functions of an operating system.

The Memory Management chapter (Chapter 3) describes the way that Linux handles the physical and virtual memory in the system.

The Processes chapter (Chapter 4) describes what a process is and how the Linux kernel creates, manages and deletes the processes in the system.

Processes communicate with each other and with the kernel to coordinate their activities. Linux supports a number of Inter-Process Communication (IPC) mechanisms. Signals and pipes are two of them but Linux also supports the System V IPC mechanisms named after the Unix™ release in which they first appeared. These interprocess communications mechanisms are described in Chapter 5.

The Peripheral Component Interconnect (PCI) standard is now firmly established as the low cost, high performance data bus for PCs. The PCI chapter (Chapter 6) describes how the Linux kernel initializes and uses PCI buses and devices in the system.

The Interrupts and Interrupt Handling chapter (Chapter 7) looks at how the Linux kernel handles interrupts. Whilst the kernel has generic mechanisms and interfaces for handling interrupts, some of the interrupt handling details are hardware and architecture specific.

One of Linux's strengths is its support for the many available hardware devices for the modern PC. The Device Drivers chapter (Chapter 8) describes how the Linux kernel controls the physical devices in the system.

The File system chapter (Chapter 9) describes how the Linux kernel maintains the files in the file systems that it supports. It describes the Virtual File System (VFS) and how the Linux kernel's real file systems are supported.

Networking and Linux are terms that are almost synonymous. In a very real sense Linux is a product of the Internet or World Wide Web (WWW). Its developers and users use the web to exchange information ideas, code and Linux itself is often used to support the networking needs of organizations. Chapter 10 describes how Linux supports the network protocols known collectively as TCP/IP.

The Kernel Mechanisms chapter (Chapter 11) looks at some of the general tasks and mechanisms that the Linux kernel needs to supply so that other parts of the kernel work effectively together.

The Modules chapter (Chapter 12) describes how the Linux kernel can dynamically load functions, for example file systems, only when they are needed.

The Processors chapter (Chapter 13) gives a brief description of some of the processors that Linux has been ported to.

The Sources chapter (Chapter 14) describes where in the Linux kernel sources you should start looking for particular kernel functions.

Conventions used in this Book

The following is a list of the typographical conventions used in this book.

serif font	identifies commands or other text that is to be typed literally by the user.
`type font`	refers to data structures or fields within data structures.

See `foo()` in `foo/bar.c`

Throughout the text there references to pieces of code within the Linux kernel source tree (for example the boxed margin note adjacent to this text). These are given in case you wish to look at the source code itself and all of the file references are relative to `/usr/src/linux`. Taking `foo/bar.c` as an example, the full filename would be `/usr/src/linux/foo/bar.c` If you are running Linux (and you should), then looking at the code is a worthwhile experience and you can use this book as an aid to understanding the code and as a guide to its many data structures.

Trademarks

The Author

I was born in 1957, a few weeks before Sputnik was launched, in the north of England. I first met Unix at University, where a lecturer used it as an example when teaching the notions of kernels, scheduling and other operating systems goodies. I loved using the newly delivered PDP-11 for my final year project. After graduating (in 1982 with a First Class Honours degree in Computer Science) I worked for Prime Computers (Primos) and then after a couple of years for Digital (VMS, Ultrix). At Digital I worked on many things but for the last 5 years there, I worked for the semiconductor group on Alpha and StrongARM evaluation boards. In 1998 I moved to ARM where I have a small group of engineers writing low level firmware and porting operating systems. My children (Esther and Stephen) describe me as a geek.

People often ask me about Linux at work and at home and I am only too happy to oblige. The more that I use Linux in both my professional and personal life the more that I become a Linux zealot. You may note that I use the term 'zealot' and not 'bigot'; I define a Linux zealot to be an enthusiast that recognizes that there are other operating systems but prefers not to use them. As my wife, Gill, who uses Windows 95 once remarked "I never realized that we would have his and her operating systems". For me, as an engineer, Linux suits my needs perfectly. It is a superb, flexible and adaptable engineering tool that I use at work and at home. Most freely available software easily builds on Linux and I can often simply download pre-built executable files or install them from a CD ROM. What else could I use to learn to program in C++, Perl or learn about Java for free?

Acknowledgements

I must thank the many people who have been kind enough to take the time to e-mail me with comments about this book. I have attempted to incorporated those

comments in each new version that I have produced and I am more than happy to receive comments, however please note my new e-mail address.

A number of lecturers have written to me asking if they can use some or parts of this book in order to teach computing. My answer is an emphatic yes; this is one use of the book that I particularly wanted. Who knows, there may be another Linus Torvalds sat in the class.

Special thanks must go to John Rigby and Michael Bauer who gave me full, detailed review notes of the whole book. Not an easy task. Alan Cox and Stephen Tweedie have patiently answered my questions - thanks. I used Larry Ewing's penguins to brighten up the chapters a bit. Finally, thank you to Greg Hankins for accepting this book into the Linux Documentation Project and onto their web site.

Contents

List of Figures

Chapter 1

Hardware Basics

An operating system has to work closely with the hardware system that acts as its foundations. The operating system needs certain services that can only be provided by the hardware. In order to fully understand the Linux operating system, you need to understand the basics of the underlying hardware. This chapter gives a brief introduction to that hardware: the modern PC.

When the "Popular Electronics" magazine for January 1975 was printed with an illustration of the Altair 8080 on its front cover, a revolution started. The Altair 8080, named after the destination of an early Star Trek episode, could be assembled by home electronics enthusiasts for a mere $397. With its Intel 8080 processor and 256 bytes of memory but no screen or keyboard it was puny by today's standards. Its inventor, Ed Roberts, coined the term "personal computer" to describe his new invention, but the term PC is now used to refer to almost any computer that you can pick up without needing help. By this definition, even some of the very powerful Alpha AXP systems are PCs.

Enthusiastic hackers saw the Altair's potential and started to write software and build hardware for it. To these early pioneers it represented freedom; the freedom from huge batch processing mainframe systems run and guarded by an elite priesthood. Overnight fortunes were made by college dropouts fascinated by this new phenomenon, a computer that you could have at home on your kitchen table. A lot of hardware appeared, all different to some degree and software hackers were happy to write software for these new machines. Paradoxically it was IBM who firmly cast the mould of the modern PC by announcing the IBM PC in 1981 and shipping it to customers early in 1982. With its Intel 8088 processor, 64K of memory (expandable to 256K), two floppy disks and an 80 character by 25 lines Colour Graphics Adapter (CGA) it was not very powerful by today's standards but it sold well. It was followed, in 1983, by the IBM PC-XT which had the luxury of a 10Mbyte hard drive. It was not long before IBM PC clones were being produced by a host of companies such as Compaq and the architecture of the PC became a de-facto standard. This

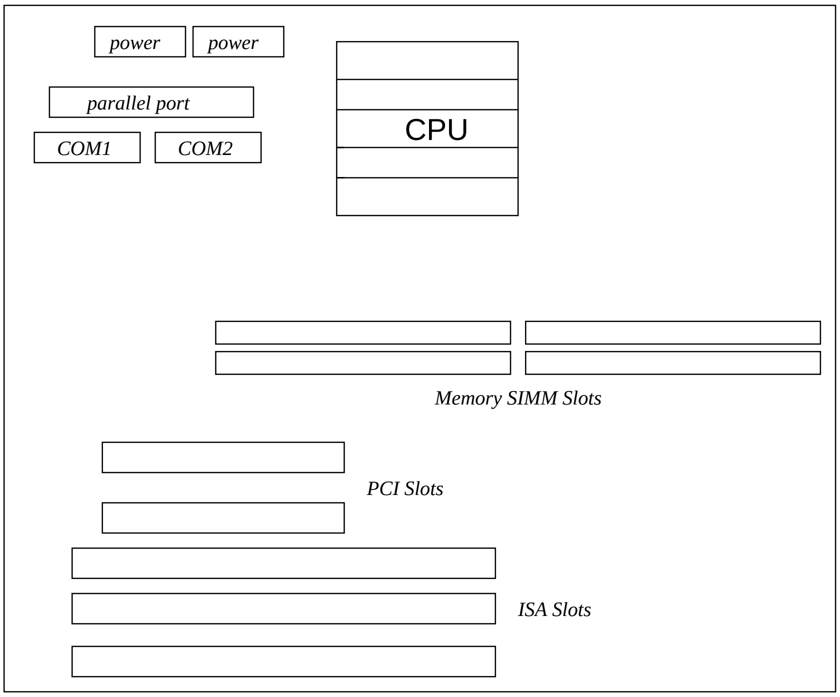

Figure 1.1: A typical PC motherboard.

de-facto standard helped a multitude of hardware companies to compete together in a growing market which, happily for consumers, kept prices low. Many of the system architectural features of these early PCs have carried over into the modern PC. For example, even the most powerful Intel Pentium Pro based system starts running in the Intel 8086's addressing mode. When Linus Torvalds started writing what was to become Linux, he picked the most plentiful and reasonably priced hardware, an Intel 80386 PC.

Looking at a PC from the outside, the most obvious components are a system box, a keyboard, a mouse and a video monitor. On the front of the system box are some buttons, a little display showing some numbers and a floppy drive. Most systems these days have a CD ROM and if you feel that you have to protect your data, then there will also be a tape drive for backups. These devices are collectively known as the peripherals.

Although the CPU is in overall control of the system, it is not the only intelligent device. All of the peripheral controllers, for example the IDE controller, have some level of intelligence. Inside the PC (Figure 1.1) you will see a motherboard containing the CPU or microprocessor, the memory and a number of slots for the ISA or PCI peripheral controllers. Some of the controllers, for example the IDE disk controller may be built directly onto the system board.

1.1 The CPU

The CPU, or rather microprocessor, is the heart of any computer system. The microprocessor calculates, performs logical operations and manages data flows by reading instructions from memory and then executing them. In the early days of computing the functional components of the microprocessor were separate (and physically large) units. This is when the term *Central Processing Unit* was coined. The modern microprocessor combines these components onto an integrated circuit etched onto a very small piece of silicon. The terms *CPU*, *microprocessor* and *processor* are all used interchangeably in this book.

Microprocessors operate on binary data; that is data composed of ones and zeros. These ones and zeros correspond to electrical switches being either on or off. Just as 42 is a decimal number meaning "4 10s and 2 units", a binary number is a series of binary digits each one representing a power of 2. In this context, a power means the number of times that a number is multiplied by itself. 10 to the power 1 (10^1) is 10, 10 to the power 2 (10^2) is 10x10, 10^3 is 10x10x10 and so on. Binary 0001 is decimal 1, binary 0010 is decimal 2, binary 0011 is 3, binary 0100 is 4 and so on. So, 42 decimal is 101010 binary or ($2+8+32$ or $2^1+2^3+2^5$). Rather than using binary to represent numbers in computer programs, another base, hexadecimal is usually used. In this base, each digital represents a power of 16. As decimal numbers only go from 0 to 9 the numbers 10 to 15 are represented as a single digit by the letters A, B, C, D, E and F. For example, hexadecimal E is decimal 14 and hexadecimal 2A is decimal 42 (two 16s) + 10). Using the C programming language notation (as I do throughout this book) hexadecimal numbers are prefaced by "*0x*"; hexadecimal 2A is written as *0x2A* .

Microprocessors can perform arithmetic operations such as add, multiply and divide and logical operations such as "is X greater than Y?".

The processor's execution is governed by an external clock. This clock, the system clock, generates regular clock pulses to the processor and, at each clock pulse, the processor does some work. For example, a processor could execute an instruction every clock pulse. A processor's speed is described in terms of the rate of the system clock ticks. A 100Mhz processor will receive 100,000,000 clock ticks every second. It is misleading to describe the power of a CPU by its clock rate as different processors perform different amounts of work per clock tick. However, all things being equal, a faster clock speed means a more powerful processor. The instructions executed by the processor are very simple; for example "read the contents of memory at location X into register Y". Registers are the microprocessor's internal storage, used for storing data and performing operations on it. The operations performed may cause the processor to stop what it is doing and jump to another instruction somewhere else in memory. These tiny building blocks give the modern microprocessor almost limitless power as it can execute millions or even billions of instructions a second.

The instructions have to be fetched from memory as they are executed. Instructions may themselves reference data within memory and that data must be fetched from memory and saved there when appropriate.

The size, number and type of register within a microprocessor is entirely dependent on its type. An Intel 4086 processor has a different register set to an Alpha AXP processor; for a start, the Intel's are 32 bits wide and the Alpha AXP's are 64 bits wide. In general, though, any given processor will have a number of general purpose registers and a smaller number of dedicated registers. Most processors have the following special purpose, dedicated, registers:

Program Counter (PC) This register contains the address of the next instruction to be executed. The contents of the PC are automatically incremented each time an instruction is fetched,

Stack Pointer (SP) Processors have to have access to large amounts of external read/write random access memory (RAM) which facilitates temporary storage of data. The stack is a way of easily saving and restoring temporary values in external memory. Usually, processors have special instructions which allow you to push values onto the stack and to pop them off again later. The stack works on a last in first out (LIFO) basis. In other words, if you push two values, x and y, onto a stack and then pop a value off of the stack then you will get back the value of y.

Some processor's stacks grow upwards towards the top of memory whilst others grow downwards towards the bottom, or base, of memory. Some processor's support both types, for example ARM.

Processor Status (PS) Instructions may yield results; for example "is the content of register X greater than the content of register Y?" will yield true or false as a result. The PS register holds this and other information about the current state of the processor. For example, most processors have at least two modes of operation, kernel (or supervisor) and user. The PS register would hold information identifying the current mode.

1.2 Memory

All systems have a memory hierarchy with memory at different speeds and sizes at different points in the hierarchy. The fastest memory is known as cache memory and is what it sounds like - memory that is used to temporarily hold, or cache, contents of the main memory. This sort of memory is very fast but expensive, therefore most processors have a small amount of on-chip cache memory and more system based (on-board) cache memory. Some processors have one cache to contain both instructions and data, but others have two, one for instructions and the other for data. The Alpha AXP processor has two internal memory caches; one for data (the D-Cache) and one for instructions (the I-Cache). The external cache (or B-Cache) mixes the two together. Finally there is the main memory which relative to the external cache memory is very slow. Relative to the on-CPU cache, main memory is positively crawling.

The cache and main memories must be kept in step (coherent). In other words, if a word of main memory is held in one or more locations in cache, then the system must make sure that the contents of cache and memory are the same. The job of cache coherency is done partially by the hardware and partially by the operating system. This is also true for a number of major system tasks where the hardware and software must cooperate closely to achieve their aims.

1.3 Buses

The individual components of the system board are interconnected by multiple connection systems known as buses. The system bus is divided into three logical functions; the address bus, the data bus and the control bus. The address bus specifies the memory locations (addresses) for the data transfers. The data bus holds the data transfered. The data bus is bidirectional; it allows data to be read into the CPU and written from the CPU. The control bus contains various lines used to route timing and control signals throughout the system. Many flavours of bus exist, for example ISA and PCI buses are popular ways of connecting peripherals to the system.

1.4 Controllers and Peripherals

Peripherals are real devices, such as graphics cards or disks controlled by controller chips on the system board or on cards plugged into it. The IDE disks are controlled by the IDE controller chip and the SCSI disks by the SCSI disk controller chips and so on. These controllers are connected to the CPU and to each other by a variety of buses. Most systems built now use PCI and ISA buses to connect together the main system components. The controllers are processors like the CPU itself, they can be viewed as intelligent helpers to the CPU. The CPU is in overall control of the system.

All controllers are different, but they usually have registers which control them. Software running on the CPU must be able to read and write those controlling registers. One register might contain status describing an error. Another might be used for control purposes; changing the mode of the controller. Each controller on a bus can be individually addressed by the CPU, this is so that the software device driver can write to its registers and thus control it. The IDE ribbon is a good example, as it gives you the ability to access each drive on the bus separately. Another good example is the PCI bus which allows each device (for example a graphics card) to be accessed independently.

1.5 Address Spaces

The system bus connects the CPU with the main memory and is separate from the buses connecting the CPU with the system's hardware peripherals. Collectively the memory space that the hardware peripherals exist in is known as I/O space. I/O space may itself be further subdivided, but we will not worry too much about that for the moment. The CPU can access both the system space memory and the I/O space memory, whereas the controllers themselves can only access system memory indirectly and then only with the help of the CPU. From the point of view of the device, say the floppy disk controller, it will see only the address space that its control registers are in (ISA), and not the system memory. Typically a CPU will have separate instructions for accessing the memory and I/O space. For example, there might be an instruction that means "read a byte from I/O address *0x3f0* into register X". This is exactly how the CPU controls the system's hardware peripherals, by reading and writing to their registers in I/O space. Where in I/O space the common peripherals (IDE controller, serial port, floppy disk controller and so on) have their registers has been set by convention over the years as the PC architecture has developed. The I/O space address *0x3f0* just happens to be the address of one of the serial port's (COM1) control registers.

There are times when controllers need to read or write large amounts of data directly to or from system memory. For example when user data is being written to the hard disk. In this case, Direct Memory Access (DMA) controllers are used to allow hardware peripherals to directly access system memory but this access is under strict control and supervision of the CPU.

1.6 Timers

All operating systems need to know the time and so the modern PC includes a special peripheral called the Real Time Clock (RTC). This provides two things: a reliable time of day and an accurate timing interval. The RTC has its own battery so that it continues to run even when the PC is not powered on, this is how your PC always "knows" the correct date and time. The interval timer allows the operating system to accurately schedule essential work.

Chapter 2

Software Basics

A program is a set of computer instructions that perform a particular task. That program can be written in assembler, a very low level computer language, or in a high level, machine independent language such as the C programming language. An operating system is a special program which allows the user to run applications such as spreadsheets and word processors. This chapter introduces basic programming principles and gives an overview of the aims and functions of an operating system.

2.1 Computer Languages

2.1.1 Assembly Languages

The instructions that a CPU fetches from memory and executes are not at all understandable to human beings. They are machine codes which tell the computer precisely what to do. The hexadecimal number *0x89E5* is an Intel 80486 instruction which copies the contents of the ESP register to the EBP register. One of the first software tools invented for the earliest computers was an assembler, a program which takes a human readable source file and assembles it into machine code. Assembly languages explicitly handle registers and operations on data and they are specific to a particular microprocessor. The assembly language for an Intel X86 microprocessor is very different to the assembly language for an Alpha AXP microprocessor. The following Alpha AXP assembly code shows the sort of operations that a program can perform:

```
    ldr r16, (r15)     ; Line 1
    ldr r17, 4(r15)    ; Line 2
    beq r16,r17,100    ; Line 3
    str r17, (r15)     ; Line 4
100:                   ; Line 5
```

The first statement (on line 1) loads register 16 from the address held in register 15. The next instruction loads register 17 from the next location in memory. Line 3 compares the contents of register 16 with that of register 17 and, if they are equal, branches to label *100*. If the registers do not contain the same value then the program continues to line 4 where the contents of r17 are saved into memory. If the registers do contain the same value then no data needs to be saved. Assembly level programs are tedious and tricky to write and prone to errors. Very little of the Linux kernel is written in assembly language and those parts that are are written only for efficiency and they are specific to particular microprocessors.

2.1.2 The C Programming Language and Compiler

Writing large programs in assembly language is a difficult and time consuming task. It is prone to error and the resulting program is not portable, being tied to one particular processor family. It is far better to use a machine independent language like C[7, The C Programming Language]. C allows you to describe programs in terms of their logical algorithms and the data that they operate on. Special programs called compilers read the C program and translate it into assembly language, generating machine specific code from it. A good compiler can generate assembly instructions that are very nearly as efficient as those written by a good assembly programmer. Most of the Linux kernel is written in the C language. The following C fragment:

```
if (x != y)
        x = y ;
```

performs exactly the same operations as the previous example assembly code. If the contents of the variable `x` are not the same as the contents of variable `y` then the contents of `y` will be copied to `x`. C code is organized into routines, each of which perform a task. Routines may return any value or data type supported by C. Large programs like the Linux kernel comprise many separate C source modules each with its own routines and data structures. These C source code modules group together logical functions such as filesystem handling code.

C supports many types of variables, a variable is a location in memory which can be referenced by a symbolic name. In the above C fragment `x` and `y` refer to locations in memory. The programmer does not care where in memory the variables are put, it is the linker (see below) that has to worry about that. Some variables contain different sorts of data, integer and floating point and others are pointers.

Pointers are variables that contain the address, the location in memory of other data. Consider a variable called x, it might live in memory at address *0x80010000*. You could have a pointer, called px, which points at x. px might live at address *0x80010030*. The value of px would be *0x80010000*: the address of the variable x.

C allows you to bundle together related variables into data structures. For example,

```
struct {
        int i ;
        char b ;
} my_struct ;
```

is a data structure called `my_struct` which contains two elements, an integer (32 bits of data storage) called `i` and a character (8 bits of data) called `b`.

2.1.3 Linkers

Linkers are programs that link together several object modules and libraries to form a single, coherent, program. Object modules are the machine code output from an assembler or compiler and contain executable machine code and data together with information that allows the linker to combine the modules together to form a program. For example one module might contain all of a program's database functions and another module its command line argument handling functions. Linkers fix up references between these object modules, where a routine or data structure referenced in one module actually exists in another module. The Linux kernel is a single, large program linked together from its many constituent object modules.

2.2 What is an Operating System?

Without software a computer is just a pile of electronics that gives off heat. If the hardware is the heart of a computer then the software is its soul. An operating system is a collection of system programs which allow the user to run application software. The operating system abstracts the real hardware of the system and presents the system's users and its applications with a virtual machine. In a very real sense the software provides the character of the system. Most PCs can run one or more operating systems and each one can have a very different look and feel. Linux is made up of a number of functionally separate pieces that, together, comprise the operating system. One obvious part of Linux is the kernel itself; but even that would be useless without libraries or shells.

In order to start understanding what an operating system is, consider what happens when you type an apparently simple command:

```
$ ls
Mail            c               images          perl
docs            tcl
$
```

The $ is a prompt put out by a login shell (in this case `bash`). This means that it is waiting for you, the user, to type some command. Typing ls causes the keyboard driver to recognize that characters have been typed. The keyboard driver passes them to the shell which processes that command by looking for an executable image of the same name. It finds that image, in `/bin/ls`. Kernel services are called to pull the ls executable image into virtual memory and start executing it. The ls image makes calls to the file subsystem of the kernel to find out what files are available. The filesystem might make use of cached filesystem information or use the disk device driver to read this information from the disk. It might even cause a network driver to exchange information with a remote machine to find out details of remote files that this system has access to (filesystems can be remotely mounted via the

Networked File System or NFS). Whichever way the information is located, ls writes that information out and the video driver displays it on the screen.

All of the above seems rather complicated but it shows that even most simple commands reveal that an operating system is in fact a co-operating set of functions that together give you, the user, a coherent view of the system.

2.2.1 Memory management

With infinite resources, for example memory, many of the things that an operating system has to do would be redundant. One of the basic tricks of any operating system is the ability to make a small amount of physical memory behave like rather more memory. This apparently large memory is known as virtual memory. The idea is that the software running in the system is fooled into believing that it is running in a lot of memory. The system divides the memory into easily handled pages and swaps these pages onto a hard disk as the system runs. The software does not notice because of another trick, multi-processing.

2.2.2 Processes

A process could be thought of as a program in action, each process is a separate entity that is running a particular program. If you look at the processes on your Linux system, you will see that there are rather a lot. For example, typing ps shows the following processes on my system:

```
$ ps
  PID TTY STAT  TIME COMMAND
  158 pRe 1     0:00 -bash
  174 pRe 1     0:00 sh /usr/X11R6/bin/startx
  175 pRe 1     0:00 xinit /usr/X11R6/lib/X11/xinit/xinitrc --
  178 pRe 1 N   0:00 bowman
  182 pRe 1 N   0:01 rxvt -geometry 120x35 -fg white -bg black
  184 pRe 1 <   0:00 xclock -bg grey -geometry -1500-1500 -padding 0
  185 pRe 1 <   0:00 xload -bg grey -geometry -0-0 -label xload
  187 pp6 1     9:26 /bin/bash
  202 pRe 1 N   0:00 rxvt -geometry 120x35 -fg white -bg black
  203 ppc 2     0:00 /bin/bash
 1796 pRe 1 N   0:00 rxvt -geometry 120x35 -fg white -bg black
 1797 v06 1     0:00 /bin/bash
 3056 pp6 3 <   0:02 emacs intro/introduction.tex
 3270 pp6 3     0:00 ps
$
```

If my system had many CPUs then each process could (theoretically at least) run on a different CPU. Unfortunately, there is only one so again the operating system resorts to trickery by running each process in turn for a short period. This period of time is known as a time-slice. This trick is known as multi-processing or scheduling and it fools each process into thinking that it is the only process. Processes are protected from one another so that if one process crashes or malfunctions then it will not affect any others. The operating system achieves this by giving each process a separate address space which only they have access to.

2.2.3 Device drivers

Device drivers make up the major part of the Linux kernel. Like other parts of the operating system, they operate in a highly privileged environment and can cause disaster if they get things wrong. Device drivers control the interaction between the operating system and the hardware device that they are controlling. For example, the filesystem makes use of a general block device interface when writing blocks to an IDE disk. The driver takes care of the details and makes device specific things happen. Device drivers are specific to the controller chip that they are driving which is why, for example, you need the NCR810 SCSI driver if your system has an NCR810 SCSI controller.

2.2.4 The Filesystems

In Linux, as it is for Unix™ , the separate filesystems that the system may use are not accessed by device identifiers (such as a drive number or a drive name) but instead they are combined into a single hierarchical tree structure that represents the filesystem as a single entity. Linux adds each new filesystem into this single filesystem tree as they are mounted onto a mount directory, for example `/mnt/cdrom`. One of the most important features of Linux is its support for many different filesystems. This makes it very flexible and well able to coexist with other operating systems. The most popular filesystem for Linux is the `EXT2` filesystem and this is the filesystem supported by most of the Linux distributions.

A filesystem gives the user a sensible view of files and directories held on the hard disks of the system regardless of the filesystem type or the characteristics of the underlying physical device. Linux transparently supports many different filesystems (for example `MS-DOS` and `EXT2`) and presents all of the mounted files and filesystems as one integrated virtual filesystem. So, in general, users and processes do not need to know what sort of filesystem that any file is part of, they just use them.

The block device drivers hide the differences between the physical block device types (for example, `IDE` and `SCSI`) and, so far as each filesystem is concerned, the physical devices are just linear collections of blocks of data. The block sizes may vary between devices, for example 512 bytes is common for floppy devices whereas 1024 bytes is common for IDE devices and, again, this is hidden from the users of the system. An `EXT2` filesystem looks the same no matter what device holds it.

2.3 Kernel Data Structures

The operating system must keep a lot of information about the current state of the system. As things happen within the system these data structures must be changed to reflect the current reality. For example, a new process might be created when a user logs onto the system. The kernel must create a data structure representing the new process and link it with the data structures representing all of the other processes in the system.

Mostly these data structures exist in physical memory and are accessible only by the kernel and its subsystems. Data structures contain data and pointers; addresses of other data structures or the addresses of routines. Taken all together, the data structures used by the Linux kernel can look very confusing. Every data structure has a purpose and although some are used by several kernel subsystems, they are more simple than they appear at first sight.

Understanding the Linux kernel hinges on understanding its data structures and the use that the various functions within the Linux kernel makes of them. This book bases its description of the Linux kernel on its data structures. It talks about each kernel subsystem in terms of its algorithms, its methods of getting things done, and their usage of the kernel's data structures.

2.3.1 Linked Lists

Linux uses a number of software engineering techniques to link together its data structures. On a lot of occasions it uses *linked* or *chained* data structures. If each data structure describes a single instance or occurance of something, for example a process or a network device, the kernel must be able to find all of the instances. In a linked list a root pointer contains the address of the first data structure, or *element*, in the list and each data structure contains a pointer to the next element in the list. The last element's next pointer would be 0 or NULL to show that it is the end of the list. In a *doubly linked* list each element contains both a pointer to the next element in the list but also a pointer to the previous element in the list. Using doubly linked lists makes it easier to add or remove elements from the middle of list although you do need more memory accesses. This is a typical operating system trade off: memory accesses versus CPU cycles.

2.3.2 Hash Tables

Linked lists are handy ways of tying data structures together but navigating linked lists can be inefficient. If you were searching for a particular element, you might easily have to look at the whole list before you find the one that you need. Linux uses another technique, *hashing* to get around this restriction. A *hash table* is an *array* or *vector* of pointers. An array, or vector, is simply a set of things coming one after another in memory. A bookshelf could be said to be an array of books. Arrays are accessed by an *index*, the index is an offset into the array. Taking the bookshelf analogy a little further, you could describe each book by its position on the shelf; you might ask for the 5th book.

A hash table is an array of pointers to data structures and its index is derived from information in those data structures. If you had data structures describing the population of a village then you could use a person's age as an index. To find a particular person's data you could use their age as an index into the population hash table and then follow the pointer to the data structure containing the person's details. Unfortunately many people in the village are likely to have the same age and so the hash table pointer becomes a pointer to a chain or list of data structures each describing people of the same age. However, searching these shorter chains is still faster than searching all of the data structures.

As a hash table speeds up access to commonly used data structures, Linux often uses hash tables to implement *caches*. Caches are handy information that needs to be accessed quickly and are usually a subset of the full set of information available. Data structures are put into a cache and kept there because the kernel often accesses them. There is a drawback to caches in that they are more complex to use and maintain than simple linked lists or hash tables. If the data structure can be found in the cache (this is known as a *cache hit*, then all well and good. If it cannot then all of the relevant data structures must be searched and, if the data structure exists at all, it must be added into the cache. In adding new data structures into the cache an old cache entry may need discarding. Linux must decide which one to discard, the danger being that the discarded data structure may be the next one that Linux needs.

2.3.3 Abstract Interfaces

The Linux kernel often abstracts its interfaces. An interface is a collection of routines and data structures which operate in a particular way. For example all network device drivers have to provide certain routines in which particular data structures are operated on. This way there can be generic layers of code using the services (interfaces) of lower layers of specific code. The network layer is generic and it is supported by device specific code that conforms to a standard interface.

Often these lower layers *register* themselves with the upper layer at boot time. This registration usually involves adding a data structure to a linked list. For example each filesystem built into the kernel registers itself with the kernel at boot time or, if you are using modules, when the filesystem is first used. You can see which filesystems have registered themselves by looking at the file `/proc/filesystems`. The registration data structure often includes pointers to functions. These are the addresses of software functions that perform particular tasks. Again, using filesystem registration as an example, the data structure that each filesystem passes to the Linux kernel as it registers includes the address of a filesystem specfic routine which must be called whenever that filesystem is mounted.

Chapter 3

Memory Management

The memory management subsystem is one of the most important parts of the operating system. Since the early days of computing, there has been a need for more memory than exists physically in a system. Strategies have been developed to overcome this limitation and the most successful of these is virtual memory. Virtual memory makes the system appear to have more memory than it actually has by sharing it between competing processes as they need it.

Virtual memory does more than just make your computer's memory go further. The memory management subsystem provides:

Large Address Spaces The operating system makes the system appear as if it has a larger amount of memory than it actually has. The virtual memory can be many times larger than the physical memory in the system,

Protection Each process in the system has its own virtual address space. These virtual address spaces are completely separate from each other and so a process running one application cannot affect another. Also, the hardware virtual memory mechanisms allow areas of memory to be protected against writing. This protects code and data from being overwritten by rogue applications.

Memory Mapping Memory mapping is used to map image and data files into a processes address space. In memory mapping, the contents of a file are linked directly into the virtual address space of a process.

Fair Physical Memory Allocation The memory management subsystem allows each running process in the system a fair share of the physical memory of the system,

Shared Virtual Memory Although virtual memory allows processes to have separate (virtual) address spaces, there are times when you need processes to share memory. For example there could be several processes in the system running

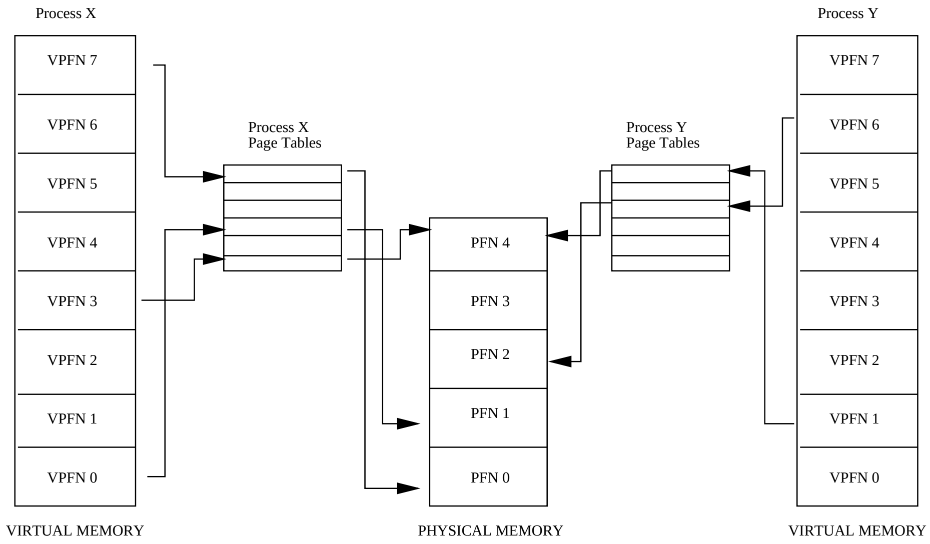

Figure 3.1: Abstract model of Virtual to Physical address mapping

the `bash` command shell. Rather than have several copies of `bash`, one in each processes virtual address space, it is better to have only one copy in physical memory and all of the processes running `bash` share it. Dynamic libraries are another common example of executing code shared between several processes.

Shared memory can also be used as an Inter Process Communication (IPC) mechanism, with two or more processes exchanging information via memory common to all of them. Linux supports the Unix™ System V shared memory IPC.

3.1 An Abstract Model of Virtual Memory

Before considering the methods that Linux uses to support virtual memory it is useful to consider an abstract model that is not cluttered by too much detail.

As the processor executes a program it reads an instruction from memory and decodes it. In decoding the instruction it may need to fetch or store the contents of a location in memory. The processor then executes the instruction and moves onto the next instruction in the program. In this way the processor is always accessing memory either to fetch instructions or to fetch and store data.

In a virtual memory system all of these addresses are virtual addresses and not physical addresses. These virtual addresses are converted into physical addresses by the processor based on information held in a set of tables maintained by the operating system.

To make this translation easier, virtual and physical memory are divided into handy sized chunks called *pages*. These pages are all the same size, they need not be but if they were not, the system would be very hard to administer. Linux on Alpha AXP systems uses 8 Kbyte pages and on Intel x86 systems it uses 4 Kbyte pages. Each of these pages is given a unique number; the page frame number (PFN). In this paged model, a virtual address is composed of two parts; an offset and a virtual page frame number. If the page size is 4 Kbytes, bits 11:0 of the virtual address contain the offset and bits 12 and above are the virtual page frame number. Each time the processor encounters a virtual address it must extract the offset and the virtual page frame number. The processor must translate the virtual page frame number into a physical one and then access the location at the correct offset into that physical page. To do this the processor uses *page tables.*

Figure 3.1 shows the virtual address spaces of two processes, process X and process Y, each with their own page tables. These page tables map each processes virtual pages into physical pages in memory. This shows that process *X's* virtual page frame number 0 is mapped into memory in physical page frame number 1 and that process *Y's* virtual page frame number 1 is mapped into physical page frame number 4. Each entry in the theoretical page table contains the following information:

- Valid flag. This indicates if this page table entry is valid,
- The physical page frame number that this entry is describing,
- Access control information. This describes how the page may be used. Can it be written to? Does it contain executable code?

The page table is accessed using the virtual page frame number as an offset. Virtual page frame 5 would be the 6th element of the table (0 is the first element).

To translate a virtual address into a physical one, the processor must first work out the virtual addresses page frame number and the offset within that virtual page. By making the page size a power of 2 this can be easily done by masking and shifting. Looking again at Figures 3.1 and assuming a page size of *0x2000* bytes (which is decimal 8192) and an address of *0x2194* in process *Y's* virtual address space then the processor would translate that address into offset *0x194* into virtual page frame number 1.

The processor uses the virtual page frame number as an index into the processes page table to retrieve its page table entry. If the page table entry at that offset is valid, the processor takes the physical page frame number from this entry. If the entry is invalid, the process has accessed a non-existent area of its virtual memory. In this case, the processor cannot resolve the address and must pass control to the operating system so that it can fix things up.

Just how the processor notifies the operating system that the correct process has attempted to access a virtual address for which there is no valid translation is specific to the processor. However the processor delivers it, this is known as a *page fault* and the operating system is notified of the faulting virtual address and the reason for the page fault.

Assuming that this is a valid page table entry, the processor takes that physical page frame number and multiplies it by the page size to get the address of the base of the page in physical memory. Finally, the processor adds in the offset to the instruction or data that it needs.

Using the above example again, process *Y's* virtual page frame number 1 is mapped to physical page frame number 4 which starts at *0x8000* (4 x *0x2000*). Adding in the *0x194* byte offset gives us a final physical address of *0x8194*.

By mapping virtual to physical addresses this way, the virtual memory can be mapped into the system's physical pages in any order. For example, in Figure 3.1 process *X's* virtual page frame number 0 is mapped to physical page frame number

1 whereas virtual page frame number 7 is mapped to physical page frame number 0 even though it is higher in virtual memory than virtual page frame number 0. This demonstrates an interesting byproduct of virtual memory; the pages of virtual memory do not have to be present in physical memory in any particular order.

3.1.1 Demand Paging

As there is much less physical memory than virtual memory the operating system must be careful that it does not use the physical memory inefficiently. One way to save physical memory is to only load virtual pages that are currently being used by the executing program. For example, a database program may be run to query a database. In this case not all of the database needs to be loaded into memory, just those data records that are being examined. If the database query is a search query then it does not make sense to load the code from the database program that deals with adding new records. This technique of only loading virtual pages into memory as they are accessed is known as demand paging.

When a process attempts to access a virtual address that is not currently in memory the processor cannot find a page table entry for the virtual page referenced. For example, in Figure 3.1 there is no entry in process *X's* page table for virtual page frame number 2 and so if process *X* attempts to read from an address within virtual page frame number 2 the processor cannot translate the address into a physical one. At this point the processor notifies the operating system that a page fault has occurred.

If the faulting virtual address is invalid this means that the process has attempted to access a virtual address that it should not have. Maybe the application has gone wrong in some way, for example writing to random addresses in memory. In this case the operating system will terminate it, protecting the other processes in the system from this rogue process.

If the faulting virtual address was valid but the page that it refers to is not currently in memory, the operating system must bring the appropriate page into memory from the image on disk. Disk access takes a long time, relatively speaking, and so the process must wait quite a while until the page has been fetched. If there are other processes that could run then the operating system will select one of them to run. The fetched page is written into a free physical page frame and an entry for the virtual page frame number is added to the processes page table. The process is then restarted at the machine instruction where the memory fault occurred. This time the virtual memory access is made, the processor can make the virtual to physical address translation and so the process continues to run.

Linux uses demand paging to load executable images into a processes virtual memory. Whenever a command is executed, the file containing it is opened and its contents are mapped into the processes virtual memory. This is done by modifying the data structures describing this processes memory map and is known as *memory mapping*. However, only the first part of the image is actually brought into physical memory. The rest of the image is left on disk. As the image executes, it generates page faults and Linux uses the processes memory map in order to determine which parts of the image to bring into memory for execution.

3.1.2 Swapping

If a process needs to bring a virtual page into physical memory and there are no free physical pages available, the operating system must make room for this page by discarding another page from physical memory.

If the page to be discarded from physical memory came from an image or data file and has not been written to then the page does not need to be saved. Instead it can be discarded and if the process needs that page again it can be brought back into memory from the image or data file.

However, if the page has been modified, the operating system must preserve the contents of that page so that it can be accessed at a later time. This type of page is known as a *dirty* page and when it is removed from memory it is saved in a special sort of file called the swap file. Accesses to the swap file are very long relative to the speed of the processor and physical memory and the operating system must juggle the need to write pages to disk with the need to retain them in memory to be used again.

If the algorithm used to decide which pages to discard or swap (the *swap algorithm* is not efficient then a condition known as *thrashing* occurs. In this case, pages are constantly being written to disk and then being read back and the operating system is too busy to allow much real work to be performed. If, for example, physical page frame number 1 in Figure 3.1 is being regularly accessed then it is not a good candidate for swapping to hard disk. The set of pages that a process is currently using is called the *working set*. An efficient swap scheme would make sure that all processes have their working set in physical memory.

Linux uses a Least Recently Used (LRU) page aging technique to fairly choose pages which might be removed from the system. This scheme involves every page in the system having an age which changes as the page is accessed. The more that a page is accessed, the younger it is; the less that it is accessed the older and more stale it becomes. Old pages are good candidates for swapping.

3.1.3 Shared Virtual Memory

Virtual memory makes it easy for several processes to share memory. All memory access are made via page tables and each process has its own separate page table. For two processes sharing a physical page of memory, its physical page frame number must appear in a page table entry in both of their page tables.

Figure 3.1 shows two processes that each share physical page frame number 4. For process X this is virtual page frame number 4 whereas for process Y this is virtual page frame number 6. This illustrates an interesting point about sharing pages: the shared physical page does not have to exist at the same place in virtual memory for any or all of the processes sharing it.

3.1.4 Physical and Virtual Addressing Modes

It does not make much sense for the operating system itself to run in virtual memory. This would be a nightmare situation where the operating system must maintain page tables for itself. Most multi-purpose processors support the notion of a physical address mode as well as a virtual address mode. Physical addressing mode requires no

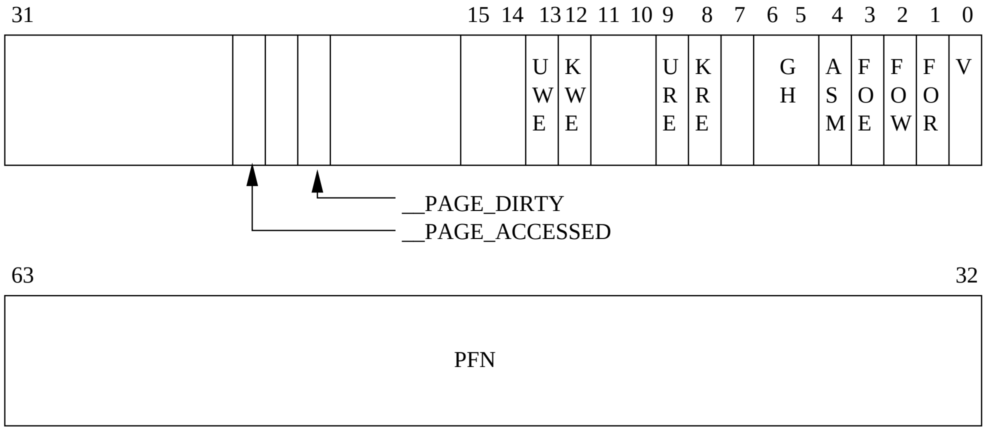

Figure 3.2: Alpha AXP Page Table Entry

page tables and the processor does not attempt to perform any address translations in this mode. The Linux kernel is linked to run in physical address space.

The Alpha AXP processor does not have a special physical addressing mode. Instead, it divides up the memory space into several areas and designates two of them as physically mapped addresses. This kernel address space is known as KSEG address space and it encompasses all addresses upwards from *0xfffffc0000000000*. In order to execute from code linked in KSEG (by definition, kernel code) or access data there, the code must be executing in kernel mode. The Linux kernel on Alpha is linked to execute from address *0xfffffc0000310000*.

3.1.5 Access Control

The page table entries also contain access control information. As the processor is already using the page table entry to map a processes virtual address to a physical one, it can easily use the access control information to check that the process is not accessing memory in a way that it should not.

There are many reasons why you would want to restrict access to areas of memory. Some memory, such as that containing executable code, is naturally read only memory; the operating system should not allow a process to write data over its executable code. By contrast, pages containing data can be written to but attempts to execute that memory as instructions should fail. Most processors have at least two modes of execution: *kernel* and *user*. You would not want kernel code executing by a user or kernel data structures to be accessible except when the processor is running in kernel mode.

The access control information is held in the PTE and is processor specific; figure 3.2 shows the PTE for Alpha AXP. The bit fields have the following meanings:

V Valid, if set this PTE is valid,

FOE "Fault on Execute", Whenever an attempt to execute instructions in this page occurs, the processor reports a page fault and passes control to the operating system,

FOW "Fault on Write", as above but page fault on an attempt to write to this page,

FOR "Fault on Read", as above but page fault on an attempt to read from this page,

ASM Address Space Match. This is used when the operating system wishes to clear only some of the entries from the Translation Buffer,

KRE Code running in kernel mode can read this page,

URE Code running in user mode can read this page,

GH Granularity hint used when mapping an entire block with a single Translation Buffer entry rather than many,

KWE Code running in kernel mode can write to this page,

UWE Code running in user mode can write to this page,

page frame number For PTEs with the **V** bit set, this field contains the physical Page Frame Number (page frame number) for this PTE. For invalid PTEs, if this field is not zero, it contains information about where the page is in the swap file.

The following two bits are defined and used by Linux:

_PAGE_DIRTY if set, the page needs to be written out to the swap file,

_PAGE_ACCESSED Used by Linux to mark a page as having been accessed.

3.2 Caches

If you were to implement a system using the above theoretical model then it would work, but not particularly efficiently. Both operating system and processor designers try hard to extract more performance from the system. Apart from making the processors, memory and so on faster the best approach is to maintain caches of useful information and data that make some operations faster. Linux uses a number of memory management related caches:

Buffer Cache The buffer cache contains data buffers that are used by the block device drivers. These buffers are of fixed sizes (for example 512 bytes) and contain blocks of information that have either been read from a block device or are being written to it. A block device is one that can only be accessed by reading and writing fixed sized blocks of data. All hard disks are block devices.

See `fs/buffer.c`

The buffer cache is indexed via the device identifier and the desired block number and is used to quickly find a block of data. Block devices are only ever accessed via the buffer cache. If data can be found in the buffer cache then it does not need to be read from the physical block device, for example a hard disk, and access to it is much faster.

Page Cache This is used to speed up access to images and data on disk. It is used to cache the logical contents of a file a page at a time and is accessed via the file and offset within the file. As pages are read into memory from disk, they are cached in the page cache.

See `mm/filemap.c`

See swap.h, mm/swap_state.c mm/swapfile.c

Swap Cache Only modified (or *dirty*) pages are saved in the swap file. So long as these pages are not modified after they have been written to the swap file then the next time the page is swapped out there is no need to write it to the swap file as the page is already in the swap file. Instead the page can simply be discarded. In a heavily swapping system this saves many unnecessary and costly disk operations.

Hardware Caches One commonly implemented hardware cache is in the processor; a cache of Page Table Entries. In this case, the processor does not always read the page table directly but instead caches translations for pages as it needs them. These are the Translation Look-aside Buffers and contain cached copies of the page table entries from one or more processes in the system.

When the reference to the virtual address is made, the processor will attempt to find a matching TLB entry. If it finds one, it can directly translate the virtual address into a physical one and perform the correct operation on the data. If the processor cannot find a matching TLB entry then it must get the operating system to help. It does this by signalling the operating system that a TLB miss has occurred. A system specific mechanism is used to deliver that exception to the operating system code that can fix things up. The operating system generates a new TLB entry for the address mapping. When the exception has been cleared, the processor will make another attempt to translate the virtual address. This time it will work because there is now a valid entry in the TLB for that address.

The drawback of using caches, hardware or otherwise, is that in order to save effort Linux must use more time and space maintaining these caches and, if the caches become corrupted, the system will crash.

3.3 Linux Page Tables

Linux assumes that there are three levels of page tables. Each Page Table accessed contains the page frame number of the next level of Page Table. Figure 3.3 shows how a virtual address can be broken into a number of fields; each field providing an offset into a particular Page Table. To translate a virtual address into a physical one, the processor must take the contents of each level field, convert it into an offset into the physical page containing the Page Table and read the page frame number of the next level of Page Table. This is repeated three times until the page frame number of the physical page containing the virtual address is found. Now the final field in the virtual address, the byte offset, is used to find the data inside the page.

See include/-asm/pgtable.h

Each platform that Linux runs on must provide translation macros that allow the kernel to traverse the page tables for a particular process. This way, the kernel does not need to know the format of the page table entries or how they are arranged. This is so successful that Linux uses the same page table manipulation code for the Alpha processor, which has three levels of page tables, and for Intel x86 processors, which have two levels of page tables.

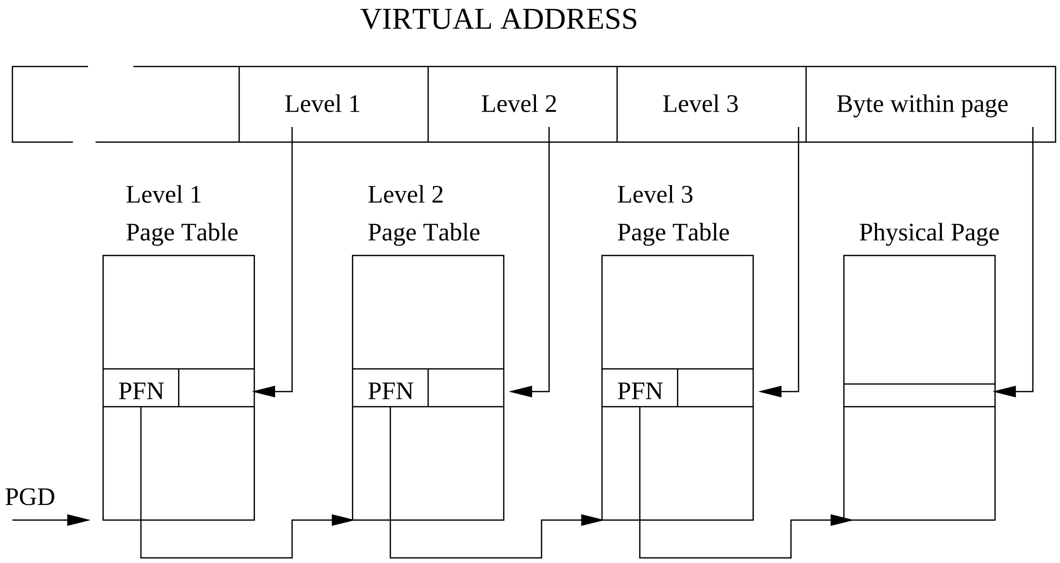

Figure 3.3: Three Level Page Tables

3.4 Page Allocation and Deallocation

There are many demands on the physical pages in the system. For example, when an image is loaded into memory the operating system needs to allocate pages. These will be freed when the image has finished executing and is unloaded. Another use for physical pages is to hold kernel specific data structures such as the page tables themselves. The mechanisms and data structures used for page allocation and deallocation are perhaps the most critical in maintaining the efficiency of the virtual memory subsystem.

All of the physical pages in the system are described by the `mem_map` data structure which is a list of `mem_map_t` [1] structures which is initialized at boot time. Each `mem_map_t` describes a single physical page in the system. Important fields (so far as memory management is concerned) are:

See `include/-linux/mm.h`

count This is a count of the number of users of this page. The count is greater than one when the page is shared between many processes,

age This field describes the age of the page and is used to decide if the page is a good candidate for discarding or swapping,

map_nr This is the physical page frame number that this `mem_map_t` describes.

The `free_area` vector is used by the page allocation code to find and free pages. The whole buffer management scheme is supported by this mechanism and so far as the code is concerned, the size of the page and physical paging mechanisms used by the processor are irrelevant.

Each element of `free_area` contains information about blocks of pages. The first element in the array describes single pages, the next blocks of 2 pages, the next blocks of 4 pages and so on upwards in powers of two. The `list` element is used as a queue head and has pointers to the `page` data structures in the `mem_map` array. Free

[1] Confusingly the structure is also known as the *page* structure.

blocks of pages are queued here. `map` is a pointer to a bitmap which keeps track of allocated groups of pages of this size. Bit N of the bitmap is set if the Nth block of pages is free.

Figure 3.4 shows the `free_area` structure. Element 0 has one free page (page frame number 0) and element 2 has 2 free blocks of 4 pages, the first starting at page frame number 4 and the second at page frame number 56.

3.4.1 Page Allocation

See `__get_free_pages()` in `mm/page_alloc.c`

Linux uses the Buddy algorithm [2] to effectively allocate and deallocate blocks of pages. The page allocation code attempts to allocate a block of one or more physical pages. Pages are allocated in blocks which are powers of 2 in size. That means that it can allocate a block 1 page, 2 pages, 4 pages and so on. So long as there are enough free pages in the system to grant this request ($nr\_free\_pages > min\_free\_pages$) the allocation code will search the `free_area` for a block of pages of the size requested. Each element of the `free_area` has a map of the allocated and free blocks of pages for that sized block. For example, element 2 of the array has a memory map that describes free and allocated blocks each of 4 pages long.

The allocation algorithm first searches for blocks of pages of the size requested. It follows the chain of free pages that is queued on the *list* element of the `free_area` data structure. If no blocks of pages of the requested size are free, blocks of the next size (which is twice that of the size requested) are looked for. This process continues until all of the `free_area` has been searched or until a block of pages has been found. If the block of pages found is larger than that requested it must be broken down until there is a block of the right size. Because the blocks are each a power of 2 pages big then this breaking down process is easy as you simply break the blocks in half. The free blocks are queued on the appropriate queue and the allocated block of pages is returned to the caller.

For example, in Figure 3.4 if a block of 2 pages was requested, the first block of 4 pages (starting at page frame number 4) would be broken into two 2 page blocks. The first, starting at page frame number 4 would be returned to the caller as the allocated pages and the second block, starting at page frame number 6 would be queued as a free block of 2 pages onto element 1 of the `free_area` array.

3.4.2 Page Deallocation

See `free_pages()` in `mm/page_alloc.c`

Allocating blocks of pages tends to fragment memory with larger blocks of free pages being broken down into smaller ones. The page deallocation code recombines pages into larger blocks of free pages whenever it can. In fact the page block size is important as it allows for easy combination of blocks into larger blocks.

Whenever a block of pages is freed, the adjacent or buddy block of the same size is checked to see if it is free. If it is, then it is combined with the newly freed block of pages to form a new free block of pages for the next size block of pages. Each time two blocks of pages are recombined into a bigger block of free pages the page deallocation code attempts to recombine that block into a yet larger one. In this way the blocks of free pages are as large as memory usage will allow.

[2] Bibliography reference here

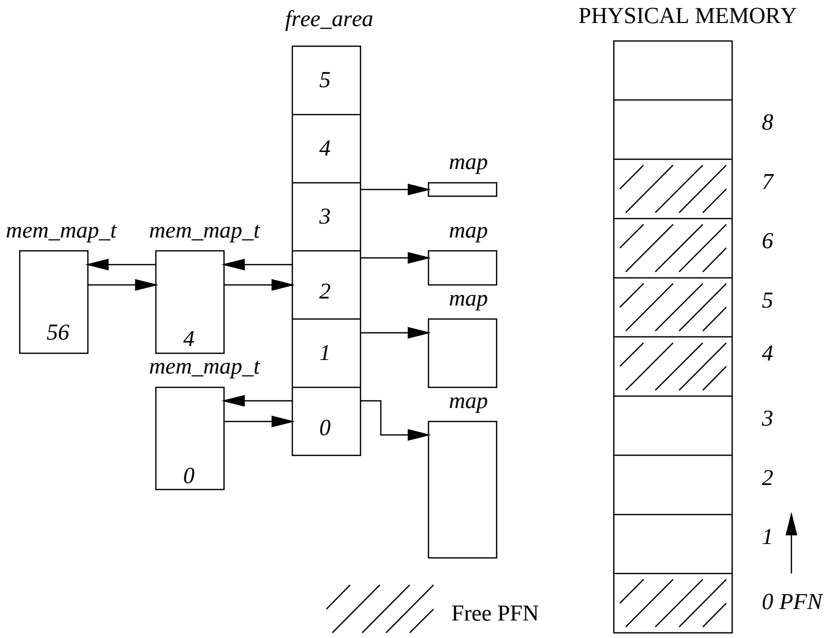

Figure 3.4: The `free_area` data structure

For example, in Figure 3.4, if page frame number 1 were to be freed, then that would be combined with the already free page frame number 0 and queued onto element 1 of the `free_area` as a free block of size 2 pages.

3.5 Memory Mapping

When an image is executed, the contents of the executable image must be brought into the processes virtual address space. The same is also true of any shared libraries that the executable image has been linked to use. The executable file is not actually brought into physical memory, instead it is merely linked into the processes virtual memory. Then, as the parts of the program are referenced by the running application, the image is brought into memory from the executable image. This linking of an image into a processes virtual address space is known as memory mapping.

Every processes virtual memory is represented by an `mm_struct` data structure. This contains information about the image that it is currently executing (for example bash) and also has pointers to a number of `vm_area_struct` data structures. Each `vm_area_struct` data structure describes the start and end of the area of virtual memory, the processes access rights to that memory and a set of operations for that memory. These operations are a set of routines that Linux must use when manipulating this area of virtual memory. For example, one of the virtual memory operations performs the correct actions when the process has attempted to access this virtual memory but finds (via a page fault) that the memory is not actually in physical memory. This operation is the *nopage* operation. The *nopage* operation is used when Linux demand pages the pages of an executable image into memory.

When an executable image is mapped into a processes virtual address a set of `vm_area_struct` data structures is generated. Each `vm_area_struct` data structure represents a part of the executable image; the executable code, initialized data

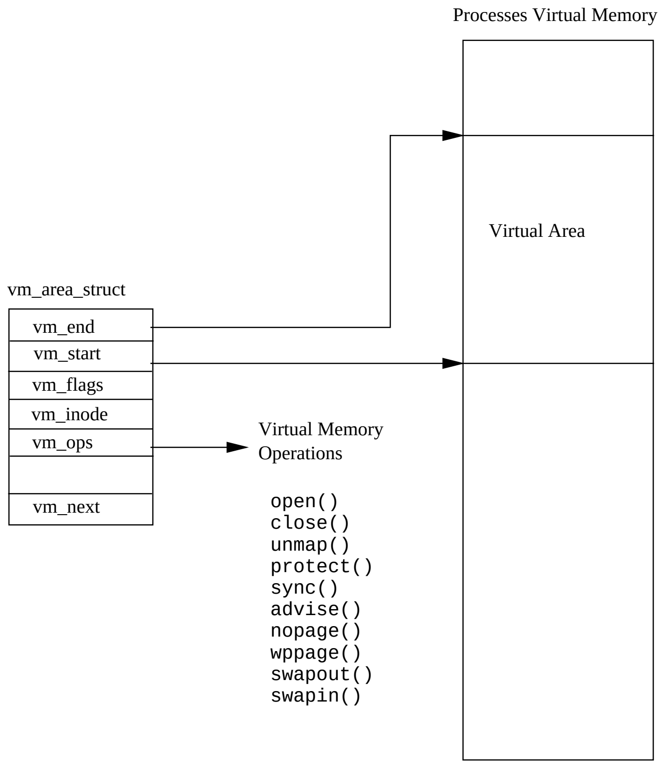

Figure 3.5: Areas of Virtual Memory

(variables), unitialized data and so on. Linux supports a number of standard virtual memory operations and as the `vm_area_struct` data structures are created, the correct set of virtual memory operations are associated with them.

3.6 Demand Paging

Once an executable image has been memory mapped into a processes virtual memory it can start to execute. As only the very start of the image is physically pulled into memory it will soon access an area of virtual memory that is not yet in physical memory. When a process accesses a virtual address that does not have a valid page table entry, the processor will report a page fault to Linux. The page fault describes the virtual address where the page fault occurred and the type of memory access that caused.

See `handle_mm_fault()` in `mm/memory.c`

Linux must find the `vm_area_struct` that represents the area of memory that the page fault occurred in. As searching through the `vm_area_struct` data structures is critical to the efficient handling of page faults, these are linked together in an AVL (Adelson-Velskii and Landis) tree structure. If there is no `vm_area_struct` data structure for this faulting virtual address, this process has accessed an illegal virtual address. Linux will signal the process, sending a `SIGSEGV` signal, and if the process does not have a handler for that signal it will be terminated.

Linux next checks the type of page fault that occurred against the types of accesses allowed for this area of virtual memory. If the process is accessing the memory in an illegal way, say writing to an area that it is only allowed to read from, it is also

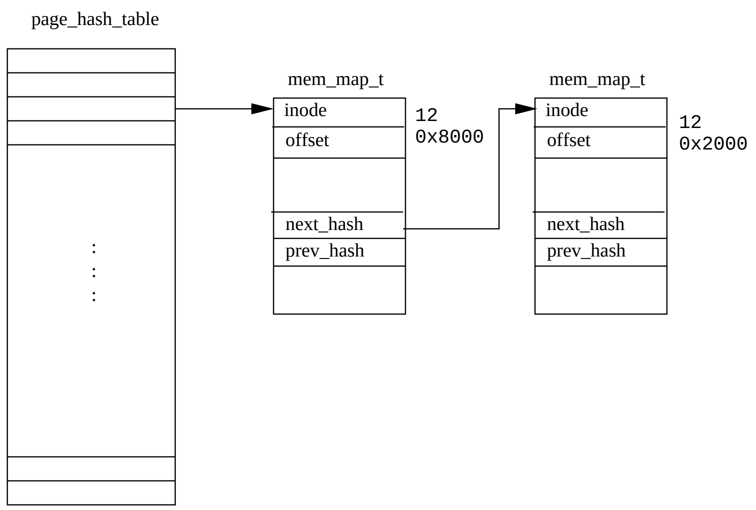

Figure 3.6: The Linux Page Cache

signalled with a memory error.

Now that Linux has determined that the page fault is legal, it must deal with it. Linux must differentiate between pages that are in the swap file and those that are part of an executable image on a disk somewhere. It does this by using the page table entry for this faulting virtual address.

See `do_no_page()` in `mm/memory.c`

If the page's page table entry is invalid but not empty, the page fault is for a page currently being held in the swap file. For Alpha AXP page table entries, these are entries which do not have their valid bit set but which have a non-zero value in their PFN field. In this case the PFN field holds information about where in the swap (and which swap file) the page is being held. How pages in the swap file are handled is described later in this chapter.

Not all `vm_area_struct` data structures have a set of virtual memory operations and even those that do may not have a *nopage* operation. This is because by default Linux will fix up the access by allocating a new physical page and creating a valid page table entry for it. If there is a *nopage* operation for this area of virtual memory, Linux will use it.

The generic Linux *nopage* operation is used for memory mapped executable images and it uses the page cache to bring the required image page into physical memory.

See `filemap_nopage()` in `mm/filemap.c`

However the required page is brought into physical memory, the processes page tables are updated. It may be necessary for hardware specific actions to update those entries, particularly if the processor uses translation look aside buffers. Now that the page fault has been handled it can be dismissed and the process is restarted at the instruction that made the faulting virtual memory access.

3.7 The Linux Page Cache

The role of the Linux page cache is to speed up access to files on disk. Memory mapped files are read a page at a time and these pages are stored in the page cache. Figure 3.6 shows that the page cache consists of the `page_hash_table`, a vector of pointers to `mem_map_t` data structures. Each file in Linux is identified by a VFS

See `include/-linux/pagemap.h`

`inode` data structure (described in Chapter 9) and each VFS `inode` is unique and fully describes one and only one file. The index into the page table is derived from the file's VFS `inode` and the offset into the file.

Whenever a page is read from a memory mapped file, for example when it needs to be brought back into memory during demand paging, the page is read through the page cache. If the page is present in the cache, a pointer to the `mem_map_t` data structure representing it is returned to the page fault handling code. Otherwise the page must be brought into memory from the file system that holds the image. Linux allocates a physical page and reads the page from the file on disk.

If it is possible, Linux will initiate a read of the next page in the file. This single page read ahead means that if the process is accessing the pages in the file serially, the next page will be waiting in memory for the process.

Over time the page cache grows as images are read and executed. Pages will be removed from the cache as they are no longer needed, say as an image is no longer being used by any process. As Linux uses memory it can start to run low on physical pages. In this case Linux will reduce the size of the page cache.

3.8 Swapping Out and Discarding Pages

When physical memory becomes scarce the Linux memory management subsystem must attempt to free physical pages. This task falls to the kernel swap daemon (*kswapd*). The kernel swap daemon is a special type of process, a kernel thread. Kernel threads are processes have no virtual memory, instead they run in kernel mode in the physical address space. The kernel swap daemon is slightly misnamed in that it does more than merely swap pages out to the system's swap files. Its role is make sure that there are enough free pages in the system to keep the memory management system operating efficiently.

See `kswapd()` in `mm/vmscan.c`

The Kernel swap daemon (*kswapd*) is started by the kernel init process at startup time and sits waiting for the kernel swap timer to periodically expire. Every time the timer expires, the swap daemon looks to see if the number of free pages in the system is getting too low. It uses two variables, *free_pages_high* and *free_pages_low* to decide if it should free some pages. So long as the number of free pages in the system remains above *free_pages_high*, the kernel swap daemon does nothing; it sleeps again until its timer next expires. For the purposes of this check the kernel swap daemon takes into account the number of pages currently being written out to the swap file. It keeps a count of these in *nr_async_pages*; this is incremented each time a page is queued waiting to be written out to the swap file and decremented when the write to the swap device has completed. *free_pages_low* and *free_pages_high* are set at system startup time and are related to the number of physical pages in the system. If the number of free pages in the system has fallen below *free_pages_high* or worse still *free_pages_low*, the kernel swap daemon will try three ways to reduce the number of physical pages being used by the system:

Reducing the size of the buffer and page caches,

Swapping out System V shared memory pages,

Swapping out and discarding pages.

If the number of free pages in the system has fallen below *free_pages_low*, the kernel swap daemon will try to free 6 pages before it next runs. Otherwise it will try to free 3 pages. Each of the above methods are tried in turn until enough pages have been freed. The kernel swap daemon remembers which method it was using the last time that it attempted to free physical pages. Each time it runs it will start trying to free pages using this last successful method.

After it has free sufficient pages, the swap daemon sleeps again until its timer expires. If the reason that the kernel swap daemon freed pages was that the number of free pages in the system had fallen below *free_pages_low*, it only sleeps for half its usual time. Once the number of free pages is more than *free_pages_low* the kernel swap daemon goes back to sleeping longer between checks.

3.8.1 Reducing the Size of the Page and Buffer Caches

The pages held in the page and buffer caches are good candidates for being freed into the `free_area` vector. The Page Cache, which contains pages of memory mapped files, may contain unneccessary pages that are filling up the system's memory. Likewise the Buffer Cache, which contains buffers read from or being written to physical devices, may also contain unneeded buffers. When the physical pages in the system start to run out, discarding pages from these caches is relatively easy as it requires no writing to physical devices (unlike swapping pages out of memory). Discarding these pages does not have too many harmful side effects other than making access to physical devices and memory mapped files slower. However, if the discarding of pages from these caches is done fairly, all processes will suffer equally.

See `shrink_mmap()` in `mm/filemap.c`

Every time the Kernel swap daemon tries to shrink these caches it examines a block of pages in the `mem_map` page vector to see if any can be discarded from physical memory. The size of the block of pages examined is higher if the kernel swap daemon is intensively swapping; that is if the number of free pages in the system has fallen dangerously low. The blocks of pages are examined in a cyclical manner; a different block of pages is examined each time an attempt is made to shrink the memory map. This is known as the *clock* algorithm as, rather like the minute hand of a clock, the whole `mem_map` page vector is examined a few pages at a time.

Each page being examined is checked to see if it is cached in either the page cache or the buffer cache. You should note that shared pages are not considered for discarding at this time and that a page cannot be in both caches at the same time. If the page is not in either cache then the next page in the `mem_map` page vector is examined.

Pages are cached in the buffer cache (or rather the buffers within the pages are cached) to make buffer allocation and deallocation more efficient. The memory map shrinking code tries to free the buffers that are contained within the page being examined. If all the buffers are freed, then the pages that contain them are also be freed. If the examined page is in the Linux page cache, it is removed from the page cache and freed.

See `try_to_free_buffer()` in `fs/buffer.c`

When enough pages have been freed on this attempt then the kernel swap daemon will wait until the next time it is periodically woken. As none of the freed pages were part of any process's virtual memory (they were cached pages), then no page tables need updating. If there were not enough cached pages discarded then the swap daemon will try to swap out some shared pages.

3.8.2 Swapping Out System V Shared Memory Pages

System V shared memory is an inter-process communication mechanism which allows two or more processes to share virtual memory in order to pass information amongst themselves. How processes share memory in this way is described in more detail in Chapter 5. For now it is enough to say that each area of System V shared memory is described by a `shmid_ds` data structure. This contains a pointer to a list of `vm_area_struct` data structures, one for each process sharing this area of virtual memory. The `vm_area_struct` data structures describe where in each processes virtual memory this area of System V shared memory goes. Each `vm_area_struct` data structure for this System V shared memory is linked together using the `vm_next_shared` and `vm_prev_shared` pointers. Each `shmid_ds` data structure also contains a list of page table entries each of which describes the physical page that a shared virtual page maps to.

See `shm_swap()` in `ipc/shm.c`

The kernel swap daemon also uses a *clock* algorithm when swapping out System V shared memory pages. . Each time it runs it remembers which page of which shared virtual memory area it last swapped out. It does this by keeping two indices, the first is an index into the set of `shmid_ds` data structures, the second into the list of page table entries for this area of System V shared memory. This makes sure that it fairly victimizes the areas of System V shared memory.

As the physical page frame number for a given virtual page of System V shared memory is contained in the page tables of all of the processes sharing this area of virtual memory, the kernel swap daemon must modify all of these page tables to show that the page is no longer in memory but is now held in the swap file. For each shared page it is swapping out, the kernel swap daemon finds the page table entry in each of the sharing processes page tables (by following a pointer from each `vm_area_struct` data structure). If this processes page table entry for this page of System V shared memory is valid, it converts it into an invalid but swapped out page table entry and reduces this (shared) page's count of users by one. The format of a swapped out System V shared page table entry contains an index into the set of `shmid_ds` data structures and an index into the page table entries for this area of System V shared memory.

If the page's count is zero after the page tables of the sharing processes have all been modified, the shared page can be written out to the swap file. The page table entry in the list pointed at by the `shmid_ds` data structure for this area of System V shared memory is replaced by a swapped out page table entry. A swapped out page table entry is invalid but contains an index into the set of open swap files and the offset in that file where the swapped out page can be found. This information will be used when the page has to be brought back into physical memory.

3.8.3 Swapping Out and Discarding Pages

See `swap_out()` in `mm/vmscan.c`

The swap daemon looks at each process in the system in turn to see if it is a good candidate for swapping. Good candidates are processes that can be swapped (some cannot) and that have one or more pages which can be swapped or discarded from memory. Pages are swapped out of physical memory into the system's swap files only if the data in them cannot be retrieved another way.

A lot of the contents of an executable image come from the image's file and can easily be re-read from that file. For example, the executable instructions of an image will never be modified by the image and so will never be written to the swap file. These pages can simply be discarded; when they are again referenced by the process, they will be brought back into memory from the executable image.

To do this it follows the `vm_next` pointer along the list of `vm_area_struct` structures queued on the `mm_struct` for the process.

Once the process to swap has been located, the swap daemon looks through all of its virtual memory regions looking for areas which are not shared or locked. Linux does not swap out all of the swappable pages of the process that it has selected; instead it removes only a small number of pages. Pages cannot be swapped or discarded if they are locked in memory.

See `swap_out_vma()` in `mm/vmscan.c`

The Linux swap algorithm uses page aging. Each page has a counter (held in the `mem_map_t` data structure) that gives the Kernel swap daemon some idea whether or not a page is worth swapping. Pages age when they are unused and rejuvinate on access; the swap daemon only swaps out old pages. The default action when a page is first allocated, is to give it an initial age of 3. Each time it is touched, it's age is increased by 3 to a maximum of 20. Every time the Kernel swap daemon runs it ages pages, decrementing their age by 1. These default actions can be changed and for this reason they (and other swap related information) are stored in the `swap_control` data structure.

If the page is old ($age = 0$), the swap daemon will process it further. *Dirty* pages are pages which can be swapped out. Linux uses an architecture specific bit in the PTE to describe pages this way (see Figure 3.2). However, not all *dirty* pages are necessarily written to the swap file. Every virtual memory region of a process may have its own swap operation (pointed at by the `vm_ops` pointer in the `vm_area_struct`) and that method is used. Otherwise, the swap daemon will allocate a page in the swap file and write the page out to that device.

The page's page table entry is replaced by one which is marked as invalid but which contains information about where the page is in the swap file. This is an offset into the swap file where the page is held and an indication of which swap file is being used. Whatever the swap method used, the original physical page is made free by putting it back into the `free_area`. Clean (or rather not *dirty*) pages can be discarded and put back into the `free_area` for re-use.

If enough of the swappable processes pages have been swapped out or discarded, the swap daemon will again sleep. The next time it wakes it will consider the next process in the system. In this way, the swap daemon nibbles away at each processes physical pages until the system is again in balance. This is much fairer than swapping out whole processes.

3.9 The Swap Cache

When swapping pages out to the swap files, Linux avoids writing pages if it does not have to. There are times when a page is both in a swap file and in physical memory. This happens when a page that was swapped out of memory was then brought back into memory when it was again accessed by a process. So long as the page in memory is not written to, the copy in the swap file remains valid.

Linux uses the swap cache to track these pages. The swap cache is a list of page table entries, one per physical page in the system. This is a page table entry for a swapped out page and describes which swap file the page is being held in together with its location in the swap file. If a swap cache entry is non-zero, it represents a page which is being held in a swap file that has not been modified. If the page is subsequently modified (by being written to), its entry is removed from the swap cache.

When Linux needs to swap a physical page out to a swap file it consults the swap cache and, if there is a valid entry for this page, it does not need to write the page out to the swap file. This is because the page in memory has not been modified since it was last read from the swap file.

The entries in the swap cache are page table entries for swapped out pages. They are marked as invalid but contain information which allow Linux to find the right swap file and the right page within that swap file.

3.10 Swapping Pages In

The dirty pages saved in the swap files may be needed again, for example when an application writes to an area of virtual memory whose contents are held in a swapped out physical page. Accessing a page of virtual memory that is not held in physical memory causes a page fault to occur. The page fault is the processor signalling the operating system that it cannot translate a virtual address into a physical one. In this case this is because the page table entry describing this page of virtual memory was marked as invalid when the page was swapped out. The processor cannot handle the virtual to physical address translation and so hands control back to the operating system describing as it does so the virtual address that faulted and the reason for the fault. The format of this information and how the processor passes control to the operating system is processor specific. The processor specific page fault handling code must locate the `vm_area_struct` data structure that describes the area of virtual memory that contains the faulting virtual address. It does this by searching the `vm_area_struct` data structures for this process until it finds the one containing the faulting virtual address. This is very time critical code and a processes `vm_area_struct` data structures are so arranged as to make this search take as little time as possible.

See `do_page_fault()` in `arch/i386/mm/fault.c`

Having carried out the appropriate processor specific actions and found that the faulting virtual address is for a valid area of virtual memory, the page fault processing becomes generic and applicable to all processors that Linux runs on. The generic page fault handling code looks for the page table entry for the faulting virtual address. If the page table entry it finds is for a swapped out page, Linux must swap the page back into physical memory. The format of the page table entry for a swapped out page is processor specific but all processors mark these pages as invalid and put the information neccessary to locate the page within the swap file into the page table entry. Linux needs this information in order to bring the page back into physical memory.

See `do_no_page()` in `mm/memory.c`

At this point, Linux knows the faulting virtual address and has a page table entry containing information about where this page has been swapped to. The `vm_area_struct` data structure may contain a pointer to a routine which will swap any page of the area of virtual memory that it describes back into physical memory. This is its *swapin* operation. If there is a *swapin* operation for this area of virtual memory then Linux will use it. This is, in fact, how swapped out System V shared memory pages are handled as it requires special handling because the format of a swapped out System V shared page is a little different from that of an ordinairy

See `do_swap_page()` in `mm/memory.c`

See `shm_swap_in()` in `ipc/shm.c`

swapped out page. There may not be a *swapin* operation, in which case Linux will assume that this is an ordinairy page that does not need to be specially handled. It allocates a free physical page and reads the swapped out page back from the swap file. Information telling it where in the swap file (and which swap file) is taken from the the invalid page table entry.

See `swap_in()` in `mm/page_alloc.c`

If the access that caused the page fault was not a write access then the page is left in the swap cache and its page table entry is not marked as writable. If the page is subsequently written to, another page fault will occur and, at that point, the page is marked as dirty and its entry is removed from the swap cache. If the page is not written to and it needs to be swapped out again, Linux can avoid the write of the page to its swap file because the page is already in the swap file.

If the access that caused the page to be brought in from the swap file was a write operation, this page is removed from the swap cache and its page table entry is marked as both dirty and writable.

Chapter 4

Processes

This chapter describes what a process is and how the Linux kernel creates, manages and deletes the processes in the system.

Processes carry out tasks within the operating system. A program is a set of machine code instructions and data stored in an executable image on disk and is, as such, a passive entity; a process can be thought of as a computer program in action. It is a dynamic entity, constantly changing as the machine code instructions are executed by the processor. As well as the program's instructions and data, the process also includes the program counter and all of the CPU's registers as well as the process stacks containing temporary data such as routine parameters, return addresses and saved variables. The current executing program, or process, includes all of the current activity in the microprocessor. Linux is a multiprocessing operating system. Processes are separate tasks each with their own rights and responsibilities. If one process crashes it will not cause another process in the system to crash. Each individual process runs in its own virtual address space and is not capable of interacting with another process except through secure, kernel managed mechanisms.

During the lifetime of a process it will use many system resources. It will use the CPUs in the system to run its instructions and the system's physical memory to hold it and its data. It will open and use files within the filesystems and may directly or indirectly use the physical devices in the system. Linux must keep track of the process itself and of the system resources that it has so that it can manage it and the other processes in the system fairly. It would not be fair to the other processes in the system if one process monopolized most of the system's physical memory or its CPUs.

The most precious resource in the system is the CPU, usually there is only one. Linux is a multiprocessing operating system, its objective is to have a process running on each CPU in the system at all times, to maximize CPU utilization. If there are more processes than CPUs (and there usually are), the rest of the processes must wait before a CPU becomes free until they can be run. Multiprocessing is a simple idea; a process is executed until it must wait, usually for some system resource; when it has this resource, it may run again. In a uniprocessing system, for example DOS, the CPU would simply sit idle and the waiting time would be wasted. In a multiprocessing system many processes are kept in memory at the same time. Whenever a process has to wait the operating system takes the CPU away from that process and gives it to another, more deserving process. It is the scheduler which chooses which is the most appropriate process to run next and Linux uses a number of scheduling strategies to ensure fairness.

Linux supports a number of different executable file formats, ELF is one, Java is another and these must be managed transparently as must the processes use of the system's shared libraries.

4.1 Linux Processes

See `include/-linux/sched.h`

So that Linux can manage the processes in the system, each process is represented by a `task_struct` data structure (task and process are terms that Linux uses interchangeably). The `task` vector is an array of pointers to every `task_struct` data structure in the system. This means that the maximum number of processes in the system is limited by the size of the `task` vector; by default it has 512 entries. As processes are created, a new `task_struct` is allocated from system memory and added into the `task` vector. To make it easy to find, the current, running, process is pointed to by the `current` pointer.

As well as the normal type of process, Linux supports real time processes. These processes have to react very quickly to external events (hence the term "real time") and they are treated differently from normal user processes by the scheduler. Although the `task_struct` data structure is quite large and complex, but its fields can be divided into a number of functional areas:

State As a process executes it changes *state* according to its circumstances. Linux processes have the following states: [1]

- **Running** The process is either running (it is the current process in the system) or it is ready to run (it is waiting to be assigned to one of the system's CPUs).
- **Waiting** The process is waiting for an event or for a resource. Linux differentiates between two types of waiting process; *interruptible* and *uninterruptible*. Interruptible waiting processes can be interrupted by signals whereas uninterruptible waiting processes are waiting directly on hardware conditions and cannot be interrupted under any circumstances.
- **Stopped** The process has been stopped, usually by receiving a signal. A process that is being debugged can be in a stopped state.
- **Zombie** This is a halted process which, for some reason, still has a `task_struct` data structure in the `task` vector. It is what it sounds like, a dead process.

Scheduling Information The scheduler needs this information in order to fairly decide which process in the system most deserves to run,

[1] REVIEW NOTE: *I left out SWAPPING because it does not appear to be used.*

Identifiers Every process in the system has a process identifier. The process identifier is not an index into the `task` vector, it is simply a number. Each process also has User and group identifiers, these are used to control this processes access to the files and devices in the system,

Inter-Process Communication Linux supports the classic Unix™ IPC mechanisms of signals, pipes and semaphores and also the System V IPC mechanisms of shared memory, semaphores and message queues. The IPC mechanisms supported by Linux are described in Chapter 5.

Links In a Linux system no process is independent of any other process. Every process in the system, except the initial process has a parent process. New processes are not created, they are copied, or rather *cloned* from previous processes. Every `task_struct` representing a process keeps pointers to its parent process and to its siblings (those processes with the same parent process) as well as to its own child processes. You can see the family relationship between the running processes in a Linux system using the pstree command:

```
init(1)-+-crond(98)
        |-emacs(387)
        |-gpm(146)
        |-inetd(110)
        |-kerneld(18)
        |-kflushd(2)
        |-klogd(87)
        |-kswapd(3)
        |-login(160)---bash(192)---emacs(225)
        |-lpd(121)
        |-mingetty(161)
        |-mingetty(162)
        |-mingetty(163)
        |-mingetty(164)
        |-login(403)---bash(404)---pstree(594)
        |-sendmail(134)
        |-syslogd(78)
        `-update(166)
```

Additionally all of the processes in the system are held in a doubly linked list whose root is the `init` processes `task_struct` data structure. This list allows the Linux kernel to look at every process in the system. It needs to do this to provide support for commands such as ps or kill.

Times and Timers The kernel keeps track of a processes creation time as well as the CPU time that it consumes during its lifetime. Each clock tick, the kernel updates the amount of time in `jiffies` that the current process has spent in system and in user mode. Linux also supports process specific *interval* timers, processes can use system calls to set up timers to send signals to themselves when the timers expire. These timers can be single-shot or periodic timers.

File system Processes can open and close files as they wish and the processes `task_struct` contains pointers to descriptors for each open file as well as pointers to two VFS inodes. Each VFS inode uniquely describes a file or directory within a file system and also provides a uniform interface to the underlying file systems. How file systems are supported under Linux is described in Chapter 9. The first is to the root of the process (its home directory) and the second is to its current or *pwd* directory. *pwd* is derived from the Unix™ command pwd, *print working directory*. These two VFS inodes have their `count` fields incremented to show that one or more processes are referencing them. This is why you cannot delete the directory that a process has as its *pwd* directory set to, or for that matter one of its sub-directories.

Virtual memory Most processes have some virtual memory (kernel threads and daemons do not) and the Linux kernel must track how that virtual memory is mapped onto the system's physical memory.

Processor Specific Context A process could be thought of as the sum total of the system's current state. Whenever a process is running it is using the processor's registers, stacks and so on. This is the processes context and, when a process is suspended, all of that CPU specific context must be saved in the `task_struct` for the process. When a process is restarted by the scheduler its context is restored from here.

4.2 Identifiers

Linux, like all Unix™ uses user and group identifiers to check for access rights to files and images in the system. All of the files in a Linux system have ownerships and permissions, these permissions describe what access the system's users have to that file or directory. Basic permissions are *read*, *write* and *execute* and are assigned to three classes of user; the owner of the file, processes belonging to a particular group and all of the processes in the system. Each class of user can have different permissions, for example a file could have permissions which allow its owner to read and write it, the file's group to read it and for all other processes in the system to have no access at all.

REVIEW NOTE: *Expand and give the bit assignments (777).*

Groups are Linux's way of assigning privileges to files and directories for a group of users rather than to a single user or to all processes in the system. You might, for example, create a group for all of the users in a software project and arrange it so that only they could read and write the source code for the project. A process can belong to several groups (a maximum of 32 is the default) and these are held in the `groups` vector in the `task_struct` for each process. So long as a file has access rights for one of the groups that a process belongs to then that process will have appropriate group access rights to that file.

There are four pairs of process and group identifiers held in a processes `task_struct`:

uid, gid The user identifier and group identifier of the user that the process is running on behalf of,

effective uid and gid There are some programs which change the uid and gid from that of the executing process into their own (held as attributes in the VFS inode describing the executable image). These programs are known as *setuid* programs and they are useful because it is a way of restricting accesses to services, particularly those that run on behalf of someone else, for example a network daemon. The effective uid and gid are those from the setuid program and the uid and gid remain as they were. The kernel checks the effective uid and gid whenever it checks for privilege rights.

file system uid and gid These are normally the same as the effective uid and gid and are used when checking file system access rights. They are needed for NFS mounted filesystems where the user mode NFS server needs to access files as if it were a particular process. In this case only the file system uid and gid are changed (not the effective uid and gid). This avoids a situation where malicious users could send a kill signal to the NFS server. Kill signals are delivered to processes with a particular effective uid and gid.

saved uid and gid These are mandated by the POSIX standard and are used by programs which change the processes uid and gid via system calls. They are used to save the real uid and gid during the time that the original uid and gid have been changed.

4.3 Scheduling

All processes run partially in user mode and partially in system mode. How these modes are supported by the underlying hardware differs but generally there is a secure mechanism for getting from user mode into system mode and back again. User mode has far less privileges than system mode. Each time a process makes a system call it swaps from user mode to system mode and continues executing. At this point the kernel is executing on behalf of the process. In Linux, processes do not preempt the current, running process, they cannot stop it from running so that they can run. Each process decides to relinquish the CPU that it is running on when it has to wait for some system event. For example, a process may have to wait for a character to be read from a file. This waiting happens within the system call, in system mode; the process used a library function to open and read the file and it, in turn made system calls to read bytes from the open file. In this case the waiting process will be suspended and another, more deserving process will be chosen to run.

Processes are always making system calls and so may often need to wait. Even so, if a process executes until it waits then it still might use a disproportionate amount of CPU time and so Linux uses pre-emptive scheduling. In this scheme, each process is allowed to run for a small amount of time, 200ms, and, when this time has expired another process is selected to run and the original process is made to wait for a little while until it can run again. This small amount of time is known as a *time-slice*.

It is the *scheduler* that must select the most deserving process to run out of all of the runnable processes in the system. A runnable process is one which is waiting only for a CPU to run on. Linux uses a reasonably simple priority based scheduling algorithm to choose between the current processes in the system. When it has chosen a new process to run it saves the state of the current process, the processor specific registers and other context being saved in the processes `task_struct` data structure. It then restores the state of the new process (again this is processor specific) to run and gives control of the system to that process. For the scheduler to fairly allocate

See `schedule()` in `kernel/sched.c`

CPU time between the runnable processes in the system it keeps information in the `task_struct` for each process:

policy This is the scheduling policy that will be applied to this process. There are two types of Linux process, normal and real time. Real time processes have a higher priority than all of the other processes. If there is a real time process ready to run, it will always run first. Real time processes may have two types of `policy`, *round robin* and *first in first out*. In *round robin* scheduling, each runnable real time process is run in turn and in *first in, first out* scheduling each runnable process is run in the order that it is in on the run queue and that order is never changed.

priority This is the priority that the scheduler will give to this process. It is also the amount of time (in `jiffies`) that this process will run for when it is allowed to run. You can alter the priority of a process by means of system calls and the renice command.

rt_priority Linux supports real time processes and these are scheduled to have a higher priority than all of the other non-real time processes in system. This field allows the scheduler to give each real time process a relative priority. The priority of a real time processes can be altered using system calls.

counter This is the amount of time (in `jiffies`) that this process is allowed to run for. It is set to `priority` when the process is first run and is decremented each clock tick.

See `schedule()` in `kernel/sched.c`

The scheduler is run from several places within the kernel. It is run after putting the current process onto a wait queue and it may also be run at the end of a system call, just before a process is returned to process mode from system mode. One reason that it might need to run is because the system timer has just set the current processes `counter` to zero. Each time the scheduler is run it does the following:

kernel work The scheduler runs the bottom half handlers and processes the scheduler task queue. These lightweight kernel threads are described in detail in chapter 11.

Current process The current process must be processed before another process can be selected to run.

If the scheduling policy of the current processes is *round robin* then it is put onto the back of the run queue.

If the task is `INTERRUPTIBLE` and it has received a signal since the last time it was scheduled then its state becomes `RUNNING`.

If the current process has timed out, then its state becomes `RUNNING`.

If the current process is `RUNNING` then it will remain in that state.

Processes that were neither `RUNNING` nor `INTERRUPTIBLE` are removed from the run queue. This means that they will not be considered for running when the scheduler looks for the most deserving process to run.

Process selection The scheduler looks through the processes on the run queue looking for the most deserving process to run. If there are any real time processes (those with a real time scheduling policy) then those will get a higher weighting than ordinary processes. The weight for a normal process is its `counter` but for a real time process it is `counter` plus 1000. This means that if there are any runnable real time processes in the system then these will always be run before any normal runnable processes. The current process, which has consumed some of its time-slice (its `counter` has been decremented) is at a disadvantage if there are other processes with equal priority in the system; that is as it should be. If several processes have the same priority, the one nearest the front of the run queue is chosen. The current process will get put onto the back of the run queue. In a balanced system with many processes of the same priority, each one will run in turn. This is known as *Round Robin* scheduling. However, as processes wait for resources, their run order tends to get moved around.

Swap processes If the most deserving process to run is not the current process, then the current process must be suspended and the new one made to run. When a process is running it is using the registers and physical memory of the CPU and of the system. Each time it calls a routine it passes its arguments in registers and may stack saved values such as the address to return to in the calling routine. So, when the scheduler is running it is running in the context of the current process. It will be in a privileged mode, kernel mode, but it is still the current process that is running. When that process comes to be suspended, all of its machine state, including the program counter (PC) and all of the processor's registers, must be saved in the processes `task_struct` data structure. Then, all of the machine state for the new process must be loaded. This is a system dependent operation, no CPUs do this in quite the same way but there is usually some hardware assistance for this act.

This swapping of process context takes place at the end of the scheduler. The saved context for the previous process is, therefore, a snapshot of the hardware context of the system as it was for this process at the end of the scheduler. Equally, when the context of the new process is loaded, it too will be a snapshot of the way things were at the end of the scheduler, including this processes program counter and register contents.

If the previous process or the new current process uses virtual memory then the system's page table entries may need to be updated. Again, this action is architecture specific. Processors like the Alpha AXP, which use Translation Look-aside Tables or cached Page Table Entries, must flush those cached table entries that belonged to the previous process.

4.3.1 Scheduling in Multiprocessor Systems

Systems with multiple CPUs are reasonably rare in the Linux world but a lot of work has already gone into making Linux an SMP (Symmetric Multi-Processing) operating system. That is, one that is capable of evenly balancing work between the CPUs in the system. Nowhere is this balancing of work more apparent than in the scheduler.

In a multiprocessor system, hopefully, all of the processors are busily running processes. Each will run the scheduler separately as its current process exhausts its time-slice or has to wait for a system resource. The first thing to notice about an SMP system is that there is not just one idle process in the system. In a single

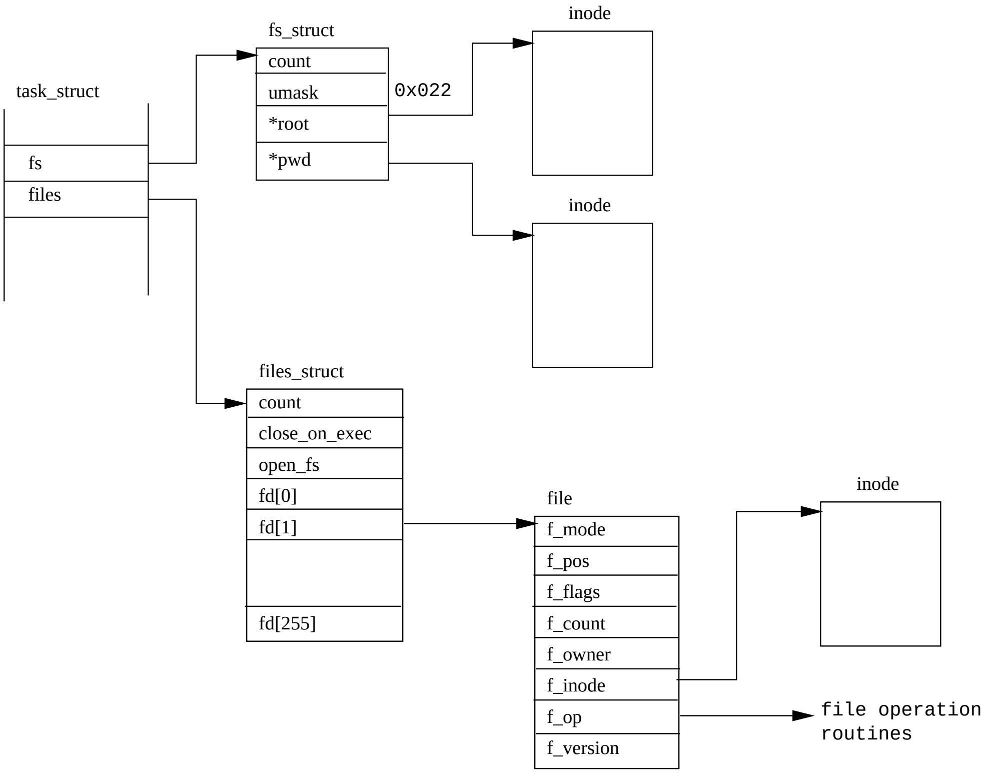

Figure 4.1: A Process's Files

processor system the idle process is the first task in the `task` vector, in an SMP system there is one idle process per CPU, and you could have more than one idle CPU. Additionally there is one current process per CPU, so SMP systems must keep track of the current and idle processes for each processor.

In an SMP system each process's `task_struct` contains the number of the processor that it is currently running on (`processor`) and its processor number of the last processor that it ran on (`last_processor`). There is no reason why a process should not run on a different CPU each time it is selected to run but Linux can restrict a process to one or more processors in the system using the `processor_mask`. If bit N is set, then this process can run on processor N. When the scheduler is choosing a new process to run it will not consider one that does not have the appropriate bit set for the current processor's number in its `processor_mask`. The scheduler also gives a slight advantage to a process that last ran on the current processor because there is often a performance overhead when moving a process to a different processor.

4.4 Files

See `include/-linux/sched.h`

Figure 4.1 shows that there are two data structures that describe file system specific information for each process in the system. The first, the `fs_struct` contains pointers to this process's VFS inodes and its `umask`. The `umask` is the default mode that new files will be created in, and it can be changed via system calls.

The second data structure, the `files_struct`, contains information about all of the files that this process is currently using. Programs read from *standard input* and write to *standard output*. Any error messages should go to *standard error*. These may be files, terminal input/output or a real device but so far as the program is concerned they are all treated as files. Every file has its own descriptor and the `files_struct` contains pointers to up to 256 `file` data structures, each one describing a file being used by this process. The `f_mode` field describes what mode the file has been created in; read only, read and write or write only. `f_pos` holds the position in the file where the next read or write operation will occur. `f_inode` points at the VFS inode describing the file and `f_ops` is a pointer to a vector of routine addresses; one for each function that you might wish to perform on a file. There is, for example, a write data function. This abstraction of the interface is very powerful and allows Linux to support a wide variety of file types. In Linux, pipes are implemented using this mechanism as we shall see later.

Every time a file is opened, one of the free `file` pointers in the `files_struct` is used to point to the new `file` structure. Linux processes expect three file descriptors to be open when they start. These are known as *standard input*, *standard output* and *standard error* and they are usually inherited from the creating parent process. All accesses to files are via standard system calls which pass or return file descriptors. These descriptors are indices into the process's `fd` vector, so *standard input*, *standard output* and *standard error* have file descriptors 0, 1 and 2. Each access to the file uses the `file` data structure's file operation routines to together with the VFS inode to achieve its needs.

4.5 Virtual Memory

A process's virtual memory contains executable code and data from many sources. First there is the program image that is loaded; for example a command like ls. This command, like all executable images, is composed of both executable code and data. The image file contains all of the information neccessary to load the executable code and associated program data into the virtual memory of the process. Secondly, processses can allocate (virtual) memory to use during their processing, say to hold the contents of files that it is reading. This newly allocated, virtual, memory needs to be linked into the process's existing virtual memory so that it can be used. Thirdly, Linux processes use libraries of commonly useful code, for example file handling routines. It does not make sense that each process has its own copy of the library, Linux uses shared libraries that can be used by several running processes at the same time. The code and the data from these shared libraries must be linked into this process's virtual address space and also into the virtual address space of the other processes sharing the library.

In any given time period a process will not have used all of the code and data contained within its virtual memory. It could contain code that is only used during certain situations, such as during initialization or to process a particular event. It may only have used some of the routines from its shared libraries. It would be wasteful to load all of this code and data into physical memory where it would lie unused. Multiply this wastage by the number of processes in the system and the system would run very inefficiently. Instead, Linux uses a technique called *demand paging* where the virtual memory of a process is brought into physical memory only when a process attempts to use it. So, instead of loading the code and data into physical memory straight away, the Linux kernel alters the process's page table,

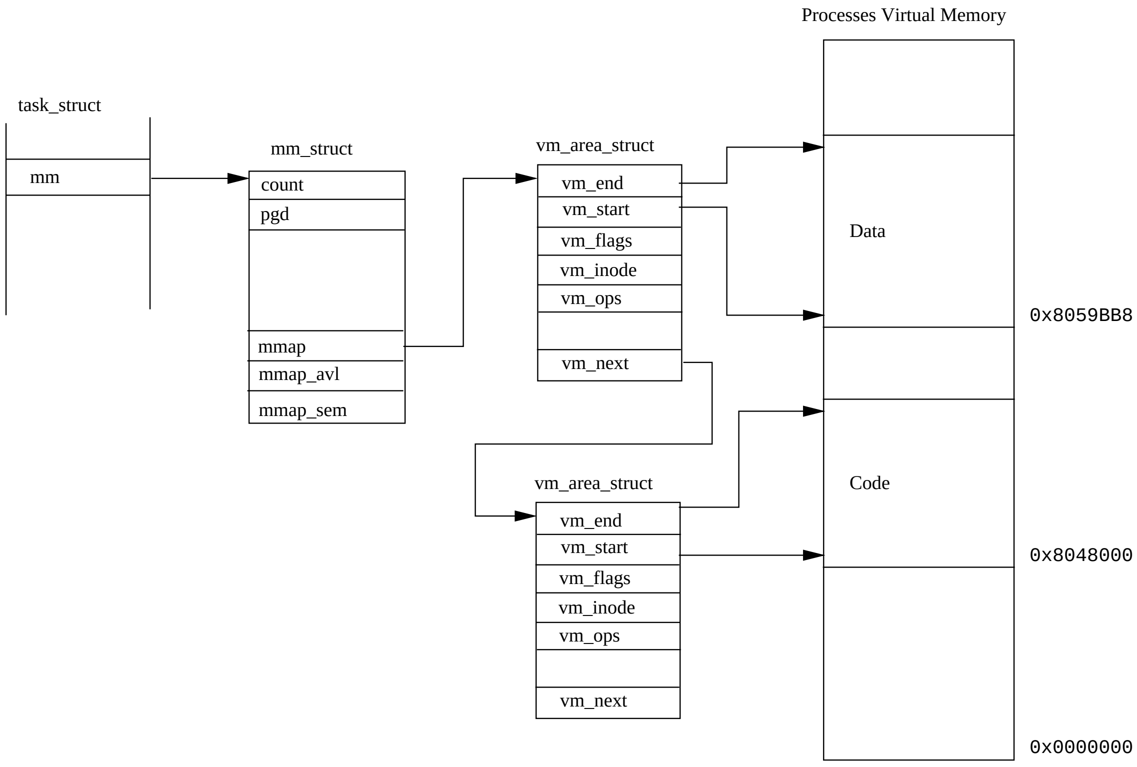

Figure 4.2: A Process's Virtual Memory

marking the virtual areas as existing but not in memory. When the process attempts to acccess the code or data the system hardware will generate a page fault and hand control to the Linux kernel to fix things up. Therefore, for every area of virtual memory in the process's address space Linux needs to know where that virtual memory comes from and how to get it into memory so that it can fix up these page faults.

The Linux kernel needs to manage all of these areas of virtual memory and the contents of each process's virtual memory is described by a `mm_struct` data structure pointed at from its `task_struct`. The process's `mm_struct` data structure also contains information about the loaded executable image and a pointer to the process's page tables. It contains pointers to a list of `vm_area_struct` data structures, each representing an area of virtual memory within this process.

This linked list is in ascending virtual memory order, figure 4.2 shows the layout in virtual memory of a simple process together with the kernel data structures managing it. As those areas of virtual memory are from several sources, Linux abstracts the interface by having the `vm_area_struct` point to a set of virtual memory handling routines (via `vm_ops`). This way all of the process's virtual memory can be handled in a consistent way no matter how the underlying services managing that memory differ. For example there is a routine that will be called when the process attempts to access the memory and it does not exist, this is how page faults are handled.

The process's set of `vm_area_struct` data structures is accessed repeatedly by the Linux kernel as it creates new areas of virtual memory for the process and as it fixes up references to virtual memory not in the system's physical memory. This makes the time that it takes to find the correct `vm_area_struct` critical to the performance of the system. To speed up this access, Linux also arranges the `vm_area_struct` data structures into an AVL (Adelson-Velskii and Landis) tree. This tree is arranged so that each `vm_area_struct` (or node) has a left and a right pointer to its neighbouring `vm_area_struct` structure. The left pointer points to node with a lower starting virtual address and the right pointer points to a node with a higher starting virtual address. To find the correct node, Linux goes to the root of the tree and follows each node's left and right pointers until it finds the right `vm_area_struct`. Of course, nothing is for free and inserting a new `vm_area_struct` into this tree takes additional processing time.

When a process allocates virtual memory, Linux does not actually reserve physical memory for the process. Instead, it describes the virtual memory by creating a new `vm_area_struct` data structure. This is linked into the process's list of virtual memory. When the process attempts to write to a virtual address within that new virtual memory region then the system will page fault. The processor will attempt to decode the virtual address, but as there are no Page Table Entries for any of this memory, it will give up and raise a page fault exception, leaving the Linux kernel to fix things up. Linux looks to see if the virtual address referenced is in the current process's virtual address space. If it is, Linux creates the appropriate PTEs and allocates a physical page of memory for this process. The code or data may need to be brought into that physical page from the filesystem or from the swap disk. The process can then be restarted at the instruction that caused the page fault and, this time as the memory physically exists, it may continue.

4.6 Creating a Process

When the system starts up it is running in kernel mode and there is, in a sense, only one process, the initial process. Like all processes, the initial process has a machine state represented by stacks, registers and so on. These will be saved in the initial process's `task_struct` data structure when other processes in the system are created and run. At the end of system initialization, the initial process starts up a kernel thread (called `init`) and then sits in an idle loop doing nothing. Whenever there is nothing else to do the scheduler will run this, idle, process. The idle process's `task_struct` is the only one that is not dynamically allocated, it is statically defined at kernel build time and is, rather confusingly, called `init_task`.

The `init` kernel thread or process has a process identifier of 1 as it is the system's first real process. It does some initial setting up of the system (such as opening the system console and mounting the root file system) and then executes the system initialization program. This is one of `/etc/init`, `/bin/init` or `/sbin/init` depending on your system. The `init` program uses `/etc/inittab` as a script file to create new processes within the system. These new processes may themselves go on to create new processes. For example the `getty` process may create a `login` process when a user attempts to login. All of the processes in the system are descended from the `init` kernel thread.

New processes are created by cloning old processes, or rather by cloning the current process. A new task is created by a system call (*fork* or *clone*) and the cloning happens within the kernel in kernel mode. At the end of the system call there is a new process waiting to run once the scheduler chooses it. A new `task_struct` data structure is allocated from the system's physical memory with one or more physical

See `do_fork()` in `kernel/fork.c`

pages for the cloned process's stacks (user and kernel). A new process identifier may be created, one that is unique within the set of process identifiers in the system. However, it is perfectly reasonable for the cloned process to keep its parents process identifier. The new `task_struct` is entered into the `task` vector and the contents of the old (`current`) process's `task_struct` are copied into the cloned `task_struct`.

When cloning processes Linux allows the two processes to share resources rather than have two separate copies. This applies to the process's files, signal handlers and virtual memory. When the resources are to be shared their respective `count` fields are incremented so that Linux will not deallocate these resources until both processes have finished using them. So, for example, if the cloned process is to share virtual memory, its `task_struct` will contain a pointer to the `mm_struct` of the original process and that `mm_struct` has its `count` field incremented to show the number of current processes sharing it.

Cloning a process's virtual memory is rather tricky. A new set of `vm_area_struct` data structures must be generated together with their owning `mm_struct` data structure and the cloned process's page tables. None of the process's virtual memory is copied at this point. That would be a rather difficult and lengthy task for some of that virtual memory would be in physical memory, some in the executable image that the process is currently executing and possibly some would be in the swap file. Instead Linux uses a technique called "copy on write" which means that virtual memory will only be copied when one of the two processes tries to write to it. Any virtual memory that is not written to, even if it can be, will be shared between the two processes without any harm occuring. The read only memory, for example the executable code, will always be shared. For "copy on write" to work, the writeable areas have their page table entries marked as read only and the `vm_area_struct` data structures describing them are marked as "copy on write". When one of the processes attempts to write to this virtual memory a page fault will occur. It is at this point that Linux will make a copy of the memory and fix up the two processes' page tables and virtual memory data structures.

4.7 Times and Timers

The kernel keeps track of a process's creation time as well as the CPU time that it consumes during its lifetime. Each clock tick, the kernel updates the amount of time in `jiffies` that the current process has spent in system and in user mode.

See `kernel/itimer.c`

In addition to these accounting timers, Linux supports process specific *interval* timers. A process can use these timers to send itself various signals each time that they expire. Three sorts of interval timers are supported:

Real the timer ticks in real time, and when the timer has expired, the process is sent a `SIGALRM` signal.

Virtual This timer only ticks when the process is running and when it expires it sends a `SIGVTALRM` signal.

Profile This timer ticks both when the process is running and when the system is executing on behalf of the process itself. `SIGPROF` is signalled when it expires.

One or all of the interval timers may be running and Linux keeps all of the neccessary information in the process's `task_struct` data structure. System calls can be made

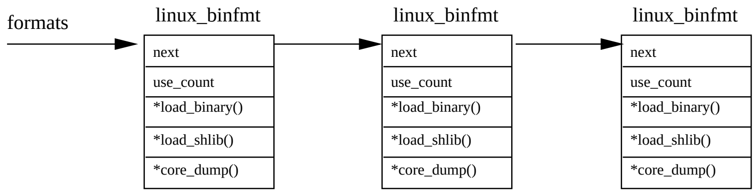

Figure 4.3: Registered Binary Formats

to set up these interval timers and to start them, stop them and read their current values. The virtual and profile timers are handled the same way. Every clock tick the current process's interval timers are decremented and, if they have expired, the appropriate signal is sent.

See `do_it_virtual()` in `kernel/sched.c`

See `do_it_prof()` in `kernel/sched.c`

Real time interval timers are a little different and for these Linux uses the timer mechanism described in Chapter 11. Each process has its own `timer_list` data structure and, when the real interval timer is running, this is queued on the system timer list. When the timer expires the timer bottom half handler removes it from the queue and calls the interval timer handler. This generates the `SIGALRM` signal and restarts the interval timer, adding it back into the system timer queue.

See `it_real_fn()` in `kernel/itimer.c`

4.8 Executing Programs

In Linux, as in Unix™, programs and commands are normally executed by a command interpreter. A command interpreter is a user process like any other process and is called a `shell` [2]. There are many shells in Linux, some of the most popular are `sh`, `bash` and `tcsh`. With the exception of a few built in commands, such as `cd` and `pwd`, a command is an executable binary file. For each command entered, the shell searches the directories in the process's *search path*, held in the `PATH` environment variable, for an executable image with a matching name. If the file is found it is loaded and executed. The shell clones itself using the *fork* mechanism described above and then the new child process replaces the binary image that it was executing, the shell, with the contents of the executable image file just found. Normally the shell waits for the command to complete, or rather for the child process to exit. You can cause the shell to run again by pushing the child process to the background by typing `control-Z`, which causes a `SIGSTOP` signal to be sent to the child process, stopping it. You then use the shell command `bg` to push it into a background, the shell sends it a `SIGCONT` signal to restart it, where it will stay until either it ends or it needs to do terminal input or output.

An executable file can have many formats or even be a script file. Script files have to be recognized and the appropriate interpreter run to handle them; for example `/bin/sh` interprets shell scripts. Executable object files contain executable code and data together with enough information to allow the operating system to load them into memory and execute them. The most commonly used object file format used by Linux is ELF but, in theory, Linux is flexible enough to handle almost any object file format.

[2]Think of a nut the kernel is the edible bit in the middle and the shell goes around it, providing an interface.

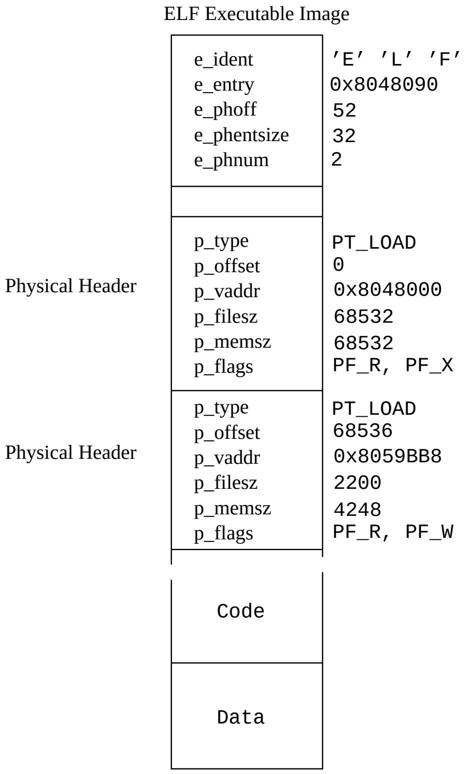

Figure 4.4: ELF Executable File Format

See `do_execve()` in `fs/exec.c`

As with file systems, the binary formats supported by Linux are either built into the kernel at kernel build time or available to be loaded as modules. The kernel keeps a list of supported binary formats (see figure 4.3) and when an attempt is made to execute a file, each binary format is tried in turn until one works. Commonly supported Linux binary formats are `a.out` and `ELF`. Executable files do not have to be read completely into memory, a technique known as demand loading is used. As each part of the executable image is used by a process it is brought into memory. Unused parts of the image may be discarded from memory.

4.8.1 ELF

The `ELF` (Executable and Linkable Format) object file format, designed by the Unix System Laboratories, is now firmly established as the most commonly used format in Linux. Whilst there is a slight performance overhead when compared with other object file formats such as `ECOFF` and `a.out`, ELF is felt to be more flexible. ELF executable files contain executable code, sometimes refered to as *text*, and *data*. Tables within the executable image describe how the program should be placed into the process's virtual memory. Statically linked images are built by the linker (ld), or link editor, into one single image containing all of the code and data needed to run this image. The image also specifies the layout in memory of this image and the address in the image of the first code to execute.

See `include/linux/elf.h`

Figure 4.4 shows the layout of a statically linked ELF executable image. It is a simple C program that prints "hello world" and then exits. The header describes it as an ELF image with two physical headers (`e_phnum` is 2) starting 52 bytes (`e_phoff`) from the start of the image file. The first physical header describes the executable code in the image. It goes at virtual address *0x8048000* and there is 65532 bytes of it. This is because it is a statically linked image which contains all of the library code for the `printf()` call to output "hello world". The entry point for the image, the first instruction for the program, is not at the start of the image but at virtual address *0x8048090* (`e_entry`). The code starts immediately after the second physical header. This physical header describes the data for the program and is to be loaded into virtual memory at address *0x8059BB8*. This data is both readable and writeable. You will notice that the size of the data in the file is 2200 bytes (`p_filesz`) whereas its size in memory is 4248 bytes. This because the first 2200 bytes contain pre-initialized data and the next 2048 bytes contain data that will be initialized by the executing code.

See `do_load_elf_binary()` in `fs/binfmt_elf.c`

When Linux loads an ELF executable image into the process's virtual address space, it does not actually load the image. It sets up the virtual memory data structures, the process's `vm_area_struct` tree and its page tables. When the program is executed page faults will cause the program's code and data to be fetched into physical memory. Unused portions of the program will never be loaded into memory. Once the ELF binary format loader is satisfied that the image is a valid ELF executable image it flushes the process's current executable image from its virtual memory. As this process is a cloned image (*all* processes are) this, old, image is the program that the parent process was executing, for example the command interpreter shell such as `bash`. This flushing of the old executable image discards the old virtual memory data structures and resets the process's page tables. It also clears away any signal handlers that were set up and closes any files that are open. At the end of the flush the process is ready for the new executable image. No matter what format the executable image is, the same information gets set up in the process's `mm_struct`. There are pointers to the start and end of the image's code and data. These values are found as the ELF executable images physical headers are read and the sections of the program that they describe are mapped into the process's virtual address space. That is also when the `vm_area_struct` data structures are set up and the process's page tables are modified. The `mm_struct` data structure also contains pointers to the parameters to be passed to the program and to this process's environment variables.

ELF Shared Libraries

A dynamically linked image, on the other hand, does not contain all of the code and data required to run. Some of it is held in shared libraries that are linked into the image at run time. The ELF shared library's tables are also used by the *dynamic linker* when the shared library is linked into the image at run time. Linux uses several dynamic linkers, `ld.so.1`, `libc.so.1` and `ld-linux.so.1`, all to be found in `/lib`. The libraries contain commonly used code such as language subroutines. Without dynamic linking, all programs would need their own copy of the these libraries and would need far more disk space and virtual memory. In dynamic linking, information is included in the ELF image's tables for every library routine referenced. The information indicates to the dynamic linker how to locate the library routine and link it into the program's address space.

REVIEW NOTE: *Do I need more detail here, worked example?*

4.8.2 Script Files

Script files are executables that need an interpreter to run them. There are a wide variety of interpreters available for Linux; for example wish, perl and command shells such as tcsh. Linux uses the standard Unux™ convention of having the first line of a script file contain the name of the interpreter. So, a typical script file would start:

```
#!/usr/bin/wish
```

See do_load_script() in fs/-binfmt_script.c

The script binary loader tries to find the intepreter for the script. It does this by attempting to open the executable file that is named in the first line of the script. If it can open it, it has a pointer to its VFS inode and it can go ahead and have it interpret the script file. The name of the script file becomes argument zero (the first argument) and all of the other arguments move up one place (the original first argument becomes the new second argument and so on). Loading the interpreter is done in the same way as Linux loads all of its executable files. Linux tries each binary format in turn until one works. This means that you could in theory stack several interpreters and binary formats making the Linux binary format handler a very flexible piece of software.

Chapter 5

Interprocess Communication Mechanisms

Processes communicate with each other and with the kernel to coordinate their activities. Linux supports a number of Inter-Process Communication (IPC) mechanisms. Signals and pipes are two of them but Linux also supports the System V IPC mechanisms named after the Unix™ release in which they first appeared.

5.1 Signals

Signals are one of the oldest inter-process communication methods used by Unix™ systems. They are used to signal asynchronous events to one or more processes. A signal could be generated by a keyboard interrupt or an error condition such as the process attempting to access a non-existent location in its virtual memory. Signals are also used by the shells to signal job control commands to their child processes.

There are a set of defined signals that the kernel can generate or that can be generated by other processes in the system, provided that they have the correct privileges. You can list a system's set of signals using the kill command (kill -l), on my Intel Linux box this gives:

```
 1) SIGHUP  2) SIGINT  3) SIGQUIT  4) SIGILL
 5) SIGTRAP  6) SIGIOT  7) SIGBUS  8) SIGFPE
 9) SIGKILL 10) SIGUSR1 11) SIGSEGV 12) SIGUSR2
13) SIGPIPE 14) SIGALRM 15) SIGTERM 17) SIGCHLD
18) SIGCONT 19) SIGSTOP 20) SIGTSTP 21) SIGTTIN
22) SIGTTOU 23) SIGURG 24) SIGXCPU 25) SIGXFSZ
26) SIGVTALRM 27) SIGPROF 28) SIGWINCH 29) SIGIO
30) SIGPWR
```

The numbers are different for an Alpha AXP Linux box. Processes can choose to ignore most of the signals that are generated, with two notable exceptions: neither the `SIGSTOP` signal which causes a process to halt its execution nor the `SIGKILL` signal which causes a process to exit can be ignored. Otherwise though, a process can choose just how it wants to handle the various signals. Processes can block the signals and, if they do not block them, they can either choose to handle them themselves or allow the kernel to handle them. If the kernel handles the signals, it will do the default actions required for this signal. For example, the default action when a process receives the `SIGFPE` (floating point exception) signal is to core dump and then exit. Signals have no inherent relative priorities. If two signals are generated for a process at the same time then they may be presented to the process or handled in any order. Also there is no mechanism for handling multiple signals of the same kind. There is no way that a process can tell if it received 1 or 42 `SIGCONT` signals.

Linux implements signals using information stored in the `task_struct` for the process. The number of supported signals is limited to the word size of the processor. Processes with a word size of 32 bits can have 32 signals whereas 64 bit processors like the Alpha AXP may have up to 64 signals. The currently pending signals are kept in the `signal` field with a mask of blocked signals held in `blocked`. With the exception of `SIGSTOP` and `SIGKILL`, all signals can be blocked. If a blocked signal is generated, it remains pending until it is unblocked. Linux also holds information about how each process handles every possible signal and this is held in an array of `sigaction` data structures pointed at by the `task_struct` for each process. Amongst other things it contains either the address of a routine that will handle the signal or a flag which tells Linux that the process either wishes to ignore this signal or let the kernel handle the signal for it. The process modifies the default signal handling by making system calls and these calls alter the `sigaction` for the appropriate signal as well as the `blocked` mask.

Not every process in the system can send signals to every other process, the kernel can and super users can. Normal processes can only send signals to processes with the same *uid* and *gid* or to processes in the same process group[1]. Signals are generated by setting the appropriate bit in the `task_struct`'s `signal` field. If the process has not blocked the signal and is waiting but interruptible (in state Interruptible) then it is woken up by changing its state to Running and making sure that it is in the run queue. That way the scheduler will consider it a candidate for running when the system next schedules. If the default handling is needed, then Linux can optimize the handling of the signal. For example if the signal `SIGWINCH` (the X window changed focus) and the default handler is being used then there is nothing to be done.

Signals are not presented to the process immediately they are generated., they must wait until the process is running again. Every time a process exits from a system call its `signal` and `blocked` fields are checked and, if there are any unblocked signals, they can now be delivered. This might seem a very unreliable method but every process in the system is making system calls, for example to write a character to the terminal, all of the time. Processes can elect to wait for signals if they wish, they are suspended in state Interruptible until a signal is presented. The Linux signal processing code looks at the `sigaction` structure for each of the current unblocked signals.

If a signal's handler is set to the default action then the kernel will handle it. The

[1] REVIEW NOTE: *Explain process groups.*

SIGSTOP signal's default handler will change the current process's state to Stopped and then run the scheduler to select a new process to run. The default action for the SIGFPE signal will core dump the process and then cause it to exit. Alternatively, the process may have specfied its own signal handler. This is a routine which will be called whenever the signal is generated and the `sigaction` structure holds the address of this routine. The kernel must call the process's signal handling routine and how this happens is processor specific but all CPUs must cope with the fact that the current process is running in kernel mode and is just about to return to the process that called the kernel or system routine in user mode. The problem is solved by manipulating the stack and registers of the process. The process's program counter is set to the address of its signal handling routine and the parameters to the routine are added to the call frame or passed in registers. When the process resumes operation it appears as if the signal handling routine were called normally.

Linux is POSIX compatible and so the process can specify which signals are blocked when a particular signal handling routine is called. This means changing the `blocked` mask during the call to the processes signal handler. The `blocked` mask must be returned to its original value when the signal handling routine has finished. Therefore Linux adds a call to a tidy up routine which will restore the original `blocked` mask onto the call stack of the signalled process. Linux also optimizes the case where several signal handling routines need to be called by stacking them so that each time one handling routine exits, the next one is called until the tidy up routine is called.

5.2 Pipes

The common Linux shells all allow redirection. For example

```
$ ls | pr | lpr
```

pipes the output from the ls command listing the directory's files into the standard input of the pr command which paginates them. Finally the standard output from the pr command is piped into the standard input of the lpr command which prints the results on the default printer. Pipes then are unidirectional byte streams which connect the standard output from one process into the standard input of another process. Neither process is aware of this redirection and behaves just as it would normally. It is the shell which sets up these temporary pipes between the processes.

In Linux, a pipe is implemented using two `file` data structures which both point at the same temporary VFS inode which itself points at a physical page within memory. Figure 5.1 shows that each `file` data structure contains pointers to different file operation routine vectors; one for writing to the pipe, the other for reading from the pipe. This hides the underlying differences from the generic system calls which read and write to ordinary files. As the writing process writes to the pipe, bytes are copied into the shared data page and when the reading process reads from the pipe, bytes are copied from the shared data page. Linux must synchronize access to the pipe. It must make sure that the reader and the writer of the pipe are in step and to do this it uses locks, wait queues and signals.

See `include/linux/-inode_fs_i.h`

When the writer wants to write to the pipe it uses the standard write library functions. These all pass file descriptors that are indices into the process's set of `file`

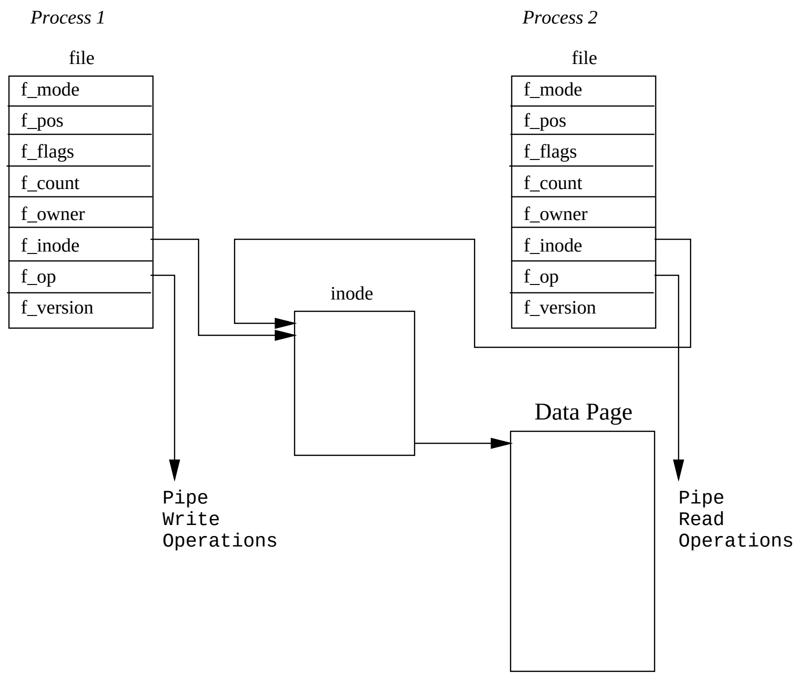

Figure 5.1: Pipes

data structures, each one representing an open file or, as in this case, an open pipe. The Linux system call uses the write routine pointed at by the `file` data structure describing this pipe. That write routine uses information held in the VFS inode representing the pipe to manage the write request. If there is enough room to write all of the bytes into the pipe and, so long as the pipe is not locked by its reader, Linux locks it for the writer and copies the bytes to be written from the process's address space into the shared data page. If the pipe is locked by the reader or if there is not enough room for the data then the current process is made to sleep on the pipe inode's wait queue and the scheduler is called so that another process can run. It is interruptible, so it can receive signals and it will be woken by the reader when there is enough room for the write data or when the pipe is unlocked. When the data has been written, the pipe's VFS inode is unlocked and any waiting readers sleeping on the inode's wait queue will themselves be woken up.

See `pipe_write()` in `fs/pipe.c`

Reading data from the pipe is a very similar process to writing to it. Processes are allowed to do non-blocking reads (it depends on the mode in which they opened the file or pipe) and, in this case, if there is no data to be read or if the pipe is locked, an error will be returned. This means that the process can continue to run. The alternative is to wait on the pipe inode's wait queue until the write process has finished. When both processes have finished with the pipe, the pipe inode is discarded along with the shared data page.

See `pipe_read()` in `fs/pipe.c`

Linux also supports *named* pipes, also known as FIFOs because pipes operate on a First In, First Out principle. The first data written into the pipe is the first data read from the pipe. Unlike pipes, FIFOs are not temporary objects, they are entities in the file system and can be created using the mkfifo command. Processes are free to use a FIFO so long as they have appropriate access rights to it. The way that FIFOs are opened is a little different from pipes. A pipe (its two `file` data structures, its

VFS inode and the shared data page) is created in one go whereas a FIFO already exists and is opened and closed by its users. Linux must handle readers opening the FIFO before writers open it as well as readers reading before any writers have written to it. That aside, FIFOs are handled almost exactly the same way as pipes and they use the same data structures and operations.

5.3 Sockets

REVIEW NOTE: *Add when networking chapter written.*

5.3.1 System V IPC Mechanisms

Linux supports three types of interprocess communication mechanisms that first appeared in Unix™ System V (1983). These are message queues, semaphores and shared memory. These System V IPC mechanisms all share common authentication methods. Processes may access these resources only by passing a unique reference identifier to the kernel via system calls. Access to these System V IPC objects is checked using access permissions, much like accesses to files are checked. The access rights to the System V IPC object is set by the creator of the object via system calls. The object's reference identifier is used by each mechanism as an index into a table of resources. It is not a straight forward index but requires some manipulation to generate the index.

All Linux data structures representing System V IPC objects in the system include an `ipc_perm` structure which contains the owner and creator process's user and group identifiers. The access mode for this object (owner, group and other) and the IPC object's key. The key is used as a way of locating the System V IPC object's reference identifier. Two sets of keys are supported: public and private. If the key is public then any process in the system, subject to rights checking, can find the reference identifier for the System V IPC object. System V IPC objects can never be referenced with a key, only by their reference identifier.

See `include/-linux/ipc.h`

5.3.2 Message Queues

Message queues allow one or more processes to write messages, which will be read by one or more reading processes. Linux maintains a list of message queues, the `msgque` vector; each element of which points to a `msqid_ds` data structure that fully describes the message queue. When message queues are created a new `msqid_ds` data structure is allocated from system memory and inserted into the vector.

Each `msqid_ds` data structure contains an `ipc_perm` data structure and pointers to the messages entered onto this queue. In addition, Linux keeps queue modification times such as the last time that this queue was written to and so on. The `msqid_ds` also contains two wait queues; one for the writers to the queue and one for the readers of the message queue.

See `include/-linux/msg.h`

Each time a process attempts to write a message to the write queue its effective user and group identifiers are compared with the mode in this queue's `ipc_perm` data structure. If the process can write to the queue then the message may be copied from the process's address space into a `msg` data structure and put at the end of this message queue. Each message is tagged with an application specific type, agreed

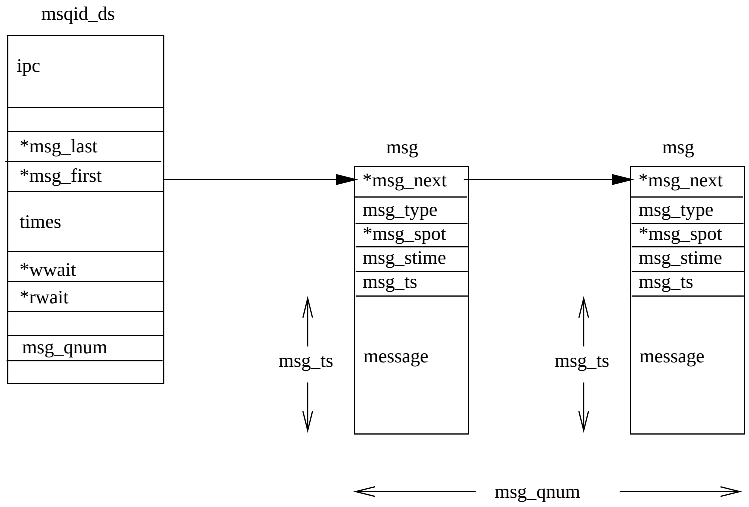

Figure 5.2: System V IPC Message Queues

between the cooperating processes. However, there may be no room for the message as Linux restricts the number and length of messages that can be written. In this case the process will be added to this message queue's write wait queue and the scheduler will be called to select a new process to run. It will be woken up when one or more messages have been read from this message queue.

Reading from the queue is a similar process. Again, the processes access rights to the write queue are checked. A reading process may choose to either get the first message in the queue regardless of its type or select messages with particular types. If no messages match this criteria the reading process will be added to the message queue's read wait queue and the scheduler run. When a new message is written to the queue this process will be woken up and run again.

5.3.3 Semaphores

In its simplest form a semaphore is a location in memory whose value can be tested and set by more than one process. The test and set operation is, so far as each process is concerned, uninterruptible or atomic; once started nothing can stop it. The result of the test and set operation is the addition of the current value of the semaphore and the set value, which can be positive or negative. Depending on the result of the test and set operation one process may have to sleep until the semphore's value is changed by another process. Semaphores can be used to implement *critical regions*, areas of critical code that only one process at a time should be executing.

Say you had many cooperating processes reading records from and writing records to a single data file. You would want that file access to be strictly coordinated. You could use a semaphore with an initial value of 1 and, around the file operating code, put two semaphore operations, the first to test and decrement the semaphore's value and the second to test and increment it. The first process to access the file would try to decrement the semaphore's value and it would succeed, the semaphore's value now being 0. This process can now go ahead and use the data file but if another process wishing to use it now tries to decrement the semaphore's value it would fail as the result would be -1. That process will be suspended until the first process has

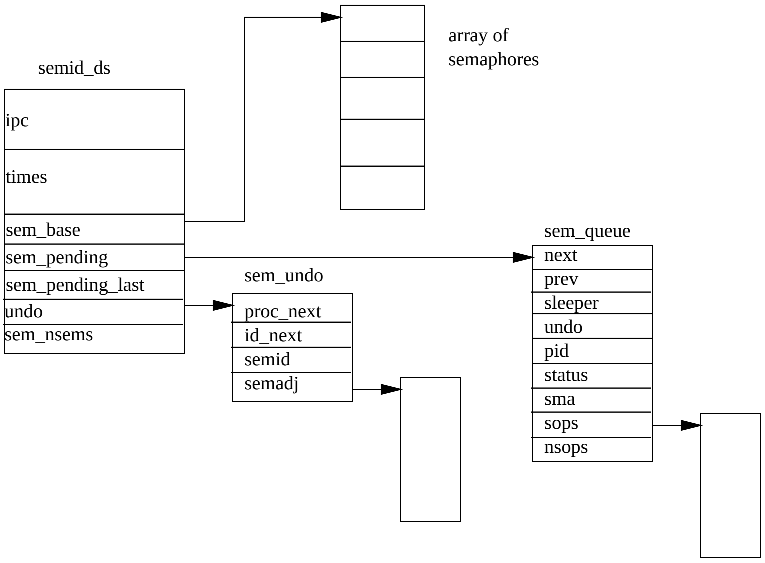

Figure 5.3: System V IPC Semaphores

finished with the data file. When the first process has finished with the data file it will increment the semaphore's value, making it 1 again. Now the waiting process can be woken and this time its attempt to increment the semaphore will succeed.

System V IPC semaphore objects each describe a semaphore array and Linux uses the `semid_ds` data structure to represent this. All of the `semid_ds` data structures in the system are pointed at by the `semary`, a vector of pointers. There are `sem_nsems` in each semaphore array, each one described by a `sem` data structure pointed at by `sem_base`. All of the processes that are allowed to manipulate the semaphore array of a System V IPC semaphore object may make system calls that perform operations on them. The system call can specify many operations and each operation is described by three inputs; the semaphore index, the operation value and a set of flags. The semaphore index is an index into the semaphore array and the operation value is a numerical value that will be added to the current value of the semaphore. First Linux tests whether or not all of the operations would succeed. An operation will succeed if the operation value added to the semaphore's current value would be greater than zero or if both the operation value and the semaphore's current value are zero. If any of the semaphore operations would fail Linux may suspend the process but only if the operation flags have not requested that the system call is non-blocking. If the process is to be suspended then Linux must save the state of the semaphore operations to be performed and put the current process onto a wait queue. It does this by building a `sem_queue` data structure on the stack and filling it out. The new `sem_queue` data structure is put at the end of this semaphore object's wait queue (using the `sem_pending` and `sem_pending_last` pointers). The current process is put on the wait queue in the `sem_queue` data structure (`sleeper`) and the scheduler called to choose another process to run.

See `include/-linux/sem.h`

If all of the semaphore operations would have succeeded and the current process does not need to be suspended, Linux goes ahead and applies the operations to the appropriate members of the semaphore array. Now Linux must check that any waiting, suspended, processes may now apply their semaphore operations. It looks at each member of the operations pending queue (`sem_pending`) in turn, testing to see if the semphore operations will succeed this time. If they will then it removes the `sem_queue` data structure from the operations pending list and applies the semaphore operations to the semaphore array. It wakes up the sleeping process making it available to be restarted the next time the scheduler runs. Linux keeps looking through the pending list from the start until there is a pass where no semaphore operations can be applied and so no more processes can be woken.

There is a problem with semaphores, *deadlocks*. These occur when one process has altered the semaphores value as it enters a critical region but then fails to leave the critical region because it crashed or was killed. Linux protects against this by maintaining lists of adjustments to the semaphore arrays. The idea is that when these adjustments are applied, the semaphores will be put back to the state that they were in before the a process's set of semaphore operations were applied. These adjustments are kept in `sem_undo` data structures queued both on the `semid_ds` data structure and on the `task_struct` data structure for the processes using these semaphore arrays.

Each individual semaphore operation may request that an adjustment be maintained. Linux will maintain at most one `sem_undo` data structure per process for each semaphore array. If the requesting process does not have one, then one is created when it is needed. The new `sem_undo` data structure is queued both onto this process's `task_struct` data structure and onto the semaphore array's `semid_ds` data structure. As operations are applied to the semphores in the semaphore array the negation of the operation value is added to this semphore's entry in the adjustment array of this process's `sem_undo` data structure. So, if the operation value is 2, then -2 is added to the adjustment entry for this semaphore.

When processes are deleted, as they exit Linux works through their set of `sem_undo` data structures applying the adjustments to the semaphore arrays. If a semaphore set is deleted, the `sem_undo` data structures are left queued on the process's `task_struct` but the semaphore array identifier is made invalid. In this case the semaphore clean up code simply discards the `sem_undo` data structure.

5.3.4 Shared Memory

Shared memory allows one or more processes to communicate via memory that appears in all of their virtual address spaces. The pages of the virtual memory is referenced by page table entries in each of the sharing processes' page tables. It does not have to be at the same address in all of the processes' virtual memory. As with all System V IPC objects, access to shared memory areas is controlled via keys and access rights checking. Once the memory is being shared, there are no checks on how the processes are using it. They must rely on other mechanisms, for example System V semaphores, to synchronize access to the memory.

See `include/-linux/sem.h`

Each newly created shared memory area is represented by a `shmid_ds` data structure. These are kept in the `shm_segs` vector. The `shmid_ds` data structure decribes how big the area of shared memory is, how many processes are using it and information about how that shared memory is mapped into their address spaces. It is the creator of the shared memory that controls the access permissions to that memory and whether its key is public or private. If it has enough access rights it may also lock the shared memory into physical memory.

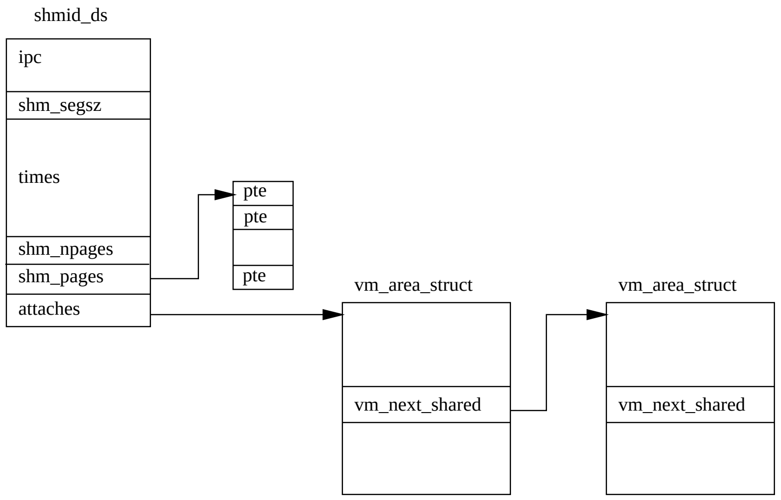

Figure 5.4: System V IPC Shared Memory

Each process that wishes to share the memory must attach to that virtual memory via a system call. This creates a new `vm_area_struct` data structure describing the shared memory for this process. The process can choose where in its virtual address space the shared memory goes or it can let Linux choose a free area large enough. The new `vm_area_struct` structure is put into the list of `vm_area_struct` pointed at by the `shmid_ds`. The `vm_next_shared` and `vm_prev_shared` pointers are used to link them together. The virtual memory is not actually created during the attach; it happens when the first process attempts to access it.

The first time that a process accesses one of the pages of the shared virtual memory, a page fault will occur. When Linux fixes up that page fault it finds the `vm_area_struct` data structure describing it. This contains pointers to handler routines for this type of shared virtual memory. The shared memory page fault handling code looks in the list of page table entries for this `shmid_ds` to see if one exists for this page of the shared virtual memory. If it does not exist, it will allocate a physical page and create a page table entry for it. As well as going into the current process's page tables, this entry is saved in the `shmid_ds`. This means that when the next process that attempts to access this memory gets a page fault, the shared memory fault handling code will use this newly created physical page for that process too. So, the first process that accesses a page of the shared memory causes it to be created and thereafter access by the other processes cause that page to be added into their virtual address spaces.

When processes no longer wish to share the virtual memory, they detach from it. So long as other processes are still using the memory the detach only affects the current process. Its `vm_area_struct` is removed from the `shmid_ds` data structure and deallocated. The current process's page tables are updated to invalidate the area of virtual memory that it used to share. When the last process sharing the memory detaches from it, the pages of the shared memory current in physical memory are freed, as is the `shmid_ds` data structure for this shared memory.

Further complications arise when shared virtual memory is not locked into physical memory. In this case the pages of the shared memory may be swapped out to the system's swap disk during periods of high memory usage. How shared memory

memory is swapped into and out of physical memory is described in Chapter 3.

Chapter 6

PCI

Peripheral Component Interconnect (PCI), as its name implies is a standard that describes how to connect the peripheral components of a system together in a structured and controlled way. The standard[3, PCI Local Bus Specification] describes the way that the system components are electrically connected and the way that they should behave. This chapter looks at how the Linux kernel initializes the system's PCI buses and devices.

Figure 6.1 is a logical diagram of an example PCI based system. The PCI buses and PCI-PCI bridges are the glue connecting the system components together; the CPU is connected to PCI bus 0, the primary PCI bus as is the video device. A special PCI device, a PCI-PCI bridge connects the primary bus to the secondary PCI bus, PCI bus 1. In the jargon of the PCI specification, PCI bus 1 is described as being *downstream* of the PCI-PCI bridge and PCI bus 0 is *up-stream* of the bridge. Connected to the secondary PCI bus are the SCSI and ethernet devices for the system. Physically the bridge, secondary PCI bus and two devices would all be contained on the same combination PCI card. The PCI-ISA bridge in the system supports older, legacy ISA devices and the diagram shows a super I/O controller chip, which controls the keyboard, mouse and floppy. [1]

6.1 PCI Address Spaces

The CPU and the PCI devices need to access memory that is shared between them. This memory is used by device drivers to control the PCI devices and to pass information between them. Typically the shared memory contains control and status registers for the device. These registers are used to control the device and to read its status. For example, the PCI SCSI device driver would read its status register to find out if the SCSI device was ready to write a block of information to the SCSI

[1] For example?

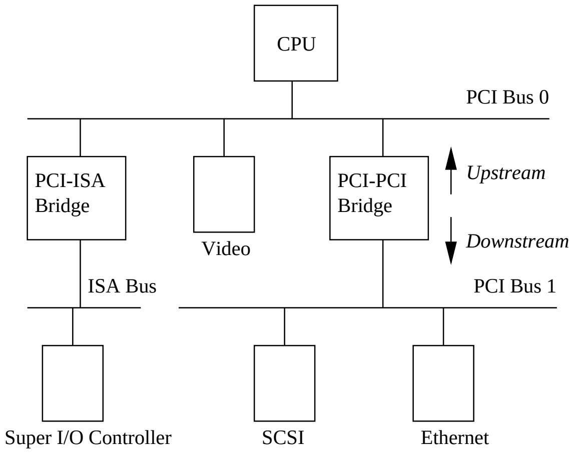

Figure 6.1: Example PCI Based System

disk. Or it might write to the control register to start the device running after it has been turned on.

The CPU's system memory could be used for this shared memory but if it were, then every time a PCI device accessed memory, the CPU would have to stall, waiting for the PCI device to finish. Access to memory is generally limited to one system component at a time. This would slow the system down. It is also not a good idea to allow the system's peripheral devices to access main memory in an uncontrolled way. This would be very dangerous; a rogue device could make the system very unstable.

Peripheral devices have their own memory spaces. The CPU can access these spaces but access by the devices into the system's memory is very strictly controlled using DMA (Direct Memory Access) channels. ISA devices have access to two address spaces, ISA I/O (Input/Output) and ISA memory. PCI has three; PCI I/O, PCI Memory and PCI Configuration space. All of these address spaces are also accessible by the CPU with the the PCI I/O and PCI Memory address spaces being used by the device drivers and the PCI Configuration space being used by the PCI initialization code within the Linux kernel.

The Alpha AXP processor does not have natural access to addresses spaces other than the system address space. It uses support chipsets to access other address spaces such as PCI Configuration space. It uses a sparse address mapping scheme which steals part of the large virtual address space and maps it to the PCI address spaces.

6.2 PCI Configuration Headers

Every PCI device in the system, including the PCI-PCI bridges has a configuration data structure that is somewhere in the PCI configuration address space. The PCI Configuration header allows the system to identify and control the device. Exactly where the header is in the PCI Configuration address space depends on where in the PCI topology that device is. For example, a PCI video card plugged into one PCI slot on the PC motherboard will have its configuration header at one location and if it is plugged into another PCI slot then its header will appear in another

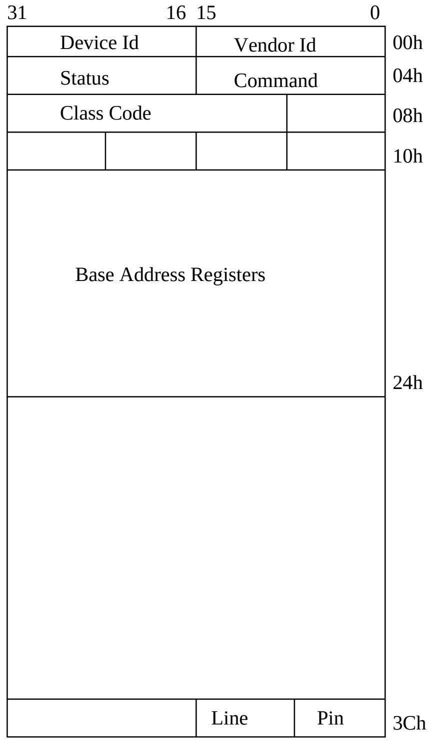

Figure 6.2: The PCI Configuration Header

location in PCI Configuration memory. This does not matter, for wherever the PCI devices and bridges are the system will find and configure them using the status and configuration registers in their configuration headers.

Typically, systems are designed so that every PCI slot has it's PCI Configuration Header in an offset that is related to its slot on the board. So, for example, the first slot on the board might have its PCI Configuration at offset 0 and the second slot at offset 256 (all headers are the same length, 256 bytes) and so on. A system specific hardware mechanism is defined so that the PCI configuration code can attempt to examine all possible PCI Configuration Headers for a given PCI bus and know which devices are present and which devices are absent simply by trying to read one of the fields in the header (usually the *Vendor Identification* field) and getting some sort of error. The [3, PCI Local Bus Specification] describes one possible error message as returning *0xFFFFFFFF* when attempting to read the *Vendor Identification* and *Device Identification* fields for an empty PCI slot.

Figure 6.2 shows the layout of the 256 byte PCI configuration header. It contains the following fields:

See `include/-linux/pci.h`

Vendor Identification A unique number describing the originator of the PCI device. Digital's PCI Vendor Identification is *0x1011* and Intel's is *0x8086*.

Device Identification A unique number describing the device itself. For example, Digital's 21141 fast ethernet device has a device identification of *0x0009*.

Status This field gives the status of the device with the meaning of the bits of this field set by the standard. [3, PCI Local Bus Specification].

Command By writing to this field the system controls the device, for example allowing the device to access PCI I/O memory,

Class Code This identifies the type of device that this is. There are standard classes for every sort of device; video, SCSI and so on. The class code for SCSI is *0x0100*.

Base Address Registers These registers are used to determine and allocate the type, amount and location of PCI I/O and PCI memory space that the device can use.

Interrupt Pin Four of the physical pins on the PCI card carry interrupts from the card to the PCI bus. The standard labels these as A, B, C and D. The *Interrupt Pin* field describes which of these pins this PCI device uses. Generally it is hardwired for a pariticular device. That is, every time the system boots, the device uses the same interrupt pin. This information allows the interrupt handling subsystem to manage interrupts from this device,

Interrupt Line The *Interrupt Line* field of the device's PCI Configuration header is used to pass an interrupt handle between the PCI initialisation code, the device's driver and Linux's interrupt handling subsystem. The number written there is meaningless to the the device driver but it allows the interrupt handler to correctly route an interrupt from the PCI device to the correct device driver's interrupt handling code within the Linux operating system. See Chapter 7 on page 75 for details on how Linux handles interrupts.

6.3 PCI I/O and PCI Memory Addresses

These two address spaces are used by the devices to communicate with their device drivers running in the Linux kernel on the CPU. For example, the DECchip 21141 fast ethernet device maps its internal registers into PCI I/O space. Its Linux device driver then reads and writes those registers to control the device. Video drivers typically use large amounts of PCI memory space to contain video information.

Until the PCI system has been set up and the device's access to these address spaces has been turned on using the *Command* field in the PCI Configuration header, nothing can access them. It should be noted that only the PCI configuration code reads and writes PCI configuration addresses; the Linux device drivers only read and write PCI I/O and PCI memory addresses.

6.4 PCI-ISA Bridges

These bridges support legacy ISA devices by translating PCI I/O and PCI Memory space accesses into ISA I/O and ISA Memory accesses. A lot of systems now sold contain several ISA bus slots and several PCI bus slots. Over time the need for this backwards compatibility will dwindle and PCI only systems will be sold. Where in the ISA address spaces (I/O and Memory) the ISA devices of the system have their registers was fixed in the dim mists of time by the early Intel 8080 based PCs. Even a $5000 Alpha AXP based computer systems will have its ISA floppy controller at the same place in ISA I/O space as the first IBM PC. The PCI specification copes with

31 … 11	10 … 8	7 … 2	1	0
Device Select	Func	Register	0	0

Figure 6.3: Type 0 PCI Configuration Cycle

31 … 24	23 … 16	15 … 11	10 … 8	7 … 2	1	0
Reserved	Bus	Device	Func	Register	0	1

Figure 6.4: Type 1 PCI Configuration Cycle

this by reserving the lower regions of the PCI I/O and PCI Memory address spaces for use by the ISA peripherals in the system and using a single PCI-ISA bridge to translate any PCI memory accesses to those regions into ISA accesses.

6.5 PCI-PCI Bridges

PCI-PCI bridges are special PCI devices that glue the PCI buses of the system together. Simple systems have a single PCI bus but there is an electrical limit on the number of PCI devices that a single PCI bus can support. Using PCI-PCI bridges to add more PCI buses allows the system to support many more PCI devices. This is particularly important for a high performance server. Of course, Linux fully supports the use of PCI-PCI bridges.

6.5.1 PCI-PCI Bridges: PCI I/O and PCI Memory Windows

PCI-PCI bridges only pass a subset of PCI I/O and PCI memory read and write requests downstream. For example, in Figure 6.1 on page 62, the PCI-PCI bridge will only pass read and write addresses from PCI bus 0 to PCI bus 1 if they are for PCI I/O or PCI memory addresses owned by either the SCSI or ethernet device; all other PCI I/O and memory addresses are ignored. This filtering stops addresses propogating needlessly throughout the system. To do this, the PCI-PCI bridges must be programmed with a base and limit for PCI I/O and PCI Memory space access that they have to pass from their primary bus onto their secondary bus. Once the PCI-PCI Bridges in a system have been configured then so long as the Linux device drivers only access PCI I/O and PCI Memory space via these windows, the PCI-PCI Bridges are invisible. This is an important feature that makes life easier for Linux PCI device driver writers. However, it also makes PCI-PCI bridges somewhat tricky for Linux to configure as we shall see later on.

6.5.2 PCI-PCI Bridges: PCI Configuration Cycles and PCI Bus Numbering

So that the CPU's PCI initialization code can address devices that are not on the main PCI bus, there has to be a mechanism that allows bridges to decide whether or not to pass Configuration cycles from their primary interface to their secondary interface. A cycle is just an address as it appears on the PCI bus. The PCI specification defines two formats for the PCI Configuration addresses; Type 0 and Type 1;

these are shown in Figure 6.3 and Figure 6.4 respectively. Type 0 PCI Configuration cycles do not contain a bus number and these are interpretted by all devices as being for PCI configuration addresses on this PCI bus. Bits 31:11 of the Type 0 configuraration cycles are treated as the device select field. One way to design a system is to have each bit select a different device. In this case bit 11 would select the PCI device in slot 0, bit 12 would select the PCI device in slot 1 and so on. Another way is to write the device's slot number directly into bits 31:11. Which mechanism is used in a system depends on the system's PCI memory controller.

Type 1 PCI Configuration cycles contain a PCI bus number and this type of configuration cycle is ignored by all PCI devices except the PCI-PCI bridges. All of the PCI-PCI Bridges seeing Type 1 configuration cycles may choose to pass them to the PCI buses downstream of themselves. Whether the PCI-PCI Bridge ignores the Type 1 configuration cycle or passes it onto the downstream PCI bus depends on how the PCI-PCI Bridge has been configured. Every PCI-PCI bridge has a primary bus interface number and a secondary bus interface number. The primary bus interface being the one nearest the CPU and the secondary bus interface being the one furthest away. Each PCI-PCI Bridge also has a subordinate bus number and this is the maximum bus number of all the PCI buses that are bridged beyond the secondary bus interface. Or to put it another way, the subordinate bus number is the highest numbered PCI bus downstream of the PCI-PCI bridge. When the PCI-PCI bridge sees a Type 1 PCI configuration cycle it does one of the following things:

- Ignore it if the bus number specified is not in between the bridge's secondary bus number and subordinate bus number (inclusive),
- Convert it to a Type 0 configuration command if the bus number specified matches the secondary bus number of the bridge,
- Pass it onto the secondary bus interface unchanged if the bus number specified is greater than the secondary bus number and less than or equal to the subordinate bus number.

So, if we want to address Device 1 on bus 3 of the topology Figure 6.9 on page 71 we must generate a Type 1 Configuration command from the CPU. $Bridge_1$ passes this unchanged onto Bus 1. $Bridge_2$ ignores it but $Bridge_3$ converts it into a Type 0 Configuration command and sends it out on Bus 3 where Device 1 responds to it.

It is up to each individual operating system to allocate bus numbers during PCI configuration but whatever the numbering scheme used the following statement must be true for all of the PCI-PCI bridges in the system:

"All PCI buses located behind a PCI-PCI bridge must reside between the seondary bus number and the subordinate bus number (inclusive)."

If this rule is broken then the PCI-PCI Bridges will not pass and translate Type 1 PCI configuration cycles correctly and the system will fail to find and initialise the PCI devices in the system. To achieve this numbering scheme, Linux configures these special devices in a particular order. Section 6.6.2 on page 68 describes Linux's PCI bridge and bus numbering scheme in detail together with a worked example.

6.6 Linux PCI Initialization

The PCI initialisation code in Linux is broken into three logical parts:

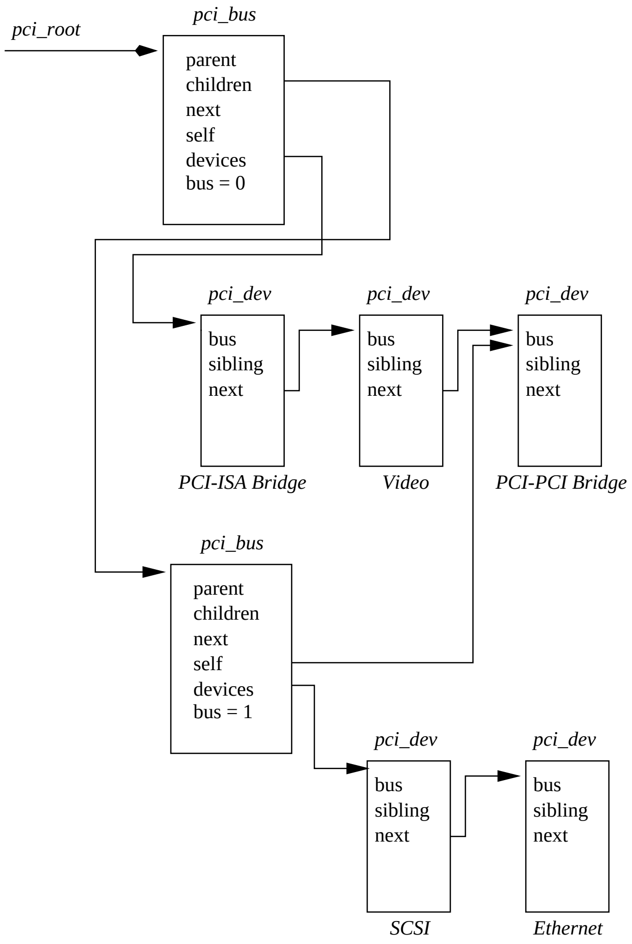

Figure 6.5: Linux Kernel PCI Data Structures

PCI Device Driver This pseudo-device driver searches the PCI system starting at Bus 0 and locates all PCI devices and bridges in the system. It builds a linked list of data structures describing the topology of the system. Additionally, it numbers all of the bridges that it finds.

See `drivers/-pci/pci.c` and `include/linux/-pci.h`

PCI BIOS This software layer provides the services described in [4, PCI BIOS ROM specification]. Even though Alpha AXP does not have BIOS services, there is equivalent code in the Linux kernel providing the same functions,

See `arch/*/-kernel/bios32.c`

PCI Fixup System specific fixup code tidies up the system specific loose ends of PCI initialization.

See `arch/*/-kernel/bios32.c`

6.6.1 The Linux Kernel PCI Data Structures

As the Linux kernel initialises the PCI system it builds data structures mirroring the real PCI topology of the system. Figure 6.5 shows the relationships of the data structures that it would build for the example PCI system in Figure 6.1 on page 62.

Each PCI device (including the PCI-PCI Bridges) is described by a `pci_dev` data structure. Each PCI bus is described by a `pci_bus` data structure. The result is a tree structure of PCI buses each of which has a number of child PCI devices attached to it. As a PCI bus can only be reached using a PCI-PCI Bridge (except the primary PCI bus, bus 0), each `pci_bus` contains a pointer to the PCI device (the PCI-PCI Bridge) that it is accessed through. That PCI device is a child of the the PCI Bus's parent PCI bus.

Not shown in the Figure 6.5 is a pointer to all of the PCI devices in the system, `pci_devices`. All of the PCI devices in the system have their `pci_dev` data structures queued onto this queue.. This queue is used by the Linux kernel to quickly find all of the PCI devices in the system.

6.6.2 The PCI Device Driver

See `Scan_bus()` in `drivers/pci/-pci.c`

The PCI device driver is not really a device driver at all but a function of the operating system called at system initialisation time. The PCI initialisation code must scan all of the PCI buses in the system looking for all PCI devices in the system (including PCI-PCI bridge devices). It uses the PCI BIOS code to find out if every possible slot in the current PCI bus that it is scanning is occupied. If the PCI slot is occupied, it builds a `pci_dev` data structure describing the device and links into the list of known PCI devices (pointed at by `pci_devices`).

The PCI initialisation code starts by scanning PCI Bus 0. It tries to read the *Vendor Identification* and *Device Identification* fields for every possible PCI device in every possible PCI slot. When it finds an occupied slot it builds a `pci_dev` data structure describing the device. All of the `pci_dev` data structures built by the PCI initialisation code (including all of the PCI-PCI Bridges) are linked into a singly linked list; `pci_devices`.

If the PCI device that was found was a PCI-PCI bridge then a `pci_bus` data structure is built and linked into the tree of `pci_bus` and `pci_dev` data structures pointed at by `pci_root`. The PCI initialisation code can tell if the PCI device is a PCI-PCI Bridge because it has a class code of *0x060400*. The Linux kernel then configures the PCI bus on the other (downstream) side of the PCI-PCI Bridge that it has just found. If more PCI-PCI Bridges are found then these are also configured. This process is known as a depthwise algorithm; the system's PCI topology is fully mapped depthwise before searching breadthwise. Looking at Figure 6.1 on page 62, Linux would configure PCI Bus 1 with its Ethernet and SCSI device before it configured the video device on PCI Bus 0.

As Linux searches for downstream PCI buses it must also configure the intervening PCI-PCI bridges' secondary and subordinate bus numbers. This is described in detail in Section 6.6.2 below.

Configuring PCI-PCI Bridges - Assigning PCI Bus Numbers

For PCI-PCI bridges to pass PCI I/O, PCI Memory or PCI Configuration address space reads and writes across them, they need to know the following:

Primary Bus Number The bus number immediately upstream of the PCI-PCI Bridge,

Secondary Bus Number The bus number immediately downstream of the PCI-PCI Bridge,

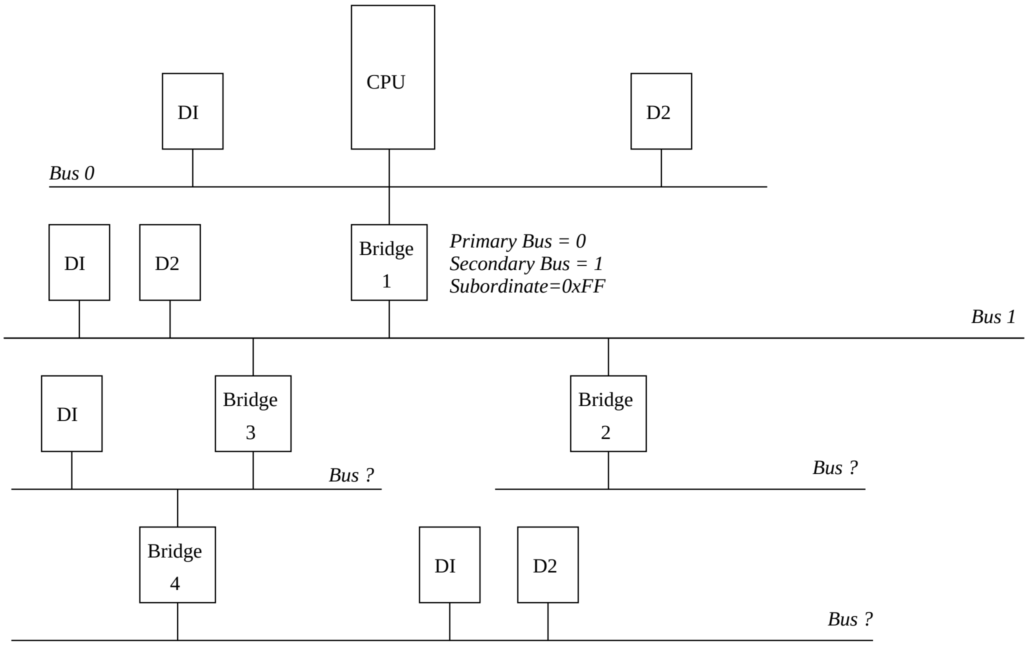

Figure 6.6: Configuring a PCI System: Part 1

Subordinate Bus Number The highest bus number of all of the buses that can be reached downstream of the bridge.

PCI I/O and PCI Memory Windows The window base and size for PCI I/O address space and PCI Memory address space for all addresses downstream of the PCI-PCI Bridge.

The problem is that at the time when you wish to configure any given PCI-PCI bridge you do not know the subordinate bus number for that bridge. You do not know if there are further PCI-PCI bridges downstream and if you did, you do not know what numbers will be assigned to them. The answer is to use a depthwise recursive algorithm and scan each bus for any PCI-PCI bridges assigning them numbers as they are found. As each PCI-PCI bridge is found and its secondary bus numbered, assign it a temporary subordinate number of *0xFF* and scan and assign numbers to all PCI-PCI bridges downstream of it. This all seems complicated but the worked example below makes this process clearer.

PCI-PCI Bridge Numbering: Step 1 Taking the topology in Figure 6.6, the first bridge the scan would find is $Bridge_1$. The PCI bus downstream of $Bridge_1$ would be numbered as 1 and $Bridge_1$ assigned a secondary bus number of 1 and a temporary subordinate bus number of *0xFF*. This means that all Type 1 PCI Configuration addresses specifying a PCI bus number of 1 or higher would be passed across $Bridge_1$ and onto PCI Bus 1. They would be translated into Type 0 Configuration cycles if they have a bus number of 1 but left untranslated for all other bus numbers. This is exactly what the Linux PCI initialisation code needs to do in order to go and scan PCI Bus 1.

PCI-PCI Bridge Numbering: Step 2 Linux uses a depthwise algorithm and so the initialisation code goes on to scan PCI Bus 1. Here it finds PCI-PCI

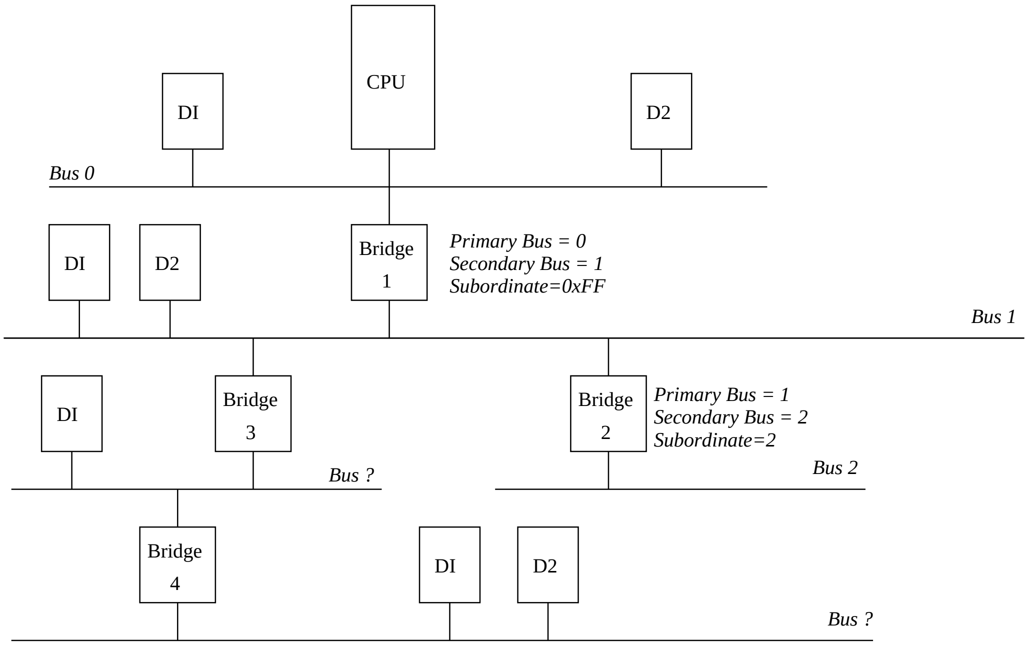

Figure 6.7: Configuring a PCI System: Part 2

$Bridge_2$. There are no further PCI-PCI bridges beyond PCI-PCI $Bridge_2$, so it is assigned a subordinate bus number of 2 which matches the number assigned to its secondary interface. Figure 6.7 shows how the buses and PCI-PCI bridges are numbered at this point.

PCI-PCI Bridge Numbering: Step 3 The PCI initialisation code returns to scanning PCI Bus 1 and finds another PCI-PCI bridge, $Bridge_3$. It is assigned 1 as its primary bus interface number, 3 as its secondary bus interface number and *0xFF* as its subordinate bus number. Figure 6.8 on page 71 shows how the system is configured now. Type 1 PCI configuration cycles with a bus number of 1, 2 or 3 wil be correctly delivered to the appropriate PCI buses.

PCI-PCI Bridge Numbering: Step 4 Linux starts scanning PCI Bus 3, downstream of PCI-PCI $Bridge_3$. PCI Bus 3 has another PCI-PCI bridge ($Bridge_4$) on it, it is assigned 3 as its primary bus number and 4 as its secondary bus number. It is the last bridge on this branch and so it is assigned a subordinate bus interface number of 4. The initialisation code returns to PCI-PCI $Bridge_3$ and assigns it a subordinate bus number of 4. Finally, the PCI initialisation code can assign 4 as the subordinate bus number for PCI-PCI $Bridge_1$. Figure 6.9 on page 71 shows the final bus numbers.

6.6.3 PCI BIOS Functions

See `arch/*/kernel/bios32.c`

The PCI BIOS functions are a series of standard routines which are common across all platforms. For example, they are the same for both Intel and Alpha AXP based systems. They allow the CPU controlled access to all of the PCI address spaces. Only Linux kernel code and device drivers may use them.

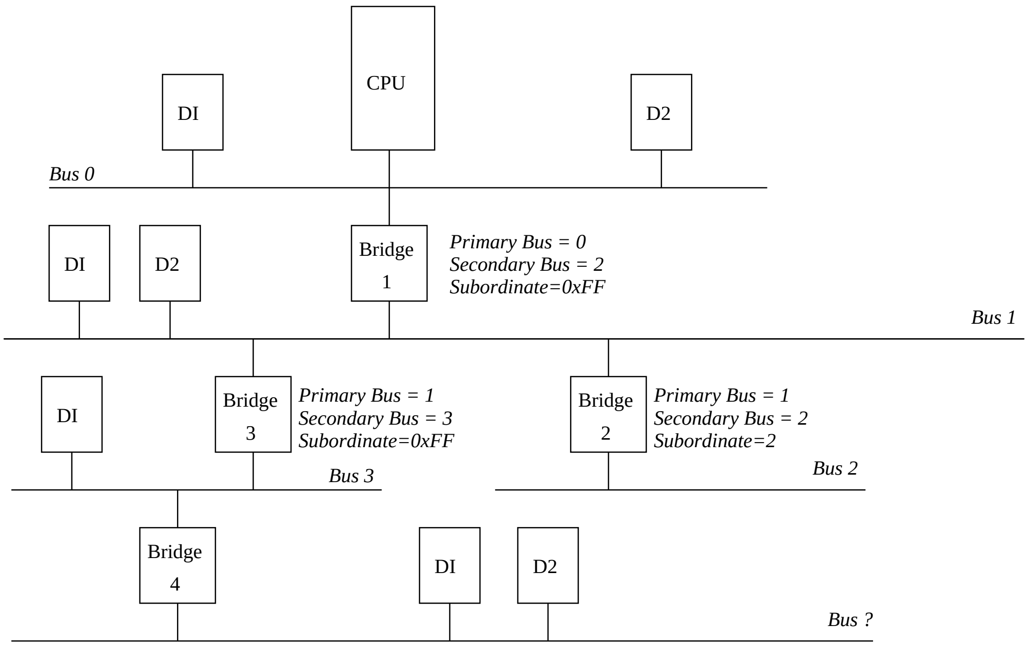

Figure 6.8: Configuring a PCI System: Part 3

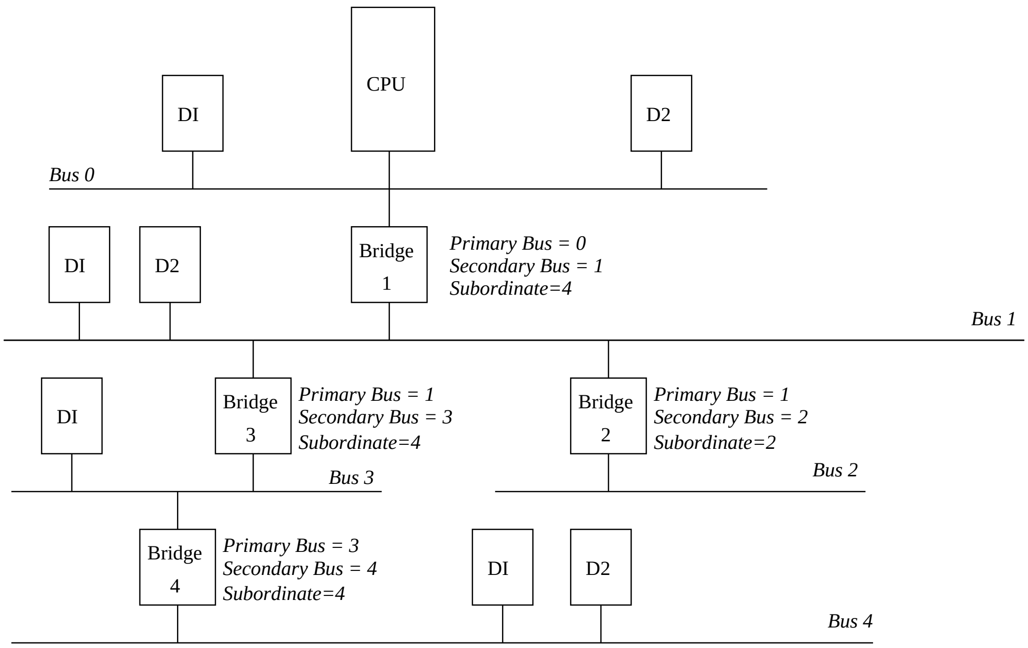

Figure 6.9: Configuring a PCI System: Part 4

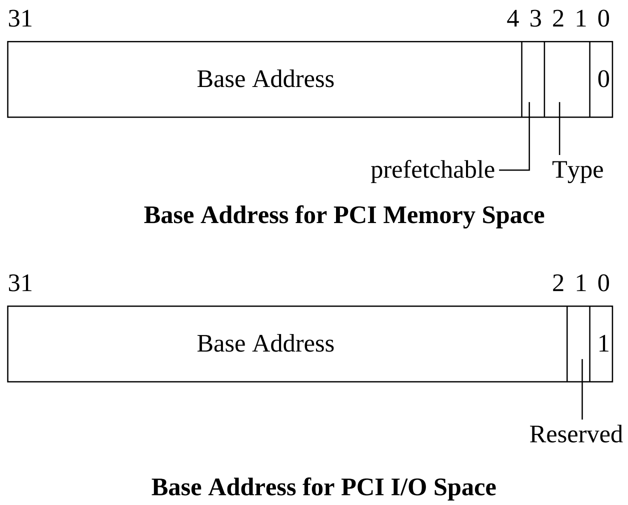

Figure 6.10: PCI Configuration Header: Base Address Registers

6.6.4 PCI Fixup

See `arch/*/kernel/bios32.c`

The PCI fixup code for Alpha AXP does rather more than that for Intel (which basically does nothing). For Intel based systems the system BIOS, which ran at boot time, has already fully configured the PCI system. This leaves Linux with little to do other than map that configuration. For non-Intel based systems further configuration needs to happen to:

- Allocate PCI I/O and PCI Memory space to each device,
- Configure the PCI I/O and PCI Memory address windows for each PCI-PCI bridge in the system,
- Generate *Interrupt Line* values for the devices; these control interrupt handling for the device.

The next subsections describe how that code works.

Finding Out How Much PCI I/O and PCI Memory Space a Device Needs

Each PCI device found is queried to find out how much PCI I/O and PCI Memory address space it requires. To do this, each Base Address Register has all 1's written to it and then read. The device will return 0's in the don't-care address bits, effectively specifying the address space required.

There are two basic types of Base Address Register, the first indicates within which address space the devices registers must reside; either PCI I/O or PCI Memory space. This is indicated by Bit 0 of the register. Figure 6.10 shows the two forms of the Base Address Register for PCI Memory and for PCI I/O.

To find out just how much of each address space a given Base Address Register is requesting, you write all 1s into the register and then read it back. The device will specify zeros in the don't care address bits, effectively specifying the address space required. This design implies that all address spaces used are a power of two and are naturally aligned.

For example when you initialize the DECChip 21142 PCI Fast Ethernet device, it tells you that it needs *0x100* bytes of space of either PCI I/O or PCI Memory. The initialization code allocates it space. The moment that it allocates space, the 21142's control and status registers can be seen at those addresses.

Allocating PCI I/O and PCI Memory to PCI-PCI Bridges and Devices

Like all memory the PCI I/O and PCI memory spaces are finite, and to some extent scarce. The PCI Fixup code for non-Intel systems (and the BIOS code for Intel systems) has to allocate each device the amount of memory that it is requesting in an efficient manner. Both PCI I/O and PCI Memory must be allocated to a device in a naturally aligned way. For example, if a device asks for *0xB0* of PCI I/O space then it must be aligned on an address that is a multiple of *0xB0*. In addition to this, the PCI I/O and PCI Memory bases for any given bridge must be aligned on 4K and on 1Mbyte boundaries respectively. Given that the address spaces for downstream devices must lie within all of the upstream PCI-PCI Bridge's memory ranges for any given device, it is a somewhat difficult problem to allocate space efficiently.

The algorithm that Linux uses relies on each device described by the bus/device tree built by the PCI Device Driver being allocated address space in ascending PCI I/O memory order. Again a recursive algorithm is used to walk the `pci_bus` and `pci_dev` data structures built by the PCI initialisation code. Starting at the root PCI bus (pointed at by `pci_root`) the BIOS fixup code:

- Aligns the current global PCI I/O and Memory bases on 4K and 1 Mbyte boundaries respectively,
- For every device on the current bus (in ascending PCI I/O memory needs),
  - allocates it space in PCI I/O and/or PCI Memory,
  - moves on the global PCI I/O and Memory bases by the appropriate amounts,
  - enables the device's use of PCI I/O and PCI Memory,
- Allocates space recursively to all of the buses downstream of the current bus. Note that this will change the global PCI I/O and Memory bases,
- Aligns the current global PCI I/O and Memory bases on 4K and 1 Mbyte boundaries respectively and in doing so figure out the size and base of PCI I/O and PCI Memory windows required by the current PCI-PCI bridge,
- Programs the PCI-PCI bridge that links to this bus with its PCI I/O and PCI Memory bases and limits,
- Turns on bridging of PCI I/O and PCI Memory accesses in the PCI-PCI Bridge. This means that if any PCI I/O or PCI Memory addresses seen on the Bridge's primary PCI bus that are within its PCI I/O and PCI Memory address windows will be bridged onto its secondary PCI bus.

Taking the PCI system in Figure 6.1 on page 62 as our example the PCI Fixup code would set up the system in the following way:

Align the PCI bases PCI I/O is *0x4000* and PCI Memory is *0x100000*. This allows the PCI-ISA bridges to translate all addresses below these into ISA address cycles,

The Video Device This is asking for *0x200000* of PCI Memory and so we allocate it that amount starting at the current PCI Memory base of *0x200000* as it has to be naturally aligned to the size requested. The PCI Memory base is moved to *0x400000* and the PCI I/O base remains at *0x4000*.

The PCI-PCI Bridge We now cross the PCI-PCI Bridge and allocate PCI memory there, note that we do not need to align the bases as they are already correctly aligned:

- **The Ethernet Device** This is asking for *0xB0* bytes of both PCI I/O and PCI Memory space. It gets allocated PCI I/O at *0x4000* and PCI Memory at *0x400000*. The PCI Memory base is moved to *0x4000B0* and the PCI I/O base to *0x40B0*.

- **The SCSI Device** This is asking for *0x1000* PCI Memory and so it is allocated it at *0x401000* after it has been naturally aligned. The PCI I/O base is still *0x40B0* and the PCI Memory base has been moved to *0x402000*.

The PCI-PCI Bridge's PCI I/O and Memory Windows We now return to the bridge and set its PCI I/O window at between *0x4000* and *0x40B0* and it's PCI Memory window at between *0x400000* and *0x402000*. This means that the PCI-PCI Bridge will ignore the PCI Memory accesses for the video device and pass them on if they are for the ethernet or SCSI devices.

Chapter 7

Interrupts and Interrupt Handling

This chapter looks at how interrupts are handled by the Linux kernel. Whilst the kernel has generic mechanisms and interfaces for handling interrupts, most of the interrupt handling details are architecture specific.

Linux uses a lot of different pieces of hardware to perform many different tasks. The video device drives the monitor, the IDE device drives the disks and so on. You could drive these devices synchronously, that is you could send a request for some operation (say writing a block of memory out to disk) and then wait for the operation to complete. That method, although it would work, is very inefficient and the operating system would spend a lot of time "busy doing nothing" as it waited for each operation to complete. A better, more efficient, way is to make the request and then do other, more useful work and later be interrupted by the device when it has finished the request. With this scheme, there may be many outstanding requests to the devices in the system all happening at the same time.

There has to be some hardware support for the devices to interrupt whatever the CPU is doing. Most, if not all, general purpose processors such as the Alpha AXP use a similar method. Some of the physical pins of the CPU are wired such that changing the voltage (for example changing it from +5v to -5v) causes the CPU to stop what it is doing and to start executing special code to handle the interruption; the interrupt handling code. One of these pins might be connected to an interval timer and receive an interrupt every 1000th of a second, others may be connected to the other devices in the system, such as the SCSI controller.

Systems often use an interrupt controller to group the device interrupts together before passing on the signal to a single interrupt pin on the CPU. This saves interrupt pins on the CPU and also gives flexibility when designing systems. The interrupt controller has mask and status registers that control the interrupts. Setting the bits in the mask register enables and disables interrupts and the status register returns

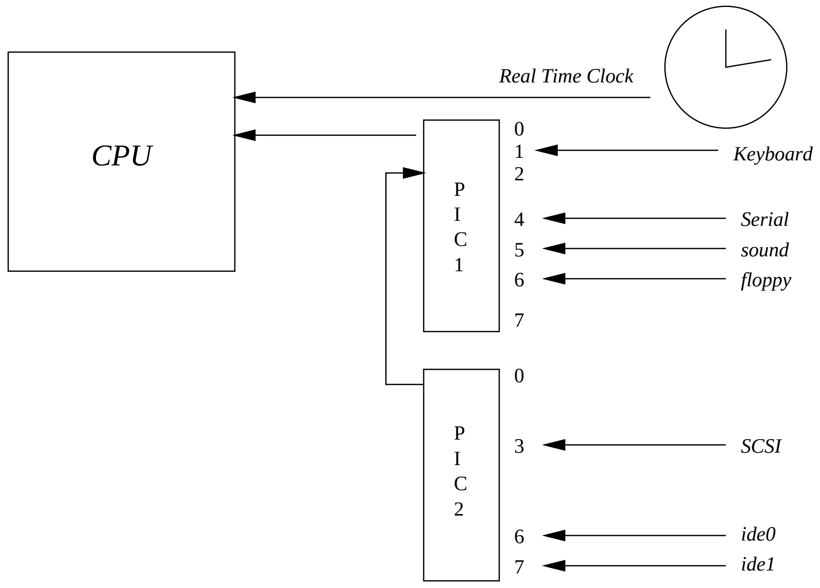

Figure 7.1: A Logical Diagram of Interrupt Routing

the currently active interrupts in the system.

Some of the interrupts in the system may be hard-wired, for example, the real time clock's interval timer may be permanently connected to pin 3 on the interrupt controller. However, what some of the pins are connected to may be determined by what controller card is plugged into a particular ISA or PCI slot. For example, pin 4 on the interrupt controller may be connected to PCI slot number 0 which might one day have an ethernet card in it but the next have a SCSI controller in it. The bottom line is that each system has its own interrupt routing mechanisms and the operating system must be flexible enough to cope.

Most modern general purpose microprocessors handle the interrupts the same way. When a hardware interrupt occurs the CPU stops executing the instructions that it was executing and jumps to a location in memory that either contains the interrupt handling code or an instruction branching to the interrupt handling code. This code usually operates in a special mode for the CPU, *interrupt mode*, and, normally, no other interrupts can happen in this mode. There are exceptions though; some CPUs rank the interrupts in priority and higher level interrupts may happen. This means that the first level interrupt handling code must be very carefully written and it often has its own stack, which it uses to store the CPU's execution state (all of the CPU's normal registers and context) before it goes off and handles the interrupt. Some CPUs have a special set of registers that only exist in interrupt mode, and the interrupt code can use these registers to do most of the context saving it needs to do.

When the interrupt has been handled, the CPU's state is restored and the interrupt is dismissed. The CPU will then continue to doing whatever it was doing before being interrupted. It is important that the interrupt processing code is as efficient as possible and that the operating system does not block interrupts too often or for too long.

7.1 Programmable Interrupt Controllers

Systems designers are free to use whatever interrupt architecture they wish but IBM PCs use the Intel 82C59A-2 CMOS Programmable Interrupt Controller [6, Intel Peripheral Components] or its derivatives. This controller has been around since the dawn of the PC and it is programmable with its registers being at well known locations in the ISA address space. Even very modern support logic chip sets keep equivalent registers in the same place in ISA memory. Non-Intel based systems such as Alpha AXP based PCs are free from these architectural constraints and so often use different interrupt controllers.

Figure 7.1 shows that there are two 8 bit controllers chained together; each having a mask and an interrupt status register, PIC1 and PIC2. The mask registers are at addresses *0x21* and *0xA1* and the status registers are at *0x20* and *0xA0* Writing a one to a particular bit of the mask register enables an interrupt, writing a zero disables it. So, writing one to bit 3 would enable interrupt 3, writing zero would disable it. Unfortunately (and irritatingly), the interrupt mask registers are write only, you cannot read back the value that you wrote. This means that Linux must keep a local copy of what it has set the mask registers to. It modifies these saved masks in the interrupt enable and disable routines and writes the full masks to the registers every time.

When an interrupt is signalled, the interrupt handling code reads the two interrupt status registers (ISRs). It treats the ISR at *0x20* as the bottom eight bits of a sixteen bit interrupt register and the ISR at *0xA0* as the top eight bits. So, an interrupt on bit 1 of the ISR at *0xA0* would be treated as system interrupt 9. Bit 2 of PIC1 is not available as this is used to chain interrupts from PIC2, any interrupt on PIC2 results in bit 2 of PIC1 being set.

7.2 Initializing the Interrupt Handling Data Structures

The kernel's interrupt handling data structures are set up by the device drivers as they request control of the system's interrupts. To do this the device driver uses a set of Linux kernel services that are used to request an interrupt, enable it and to disable it. The individual device drivers call these routines to register their interrupt handling routine addresses.

See request_irq(), enable_irq() and disable_irq() in arch/*/kernel irq.c

Some interrupts are fixed by convention for the PC architecture and so the driver simply requests its interrupt when it is initialized. This is what the floppy disk device driver does; it always requests IRQ 6. There may be occassions when a device driver does not know which interrupt the device will use. This is not a problem for PCI device drivers as they always know what their interrupt number is. Unfortunately there is no easy way for ISA device drivers to find their interrupt number. Linux solves this problem by allowing device drivers to probe for their interrupts.

First, the device driver does something to the device that causes it to interrupt. Then all of the unassigned interrupts in the system are enabled. This means that the device's pending interrupt will now be delivered via the programmable interrupt controller. Linux reads the interrupt status register and returns its contents to the device driver. A non-zero result means that one or more interrupts occured during

See irq_probe_*() in arch/*/kernel/-irq.c

the probe. The driver now turns probing off and the unassigned interrupts are all disabled. If the ISA device driver has successfully found its IRQ number then it can now request control of it as normal.

PCI based systems are much more dynamic than ISA based systems. The interrupt pin that an ISA device uses is often set using jumpers on the hardware device and fixed in the device driver. On the other hand, PCI devices have their interrupts allocated by the PCI BIOS or the PCI subsystem as PCI is initialized when the system boots. Each PCI device may use one of four interrupt pins, A, B, C or D. This was fixed when the device was built and most devices default to interrupt on pin A. The PCI interrupt lines A, B, C and D for each PCI slot are routed to the interrupt controller. So, Pin A from PCI slot 4 might be routed to pin 6 of the interrupt controller, pin B of PCI slot 4 to pin 7 of the interrupt controller and so on.

See arch/alpha/-kernel/bios32.c

How the PCI interrupts are routed is entirely system specific and there must be some set up code which understands this PCI interrupt routing topology. On Intel based PCs this is the system BIOS code that runs at boot time but for system's without BIOS (for example Alpha AXP based systems) the Linux kernel does this setup. The PCI set up code writes the pin number of the interrupt controller into the PCI configuration header for each device. It determines the interrupt pin (or IRQ) number using its knowledge of the PCI interrupt routing topology together with the devices PCI slot number and which PCI interrupt pin that it is using. The interrupt pin that a device uses is fixed and is kept in a field in the PCI configuration header for this device. It writes this information into the *interrupt line* field that is reserved for this purpose. When the device driver runs, it reads this information and uses it to request control of the interrupt from the Linux kernel.

There may be many PCI interrupt sources in the system, for example when PCI-PCI bridges are used. The number of interrupt sources may exceed the number of pins on the system's programmable interrupt controllers. In this case, PCI devices may share interrupts, one pin on the interrupt controller taking interrupts from more than one PCI device. Linux supports this by allowing the first requestor of an interrupt source declare whether it may be shared. Sharing interrupts results in several `irqaction` data structures being pointed at by one entry in the `irq_action vector` vector. When a shared interrupt happens, Linux will call all of the interrupt handlers for that source. Any device driver that can share interrupts (which should be all PCI device drivers) must be prepared to have its interrupt handler called when there is no interrupt to be serviced.

7.3 Interrupt Handling

One of the principal tasks of Linux's interrupt handling subsystem is to route the interrupts to the right pieces of interrupt handling code. This code must understand the interrupt topology of the system. If, for example, the floppy controller interrupts on pin 6 [1] of the interrupt controller then it must recognize the interrupt as from the floppy and route it to the floppy device driver's interrupt handling code. Linux uses a set of pointers to data structures containing the addresses of the routines that handle the system's interrupts. These routines belong to the device drivers for the devices in

[1] Actually, the floppy controller is one of the fixed interrupts in a PC system as, by convention, the floppy controller is always wired to interrupt 6.

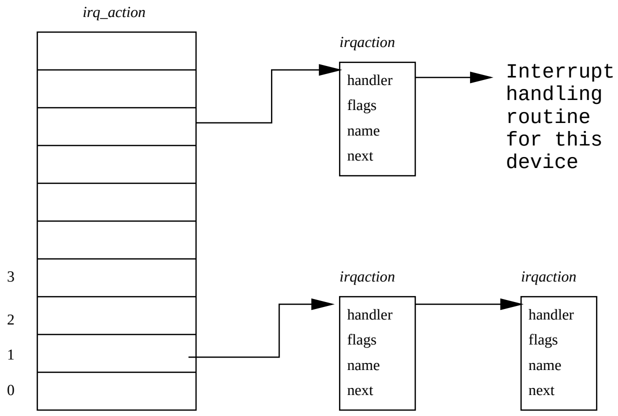

Figure 7.2: Linux Interrupt Handling Data Structures

the system and it is the responsibility of each device driver to request the interrupt that it wants when the driver is initialized. Figure 7.2 shows that `irq_action` is a vector of pointers to the `irqaction` data structure. Each `irqaction` data structure contains information about the handler for this interrupt, including the address of the interrupt handling routine. As the number of interrupts and how they are handled varies between architectures and, sometimes, between systems, the Linux interrupt handling code is architecture specific. This means that the size of the `irq_action vector` vector varies depending on the number of interrupt sources that there are.

When the interrupt happens, Linux must first determine its source by reading the interrupt status register of the system's programmable interrupt controllers. It then translates that source into an offset into the `irq_action vector` vector. So, for example, an interrupt on pin 6 of the interrupt controller from the floppy controller would be translated into the seventh pointer in the vector of interrupt handlers. If there is not an interrupt handler for the interrupt that occurred then the Linux kernel will log an error, otherwise it will call into the interrupt handling routines for all of the `irqaction` data structures for this interrupt source.

When the device driver's interrupt handling routine is called by the Linux kernel it must efficiently work out why it was interrupted and respond. To find the cause of the interrupt the device driver would read the status register of the device that interrupted. The device may be reporting an error or that a requested operation has completed. For example the floppy controller may be reporting that it has completed the positioning of the floppy's read head over the correct sector on the floppy disk. Once the reason for the interrupt has been determined, the device driver may need to do more work. If it does, the Linux kernel has mechanisms that allow it to postpone that work until later. This avoids the CPU spending too much time in interrupt mode. See the Device Driver chapter (Chapter 8) for more details.

REVIEW NOTE: *Fast and slow interrupts, are these an Intel thing?*

Chapter 8

Device Drivers

One of the purposes of an operating system is to hide the peculiarities of the system's hardware devices from its users. For example the Virtual File System presents a uniform view of the mounted filesystems irrespective of the underlying physical devices. This chapter describes how the Linux kernel manages the physical devices in the system.

The CPU is not the only intelligent device in the system, every physical device has its own hardware controller. The keyboard, mouse and serial ports are controlled by a SuperIO chip, the IDE disks by an IDE controller, SCSI disks by a SCSI controller and so on. Each hardware controller has its own control and status registers (CSRs) and these differ between devices. The CSRs for an Adaptec 2940 SCSI controller are completely different from those of an NCR 810 SCSI controller. The CSRs are used to start and stop the device, to initialize it and to diagnose any problems with it. Instead of putting code to manage the hardware controllers in the system into every application, the code is kept in the Linux kernel. The software that handles or manages a hardware controller is known as a device driver. The Linux kernel device drivers are, essentially, a shared library of privileged, memory resident, low level hardware handling routines. It is Linux's device drivers that handle the peculiarities of the devices they are managing.

One of the basic features of UN⋆X is that it abstracts the handling of devices. All hardware devices look like regular files; they can be opened, closed, read and written using the same, standard, system calls that are used to manipulate files. Every device in the system is represented by a *device special file*, for example the first IDE disk in the system is represented by `/dev/hda`. For block (disk) and character devices, these device special files are created by the `mknod` command and they describe the device using major and minor device numbers. Network devices are also represented by device special files but they are created by Linux as it finds and initializes the network controllers in the system. All devices controlled by the same device driver have a common major device number. The minor device numbers are used to distinguish between different devices and their controllers, for example each partition on the primary IDE disk has a different minor device number. So, `/dev/hda2`, the second partition of the primary IDE disk has a major number of 3 and a minor number of 2. Linux maps the device special file passed in system calls (say to mount a file system on a block device) to the device's device driver using the major device number and a number of system tables, for example the character device table, `chrdevs` .

See `fs/devices.c`

Linux supports three types of hardware device: character, block and network. Character devices are read and written directly without buffering, for example the system's serial ports `/dev/cua0` and `/dev/cua1`. Block devices can only be written to and read from in multiples of the block size, typically 512 or 1024 bytes. Block devices are accessed via the buffer cache and may be randomly accessed, that is to say, any block can be read or written no matter where it is on the device. Block devices can be accessed via their device special file but more commonly they are accessed via the file system. Only a block device can support a mounted file system. Network devices are accessed via the BSD socket interface and the networking subsytems described in the Networking chapter (Chapter 10).

There are many different device drivers in the Linux kernel (that is one of Linux's strengths) but they all share some common attributes:

kernel code Device drivers are part of the kernel and, like other code within the kernel, if they go wrong they can seriously damage the system. A badly written driver may even crash the system, possibly corrupting file systems and losing data,

Kernel interfaces Device drivers must provide a standard interface to the Linux kernel or to the subsystem that they are part of. For example, the terminal driver provides a file I/O interface to the Linux kernel and a SCSI device driver provides a SCSI device interface to the SCSI subsystem which, in turn, provides both file I/O and buffer cache interfaces to the kernel.

Kernel mechanisms and services Device drivers make use of standard kernel services such as memory allocation, interrupt delivery and wait queues to operate,

Loadable Most of the Linux device drivers can be loaded on demand as kernel modules when they are needed and unloaded when they are no longer being used. This makes the kernel very adaptable and efficient with the system's resources,

Configurable Linux device drivers can be built into the kernel. Which devices are built is configurable when the kernel is compiled,

Dynamic As the system boots and each device driver is initialized it looks for the hardware devices that it is controlling. It does not matter if the device being controlled by a particular device driver does not exist. In this case the device driver is simply redundant and causes no harm apart from occupying a little of the system's memory.

8.1 Polling and Interrupts

Each time the device is given a command, for example *"move the read head to sector 42 of the floppy disk"* the device driver has a choice as to how it finds out that the command has completed. The device drivers can either poll the device or they can use interrupts.

Polling the device usually means reading its status register every so often until the device's status changes to indicate that it has completed the request. As a device driver is part of the kernel it would be disasterous if a driver were to poll as nothing else in the kernel would run until the device had completed the request. Instead polling device drivers use system timers to have the kernel call a routine within the device driver at some later time. This timer routine would check the status of the command and this is exactly how Linux's floppy driver works. Polling by means of timers is at best approximate, a much more efficient method is to use interrupts.

An interrupt driven device driver is one where the hardware device being controlled will raise a hardware interrupt whenever it needs to be serviced. For example, an ethernet device driver would interrupt whenever it receives an ethernet packet from the network. The Linux kernel needs to be able to deliver the interrupt from the hardware device to the correct device driver. This is achieved by the device driver registering its usage of the interrupt with the kernel. It registers the address of an interrupt handling routine and the interrupt number that it wishes to own. You can see which interrupts are being used by the device drivers, as well as how many of each type of interrupts there have been, by looking at `/proc/interrupts`:

```
 0:     727432   timer
 1:      20534   keyboard
 2:          0   cascade
 3:      79691 + serial
 4:      28258 + serial
 5:          1   sound blaster
11:      20868 + aic7xxx
13:          1   math error
14:        247 + ide0
15:        170 + ide1
```

This requesting of interrupt resources is done at driver initialization time. Some of the interrupts in the system are fixed, this is a legacy of the IBM PC's architecture. So, for example, the floppy disk controller always uses interrupt 6. Other interrupts, for example the interrupts from PCI devices are dynamically allocated at boot time. In this case the device driver must first discover the interrupt number (IRQ) of the device that it is controlling before it requests ownership of that interrupt. For PCI interrupts Linux supports standard PCI BIOS callbacks to determine information about the devices in the system, including their IRQ numbers.

How an interrupt is delivered to the CPU itself is architecture dependent but on most architectures the interrupt is delivered in a special mode that stops other interrupts from happening in the system. A device driver should do as little as possible in its interrupt handling routine so that the Linux kernel can dismiss the interrupt and return to what it was doing before it was interrupted. Device drivers that need to do a lot of work as a result of receiving an interrupt can use the kernel's bottom half handlers or task queues to queue routines to be called later on.

8.2 Direct Memory Access (DMA)

Using interrupts driven device drivers to transfer data to or from hardware devices works well when the amount of data is reasonably low. For example a 9600 baud modem can transfer approximately one character every millisecond (1/1000'th second). If the interrupt latency, the amount of time that it takes between the hardware device raising the interrupt and the device driver's interrupt handling routine being called, is low (say 2 milliseconds) then the overall system impact of the data transfer is very low. The 9600 baud modem data transfer would only take 0.002% of the CPU's processing time. For high speed devices, such as hard disk controllers or ethernet devices the data transfer rate is a lot higher. A SCSI device can transfer up to 40 Mbytes of information per second.

Direct Memory Access, or DMA, was invented to solve this problem. A DMA controller allows devices to transfer data to or from the system's memory without the intervention of the processor. A PC's ISA DMA controller has 8 DMA channels of which 7 are available for use by the device drivers. Each DMA channel has associated with it a 16 bit address register and a 16 bit count register. To initiate a data transfer the device driver sets up the DMA channel's address and count registers together with the direction of the data transfer, read or write. It then tells the device that it may start the DMA when it wishes. When the transfer is complete the device interrupts the PC. Whilst the transfer is taking place the CPU is free to do other things.

Device drivers have to be careful when using DMA. First of all the DMA controller knows nothing of virtual memory, it only has access to the physical memory in the system. Therefore the memory that is being DMA'd to or from must be a contiguous block of physical memory. This means that you cannot DMA directly into the virtual address space of a process. You can however lock the process's physical pages into memory, preventing them from being swapped out to the swap device during a DMA operation. Secondly, the DMA controller cannot access the whole of physical memory. The DMA channel's address register represents the first 16 bits of the DMA address, the next 8 bits come from the page register. This means that DMA requests are limited to the bottom 16 Mbytes of memory.

DMA channels are scarce resources, there are only 7 of them, and they cannot be shared between device drivers. Just like interrupts, the device driver must be able to work out which DMA channel it should use. Like interrupts, some devices have a fixed DMA channel. The floppy device, for example, always uses DMA channel 2. Sometimes the DMA channel for a device can be set by jumpers; a number of ethernet devices use this technique. The more flexible devices can be told (via their CSRs) which DMA channels to use and, in this case, the device driver can simply pick a free DMA channel to use.

Linux tracks the usage of the DMA channels using a vector of `dma_chan` data structures (one per DMA channel). The `dma_chan` data structure contains just two fields, a pointer to a string describing the owner of the DMA channel and a flag indicating if the DMA channel is allocated or not. It is this vector of `dma_chan` data structures that is printed when you `cat /proc/dma`.

8.3 Memory

Device drivers have to be careful when using memory. As they are part of the Linux kernel they cannot use virtual memory. Each time a device driver runs, maybe as an interrupt is received or as a bottom half or task queue handler is scheduled, the `current` process may change. The device driver cannot rely on a particular process running even if it is doing work on its behalf. Like the rest of the kernel, device drivers use data structures to keep track of the device that it is controlling. These data structures can be statically allocated, part of the device driver's code, but that would be wasteful as it makes the kernel larger than it need be. Most device drivers allocate kernel, non-paged, memory to hold their data.

Linux provides kernel memory allocation and deallocation routines and it is these that the device drivers use. Kernel memory is allocated in chunks that are powers of 2. For example 128 or 512 bytes, even if the device driver asks for less. The number of bytes that the device driver requests is rounded up to the next block size boundary. This makes kernel memory deallocation easier as the smaller free blocks can be recombined into bigger blocks.

It may be that Linux needs to do quite a lot of extra work when the kernel memory is requested. If the amount of free memory is low, physical pages may need to be discarded or written to the swap device. Normally, Linux would suspend the requestor, putting the process onto a wait queue until there is enough physical memory. Not all device drivers (or indeed Linux kernel code) may want this to happen and so the kernel memory allocation routines can be requested to fail if they cannot immediately allocate memory. If the device driver wishes to DMA to or from the allocated memory it can also specify that the memory is DMA'able. This way it is the Linux kernel that needs to understand what constitutes DMA'able memory for this system, and not the device driver.

8.4 Interfacing Device Drivers with the Kernel

The Linux kernel must be able to interact with them in standard ways. Each class of device driver, character, block and network, provides common interfaces that the kernel uses when requesting services from them. These common interfaces mean that the kernel can treat often very different devices and their device drivers absolutely the same. For example, SCSI and IDE disks behave very differently but the Linux kernel uses the same interface to both of them.

Linux is very dynamic, every time a Linux kernel boots it may encounter different physical devices and thus need different device drivers. Linux allows you to include device drivers at kernel build time via its configuration scripts. When these drivers are initialized at boot time they may not discover any hardware to control. Other drivers can be loaded as kernel modules when they are needed. To cope with this dynamic nature of device drivers, device drivers register themselves with the kernel as they are initialized. Linux maintains tables of registered device drivers as part of its interfaces with them. These tables include pointers to routines and information that support the interface with that class of devices.

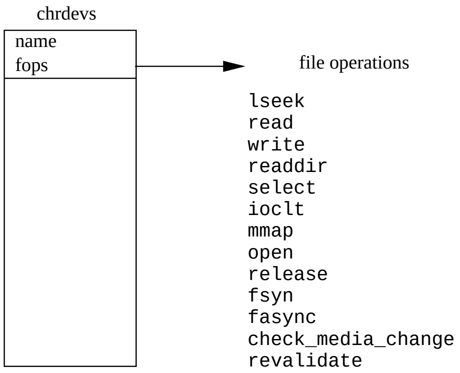

Figure 8.1: Character Devices

8.4.1 Character Devices

Character devices, the simplest of Linux's devices, are accessed as files, applications use standard system calls to open them, read from them, write to them and close them exactly as if the device were a file. This is true even if the device is a modem being used by the PPP daemon to connect a Linux system onto a network. As a character device is initialized its device driver registers itself with the Linux kernel by adding an entry into the `chrdevs` vector of `device_struct` data structures. The device's major device identifier (for example 4 for the `tty` device) is used as an index into this vector. The major device identifier for a device is fixed. Each entry in the `chrdevs` vector, a `device_struct` data structure contains two elements; a pointer to the name of the registered device driver and a pointer to a block of file operations. This block of file operations is itself the addresses of routines within the device character device driver each of which handles specific file operations such as open, read, write and close. The contents of `/proc/devices` for character devices is taken from the `chrdevs` vector.

See `include/linux/major.h`

When a character special file representing a character device (for example `/dev/cua0`) is opened the kernel must set things up so that the correct character device driver's file operation routines will be called. Just like an ordinairy file or directory, each device special file is represented by a VFS inode . The VFS inode for a character special file, indeed for all device special files, contains both the major and minor identifiers for the device. This VFS inode was created by the underlying filesystem, for example `EXT2`, from information in the real filesystem when the device special file's name was looked up.

See `ext2_read_inode()` in `fs/ext2/inode.c`

Each VFS inode has associated with it a set of file operations and these are different depending on the filesystem object that the inode represents. Whenever a VFS inode representing a character special file is created, its file operations are set to the default character device operations . This has only one file operation, the open file operation. When the character special file is opened by an application the generic open file operation uses the device's major identifier as an index into the `chrdevs` vector to retrieve the file operations block for this particular device. It also sets up the `file` data structure describing this character special file, making its file operations pointer point to those of the device driver. Thereafter all of the applications file operations will be mapped to calls to the character devices set of file operations.

`def_chr_fops`

See `chrdev_open()` in `fs/devices.c`

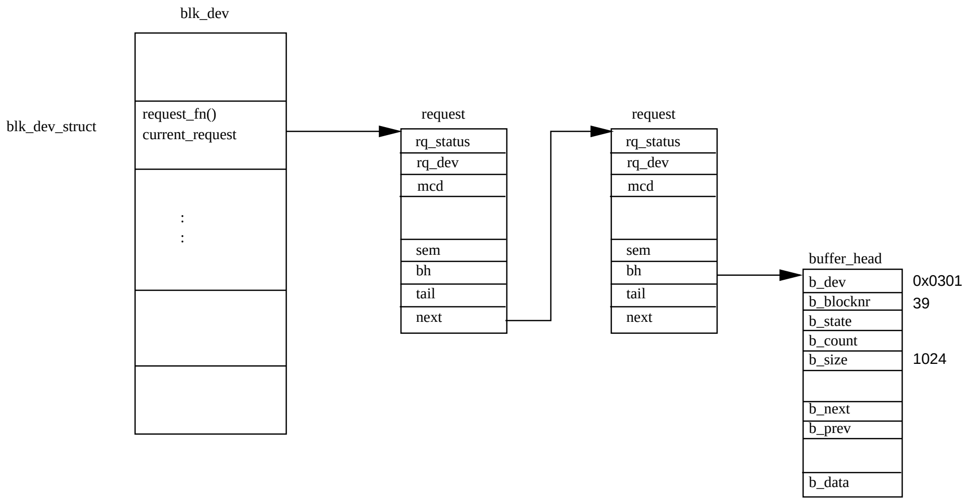

Figure 8.2: Buffer Cache Block Device Requests

8.4.2 Block Devices

Block devices also support being accessed like files. The mechanisms used to provide the correct set of file operations for the opened block special file are very much the same as for character devices. Linux maintains the set of registered block devices as the `blkdevs` vector. It, like the `chrdevs` vector, is indexed using the device's major device number. Its entries are also `device_struct` data structures. Unlike character devices, there are classes of block devices. SCSI devices are one such class and IDE devices are another. It is the class that registers itself with the Linux kernel and provides file operations to the kernel. The device drivers for a class of block device provide class specific interfaces to the class. So, for example, a SCSI device driver has to provide interfaces to the SCSI subsystem which the SCSI subsystem uses to provide file operations for this device to the kernel.

See `fs/devices.c`

Every block device driver must provide an interface to the buffer cache as well as the normal file operations interface. Each block device driver fills in its entry in the `blk_dev` vector of `blk_dev_struct` data structures . The index into this vector is, again, the device's major number. The `blk_dev_struct` data structure consists of the address of a request routine and a pointer to a list of `request` data structures, each one representing a request from the buffer cache for the driver to read or write a block of data.

See `drivers/block/-ll_rw_blk.c`

See `include/-linux/blkdev.h`

Each time the buffer cache wishes to read or write a block of data to or from a registered device it adds a `request` data structure onto its `blk_dev_struct`. Figure 8.2 shows that each request has a pointer to one or more `buffer_head` data structures, each one a request to read or write a block of data. The `buffer_head` structures are locked (by the buffer cache) and there may be a process waiting on the block operation to this buffer to complete. Each `request` structure is allocated from a static list, the `all_requests` list. If the request is being added to an empty request list, the driver's request function is called to start processing the request queue. Otherwise the driver will simply process every `request` on the request list.

Once the device driver has completed a request it must remove each of the `buffer_head` structures from the `request` structure, mark them as up to date and unlock them.

This unlocking of the `buffer_head` will wake up any process that has been sleeping waiting for the block operation to complete. An example of this would be where a file name is being resolved and the `EXT2` filesystem must read the block of data that contains the next `EXT2` directory entry from the block device that holds the filesystem. The process sleeps on the `buffer_head` that will contain the directory entry until the device driver wakes it up. The `request` data structure is marked as free so that it can be used in another block request.

8.5 Hard Disks

Disk drives provide a more permanent method for storing data, keeping it on spinning disk platters. To write data, a tiny head magnetizes minute particles on the platter's surface. The data is read by a head, which can detect whether a particular minute particle is magnetized.

A disk drive consists of one or more *platters*, each made of finely polished glass or ceramic composites and coated with a fine layer of iron oxide. The platters are attached to a central spindle and spin at a constant speed that can vary between 3000 and 10,000 RPM depending on the model. Compare this to a floppy disk which only spins at 360 RPM. The disk's read/write heads are responsible for reading and writing data and there is a pair for each platter, one head for each surface. The read/write heads do not physically touch the surface of the platters, instead they float on a very thin (10 millionths of an inch) cushion of air. The read/write heads are moved across the surface of the platters by an actuator. All of the read/write heads are attached together, they all move across the surfaces of the platters together.

Each surface of the platter is divided into narrow, concentric circles called *tracks*. Track 0 is the outermost track and the highest numbered track is the track closest to the central spindle. A *cylinder* is the set of all tracks with the same number. So all of the 5th tracks from each side of every platter in the disk is known as cylinder 5. As the number of cylinders is the same as the number of tracks, you often see disk geometries described in terms of cylinders. Each track is divided into *sectors*. A sector is the smallest unit of data that can be written to or read from a hard disk and it is also the disk's block size. A common sector size is 512 bytes and the sector size was set when the disk was formatted, usually when the disk is manufactured.

A disk is usually described by its geometry, the number of cylinders, heads and sectors. For example, at boot time Linux describes one of my IDE disks as:

```
hdb: Conner Peripherals 540MB - CFS540A, 516MB w/64kB Cache, CHS=1050/16/63
```

This means that it has 1050 cylinders (tracks), 16 heads (8 platters) and 63 sectors per track. With a sector, or block, size of 512 bytes this gives the disk a storage capacity of 529200 bytes. This does not match the disk's stated capacity of 516 Mbytes as some of the sectors are used for disk partitioning information. Some disks automatically find bad sectors and re-index the disk to work around them.

Hard disks can be further subdivided into *partitions*. A partition is a large group of sectors allocated for a particular purpose. Partitioning a disk allows the disk to be used by several operating system or for several purposes. A lot of Linux systems have a single disk with three partitions; one containing a DOS filesystem, another an EXT2 filesystem and a third for the swap partition. The partitions of a hard disk

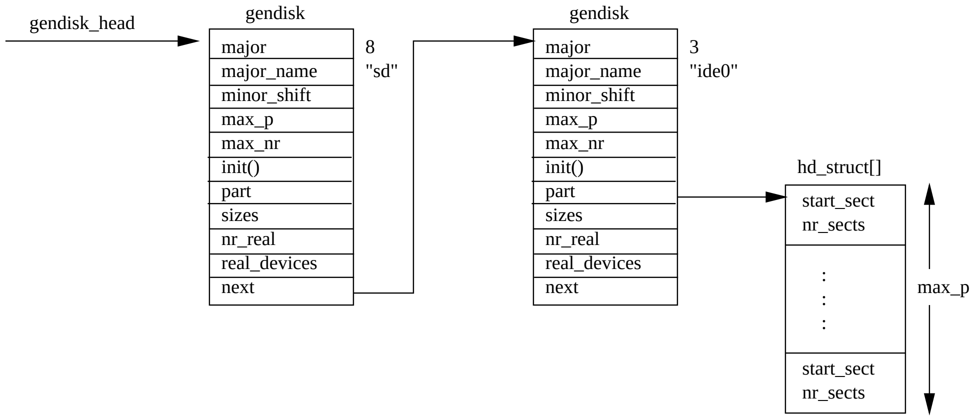

Figure 8.3: Linked list of disks

are described by a partition table; each entry describing where the partition starts and ends in terms of heads, sectors and cylinder numbers. For DOS formatted disks, those formatted by fdisk, there are four primary disk partitions. Not all four entries in the partition table have to be used. There are three types of partition supported by fdisk, primary, extended and logical. Extended partitions are not real partitions at all, they contain any number of logical paritititions. Extended and logical partitions were invented as a way around the limit of four primary partitions. The following is the output from fdisk for a disk containing two primary partitions:

```
Disk /dev/sda: 64 heads, 32 sectors, 510 cylinders
Units = cylinders of 2048 * 512 bytes

   Device Boot   Begin    Start      End   Blocks   Id  System
/dev/sda1            1        1      478   489456   83  Linux native
/dev/sda2          479      479      510    32768   82  Linux swap

Expert command (m for help): p

Disk /dev/sda: 64 heads, 32 sectors, 510 cylinders

Nr AF  Hd Sec  Cyl  Hd Sec  Cyl   Start    Size ID
 1 00   1   1    0  63  32  477      32  978912 83
 2 00   0   1  478  63  32  509  978944   65536 82
 3 00   0   0    0   0   0    0       0       0 00
 4 00   0   0    0   0   0    0       0       0 00
```

This shows that the first partition starts at cylinder or track 0, head 1 and sector 1 and extends to include cylinder 477, sector 32 and head 63. As there are 32 sectors in a track and 64 read/write heads, this partition is a whole number of cylinders in size. fdisk alligns partitions on cylinder boundaries by default. It starts at the outermost cylinder (0) and extends inwards, towards the spindle, for 478 cylinders. The second partition, the swap partition, starts at the next cylinder (478) and extends to the innermost cylinder of the disk.

During initialization Linux maps the topology of the hard disks in the system. It finds out how many hard disks there are and of what type. Additionally, Linux discovers how the individual disks have been partitioned. This is all represented by a list of `gendisk` data structures pointed at by the `gendisk_head` list pointer. As each disk subsystem, for example IDE, is initialized it generates `gendisk` data structures representing the disks that it finds. It does this at the same time as it registers its file operations and adds its entry into the `blk_dev` data structure. Each `gendisk` data structure has a unique major device number and these match the major numbers of the block special devices. For example, the SCSI disk subsystem creates a single `gendisk` entry (`''sd''`) with a major number of 8, the major number of all SCSI disk devices. Figure 8.3 shows two `gendisk` entries, the first one for the SCSI disk subsystem and the second for an IDE disk controller. This is `ide0`, the primary IDE controller.

Although the disk subsystems build the `gendisk` entries during their initialization they are only used by Linux during partition checking. Instead, each disk subsystem maintains its own data structures which allow it to map device special major and minor device numbers to partitions within physical disks. Whenever a block device is read from or written to, either via the buffer cache or file operations, the kernel directs the operation to the appropriate device using the major device number found in its block special device file (for example `/dev/sda2`). It is the individual device driver or subsystem that maps the minor device number to the real physical device.

8.5.1 IDE Disks

The most common disks used in Linux systems today are Integrated Disk Electronic or IDE disks. IDE is a disk interface rather than an I/O bus like SCSI. Each IDE controller can support up to two disks, one the master disk and the other the slave disk. The master and slave functions are usually set by jumpers on the disk. The first IDE controller in the system is known as the primary IDE controller, the next the secondary controller and so on. IDE can manage about 3.3 Mbytes per second of data transfer to or from the disk and the maximum IDE disk size is 538Mbytes. Extended IDE, or EIDE, has raised the disk size to a maximum of 8.6 Gbytes and the data transfer rate up to 16.6 Mbytes per second. IDE and EIDE disks are cheaper than SCSI disks and most modern PCs contain one or more on board IDE controllers.

Linux names IDE disks in the order in which it finds their controllers. The master disk on the primary controller is `/dev/hda` and the slave disk is `/dev/hdb`. `/dev/hdc` is the master disk on the secondary IDE controller. The IDE subsystem registers IDE controllers and *not* disks with the Linux kernel. The major identifier for the primary IDE controller is 3 and is 22 for the secondary IDE controller. This means that if a system has two IDE controllers there will be entries for the IDE subsystem at indices at 3 and 22 in the `blk_dev` and `blkdevs` vectors. The block special files for IDE disks reflect this numbering, disks `/dev/hda` and `/dev/hdb`, both connected to the primary IDE controller, have a major identifier of 3. Any file or buffer cache operations for the IDE subsystem operations on these block special files will be directed to the IDE subsystem as the kernel uses the major identifier as an index. When the request is made, it is up to the IDE subsystem to work out which IDE disk the request is for. To do this the IDE subsystem uses the minor device number from the device special identifier, this contains information that allows it to direct the request to the correct partition of the correct disk. The device identifier for `/dev/hdb`, the slave IDE drive on the primary IDE controller is *(3,64)*. The device identifier for the first partition of that disk (`/dev/hdb1`) is *(3,65)*.

8.5.2 Initializing the IDE Subsystem

IDE disks have been around for much of the IBM PC's history. Throughout this time the interface to these devices has changed. This makes the initialization of the IDE subsystem more complex than it might at first appear.

The maximum number of IDE controllers that Linux can support is 4. Each controller is represented by an `ide_hwif_t` data structure in the `ide_hwifs` vector. Each `ide_hwif_t` data structure contains two `ide_drive_t` data structures, one per possible supported master and slave IDE drive. During the initializing of the IDE subsystem, Linux first looks to see if there is information about the disks present in the system's CMOS memory. This is battery backed memory that does not lose its contents when the PC is powered off. This CMOS memory is actually in the system's real time clock device which always runs no matter if your PC is on or off. The CMOS memory locations are set up by the system's BIOS and tell Linux what IDE controllers and drives have been found. Linux retrieves the found disk's geometry from BIOS and uses the information to set up the `ide_hwif_t` data structure for this drive. More modern PCs use PCI chipsets such as Intel's 82430 VX chipset which includes a PCI EIDE controller. The IDE subsystem uses PCI BIOS callbacks to locate the PCI (E)IDE controllers in the system. It then calls PCI specific interrogation routines for those chipsets that are present.

Once each IDE interface or controller has been discovered, its `ide_hwif_t` is set up to reflect the controllers and attached disks. During operation the IDE driver writes commands to IDE command registers that exist in the I/O memory space. The default I/O address for the primary IDE controller's control and status registers is *0x1F0 - 0x1F7*. These addresses were set by convention in the early days of the IBM PC. The IDE driver registers each controller with the Linux block buffer cache and VFS, adding it to the `blk_dev` and `blkdevs` vectors respectively. The IDE drive will also request control of the appropriate interrupt. Again these interrupts are set by convention to be 14 for the primary IDE controller and 15 for the secondary IDE controller. However, they like all IDE details, can be overridden by command line options to the kernel. The IDE driver also adds a `gendisk` entry into the list of `gendisk`'s discovered during boot for each IDE controller found. This list will later be used to discover the partition tables of all of the hard disks found at boot time. The partition checking code understands that IDE controllers may each control two IDE disks.

8.5.3 SCSI Disks

The SCSI (Small Computer System Interface) bus is an efficient peer-to-peer data bus that supports up to eight devices per bus, including one or more hosts. Each device has to have a unique identifier and this is usually set by jumpers on the disks. Data can be transfered synchronously or asynchronously between any two devices on the bus and with 32 bit wide data transfers up to 40 Mbytes per second are possible. The SCSI bus transfers both data and state information between devices, and a single transaction between an *initiator* and a *target* can involve up to eight distinct phases. You can tell the current phase of a SCSI bus from five signals from the bus. The eight phases are:

BUS FREE No device has control of the bus and there are no transactions currently happening,

ARBITRATION A SCSI device has attempted to get control of the SCSI bus, it does this by asserting its SCSI identifer onto the address pins. The highest number SCSI identifier wins.

SELECTION When a device has succeeded in getting control of the SCSI bus through arbitration it must now signal the target of this SCSI request that it wants to send a command to it. It does this by asserting the SCSI identifier of the target on the address pins.

RESELECTION SCSI devices may disconnect during the processing of a request. The target may then reselect the initiator. Not all SCSI devices support this phase.

COMMAND 6,10 or 12 bytes of command can be transfered from the initiator to the target,

DATA IN, DATA OUT During these phases data is transfered between the initiator and the target,

STATUS This phase is entered after completion of all commands and allows the target to send a status byte indicating success or failure to the initiator,

MESSAGE IN, MESSAGE OUT Additional information is transfered between the initiator and the target.

The Linux SCSI subsystem is made up of two basic elements, each of which is represented by data structures:

host A SCSI host is a physical piece of hardware, a SCSI controller. The NCR810 PCI SCSI controller is an example of a SCSI host. If a Linux system has more than one SCSI controller of the same type, each instance will be represented by a separate SCSI host. This means that a SCSI device driver may control more than one instance of its controller. SCSI hosts are almost always the *initiator* of SCSI commands.

Device The most common set of SCSI device is a SCSI disk but the SCSI standard supports several more types; tape, CD-ROM and also a generic SCSI device. SCSI devices are almost always the *targets* of SCSI commands. These devices must be treated differently, for example with removable media such as CD-ROMs or tapes, Linux needs to detect if the media was removed. The different disk types have different major device numbers, allowing Linux to direct block device requests to the appropriate SCSI type.

Initializing the SCSI Subsystem

Initializing the SCSI subsystem is quite complex, reflecting the dynamic nature of SCSI buses and their devices. Linux initializes the SCSI subsystem at boot time; it finds the SCSI controllers (known as SCSI hosts) in the system and then probes each of their SCSI buses finding all of their devices. It then initializes those devices and makes them available to the rest of the Linux kernel via the normal file and buffer cache block device operations. This initialization is done in four phases:

First, Linux finds out which of the SCSI host adapters, or controllers, that were built into the kernel at kernel build time have hardware to control. Each built in

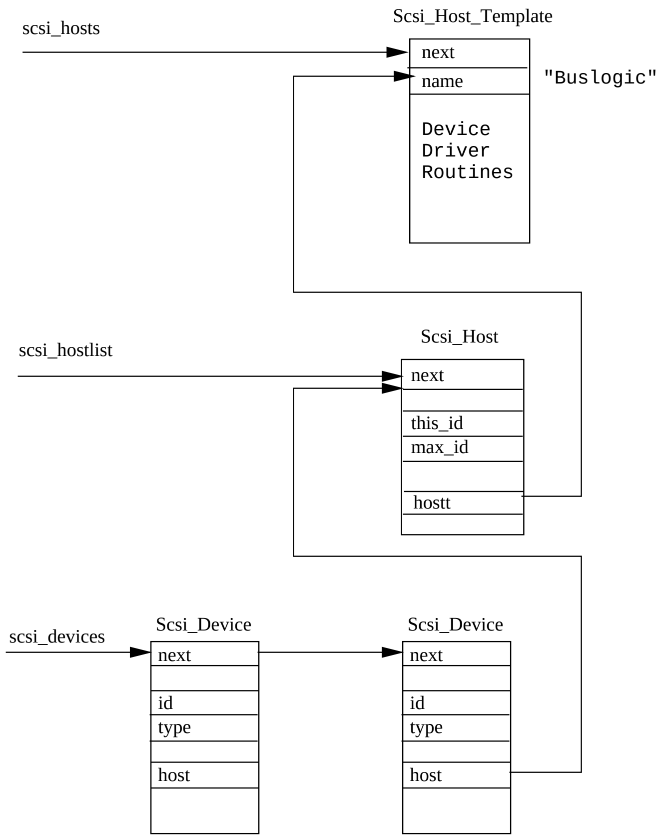

Figure 8.4: SCSI Data Structures

SCSI host has a `Scsi_Host_Template` entry in the `builtin_scsi_hosts` vector The `Scsi_Host_Template` data structure contains pointers to routines that carry out SCSI host specific actions such as detecting what SCSI devices are attached to this SCSI host. These routines are called by the SCSI subsystem as it configures itself and they are part of the SCSI device driver supporting this host type. Each detected SCSI host, those for which there are real SCSI devices attached, has its `Scsi_Host_Template` data structure added to the `scsi_hosts` list of active SCSI hosts. Each instance of a detected host type is represented by a `Scsi_Host` data structure held in the `scsi_hostlist` list. For example a system with two NCR810 PCI SCSI controllers would have two `Scsi_Host` entries in the list, one per controller. Each `Scsi_Host` points at the `Scsi_Host_Template` representing its device driver.

Now that every SCSI host has been discovered, the SCSI subsystem must find out what SCSI devices are attached to each host's bus. SCSI devices are numbered between 0 and 7 inclusively, each device's number or SCSI identifier being unique on the SCSI bus to which it is attached. SCSI identifiers are usually set by jumpers on the device. The SCSI initialization code finds each SCSI device on a SCSI bus by sending it a TEST_UNIT_READY command. When a device responds, its identification is read by sending it an ENQUIRY command. This gives Linux the vendor's name and the device's model and revision names. SCSI commands are represented by a `Scsi_Cmnd` data structure and these are passed to the device driver for this SCSI host by calling the device driver routines within its `Scsi_Host_Template` data structure. Every SCSI device that is found is represented by a `Scsi_Device` data structure, each of which points to its parent `Scsi_Host`. All of the `Scsi_Device` data structures are added to the `scsi_devices` list. Figure 8.4 shows how the main data structures relate to one another.

There are four SCSI device types: disk, tape, CD and generic. Each of these SCSI types are individually registered with the kernel as different major block device types. However they will only register themselves if one or more of a given SCSI device type has been found. Each SCSI type, for example SCSI disk, maintains its own tables of devices. It uses these tables to direct kernel block operations (file or buffer cache) to the correct device driver or SCSI host. Each SCSI type is represented by a `Scsi_Device_Template` data structure. This contains information about this type of SCSI device and the addresses of routines to perform various tasks. The SCSI subsystem uses these templates to call the SCSI type routines for each type of SCSI device. In other words, if the SCSI subsystem wishes to attach a SCSI disk device it will call the SCSI disk type attach routine. The `Scsi_Type_Template` data structures are added to the `scsi_devicelist` list if one or more SCSI devices of that type have been detected.

The final phase of the SCSI subsystem initialization is to call the finish functions for each registered `Scsi_Device_Template`. For the SCSI disk type this spins up all of the SCSI disks that were found and then records their disk geometry. It also adds the `gendisk` data structure representing all SCSI disks to the linked list of disks shown in Figure 8.3.

Delivering Block Device Requests

Once Linux has initialized the SCSI subsystem, the SCSI devices may be used. Each active SCSI device type registers itself with the kernel so that Linux can direct block device requests to it. There can be buffer cache requests via `blk_dev` or file operations via `blkdevs`. Taking a SCSI disk driver that has one or more EXT2 filesystem partitions as an example, how do kernel buffer requests get directed to the right SCSI disk when one of its EXT2 partitions is mounted?

Each request to read or write a block of data to or from a SCSI disk partition results in a new `request` structure being added to the SCSI disks `current_request` list in the `blk_dev` vector. If the `request` list is being processed, the buffer cache need not do anything else; otherwise it must nudge the SCSI disk subsystem to go and process its request queue. Each SCSI disk in the system is represented by a `Scsi_Disk` data structure. These are kept in the `rscsi_disks` vector that is indexed using part of the SCSI disk partition's minor device number. For exmaple, `/dev/sdb1` has a major number of 8 and a minor number of 17; this generates an index of 1. Each `Scsi_Disk` data structure contains a pointer to the `Scsi_Device` data structure representing this device. That in turn points at the `Scsi_Host` data structure which "owns" it. The `request` data structures from the buffer cache are translated into `Scsi_Cmd` structures describing the SCSI command that needs to be sent to the SCSI device and this is queued onto the `Scsi_Host` structure representing this device. These will be processed by the individual SCSI device driver once the appropriate data blocks have been read or written.

8.6 Network Devices

A network device is, so far as Linux's network subsystem is concerned, an entity that sends and receives packets of data. This is normally a physical device such as an ethernet card. Some network devices though are software only such as the loopback device which is used for sending data to yourself. Each network device is represented by a `device` data structure. Network device drivers register the devices that they control with Linux during network initialization at kernel boot time. The `device` data structure contains information about the device and the addresses of functions that allow the various supported network protocols to use the device's services. These functions are mostly concerned with transmitting data using the network device. The device uses standard networking support mechanisms to pass received data up to the appropriate protocol layer. All network data (packets) transmitted and received are represented by `sk_buff` data structures, these are flexible data structures that allow network protocol headers to be easily added and removed. How the network protocol layers use the network devices, how they pass data back and forth using `sk_buff` data structures is described in detail in the Networks chapter (Chapter 10). This chapter concentrates on the `device` data structure and on how network devices are discovered and initialized.

See `include/linux/-netdevice.h`

The `device` data structure contains information about the network device:

Name Unlike block and character devices which have their device special files created using the mknod command, network device special files appear spontaniously as the system's network devices are discovered and initialized. Their names are standard, each name representing the type of device that it is. Multiple devices of the same type are numbered upwards from 0. Thus the ethernet devices are known as `/dev/eth0`,`/dev/eth1`,`/dev/eth2` and so on. Some common network devices are:

/dev/ethN	Ethernet devices
/dev/slN	SLIP devices
/dev/pppN	PPP devices
/dev/lo	Loopback devices

Bus Information This is information that the device driver needs in order to control the device. The *irq* number is the interrupt that this device is using. The *base address* is the address of any of the device's control and status registers in I/O memory. The *DMA channel* is the DMA channel number that this network device is using. All of this information is set at boot time as the device is initialized.

Interface Flags These describe the characteristics and abilities of the network device:

IFF_UP	Interface is up and running,
IFF_BROADCAST	Broadcast address in `device` is valid
IFF_DEBUG	Device debugging turned on
IFF_LOOPBACK	This is a loopback device
IFF_POINTTOPOINT	This is point to point link (SLIP and PPP)
IFF_NOTRAILERS	No network trailers
IFF_RUNNING	Resources allocated
IFF_NOARP	Does not support ARP protocol
IFF_PROMISC	Device in promiscuous receive mode, it will receive all packets no matter who they are addressed to
IFF_ALLMULTI	Receive all IP multicast frames
IFF_MULTICAST	Can receive IP multicast frames

Protocol Information Each device describes how it may be used by the network protocool layers:

mtu The size of the largest packet that this network can transmit not including any link layer headers that it needs to add. This maximum is used by the protocol layers, for example IP, to select suitable packet sizes to send.

Family The family indicates the protocol family that the device can support. The family for all Linux network devices is AF_INET, the Internet address family.

Type The hardware interface type describes the media that this network device is attached to. There are many different types of media that Linux network devices support. These include Ethernet, X.25, Token Ring, Slip, PPP and Apple Localtalk.

Addresses The `device` data structure holds a number of addresses that are relevent to this network device, including its IP addresses.

Packet Queue This is the queue of `sk_buff` packets queued waiting to be transmitted on this network device,

Support Functions Each device provides a standard set of routines that protocol layers call as part of their interface to this device's link layer. These include setup and frame transmit routines as well as routines to add standard frame headers and collect statistics. These statistics can be seen using the ifconfig command.

8.6.1 Initializing Network Devices

Network device drivers can, like other Linux device drivers, be built into the Linux kernel. Each potential network device is represented by a `device` data structure within the network device list pointed at by `dev_base` list pointer. The network layers call one of a number of network device service routines whose addresses are held in the `device` data structure if they need device specific work performing. Initially though, each `device` data structure holds only the address of an initialization or probe routine.

There are two problems to be solved for network device drivers. Firstly, not all of the network device drivers built into the Linux kernel will have devices to control. Secondly, the ethernet devices in the system are always called `/dev/eth0`, `/dev/eth1` and so on, no matter what their underlying device drivers are. The problem of "missing" network devices is easily solved. As the initialization routine for each network device is called, it returns a status indicating whether or not it located an instance of the controller that it is driving. If the driver could not find any devices, its entry in the `device` list pointed at by `dev_base` is removed. If the driver could find a device it fills out the rest of the `device` data structure with information about the device and the addresses of the support functions within the network device driver.

The second problem, that of dynamically assigning ethernet devices to the standard `/dev/ethN` device special files is solved more elegantly. There are eight standard entries in the devices list; one for `eth0`, `eth1` and so on to `eth7`. The initialization routine is the same for all of them, it tries each ethernet device driver built into the kernel in turn until one finds a device. When the driver finds its ethernet device it fills out the `ethN device` data structure, which it now owns. It is also at this time that the network device driver initializes the physical hardware that it is controlling and works out which IRQ it is using, which DMA channel (if any) and so on. A driver may find several instances of the network device that it is controlling and, in this case, it will take over several of the `/dev/ethN device` data structures. Once all eight standard `/dev/ethN` have been allocated, no more ethernet devices will be probed for.

Chapter 9

The File system

This chapter describes how the Linux kernel maintains the files in the file systems that it supports. It describes the Virtual File System (VFS) and explains how the Linux kernel's real file systems are supported.

One of the most important features of Linux is its support for many different file systems. This makes it very flexible and well able to coexist with many other operating systems. At the time of writing, Linux supports 15 file systems; `ext`, `ext2`, `xia`, `minix`, `umsdos`, `msdos`, `vfat`, `proc`, `smb`, `ncp`, `iso9660`, `sysv`, `hpfs`, `affs` and `ufs`, and no doubt, over time more will be added.

In Linux, as it is for Unix™, the separate file systems the system may use are not accessed by device identifiers (such as a drive number or a drive name) but instead they are combined into a single hierarchical tree structure that represents the file system as one whole single entity. Linux adds each new file system into this single file system tree as it is mounted. All file systems, of whatever type, are mounted onto a directory and the files of the mounted file system cover up the existing contents of that directory. This directory is known as the mount directory or mount point. When the file system is unmounted, the mount directory's own files are once again revealed.

When disks are initialized (using `fdisk`, say) they have a partition structure imposed on them that divides the physical disk into a number of logical partitions. Each partition may hold a single file system, for example an `EXT2` file system. File systems organize files into logical hierarchical structures with directories, soft links and so on held in blocks on physical devices. Devices that can contain file systems are known as block devices. The IDE disk partition `/dev/hda1`, the first partition of the first IDE disk drive in the system, is a block device. The Linux file systems regard these block devices as simply linear collections of blocks, they do not know or care about the underlying physical disk's geometry. It is the task of each block device driver to map a request to read a particular block of its device into terms meaningful to its device; the particular track, sector and cylinder of its hard disk where the block is kept. A file system has to look, feel and operate in the same way no matter what device is holding it. Moreover, using Linux's file systems, it does not matter (at least to the system user) that these different file systems are on different physical media controlled by different hardware controllers. The file system might not even be on the local system, it could just as well be a disk remotely mounted over a network link. Consider the following example where a Linux system has its root file system on a SCSI disk:

```
A         E         boot      etc       lib       opt       tmp       usr
C         F         cdrom     fd        proc      root      var       sbin
D         bin       dev       home      mnt       lost+found
```

Neither the users nor the programs that operate on the files themselves need know that `/C` is in fact a mounted VFAT file system that is on the first IDE disk in the system. In the example (which is actually my home Linux system), `/E` is the master IDE disk on the second IDE controller. It does not matter either that the first IDE controller is a PCI controller and that the second is an ISA controller which also controls the IDE CDROM. I can dial into the network where I work using a modem and the PPP network protocol using a modem and in this case I can remotely mount my Alpha AXP Linux system's file systems on `/mnt/remote`.

The files in a file system are collections of data; the file holding the sources to this chapter is an ASCII file called `filesystems.tex`. A file system not only holds the data that is contained within the files of the file system but also the structure of the file system. It holds all of the information that Linux users and processes see as files, directories soft links, file protection information and so on. Moreover it must hold that information safely and securely, the basic integrity of the operating system depends on its file systems. Nobody would use an operating system that randomly lost data and files[1].

`Minix`, the first file system that Linux had is rather restrictive and lacking in performance. Its filenames cannot be longer than 14 characters (which is still better than 8.3 filenames) and the maximum file size is 64MBytes. 64Mbytes might at first glance seem large enough but large file sizes are necessary to hold even modest databases. The first file system designed specifically for Linux, the Extended File system, or `EXT`, was introduced in April 1992 and cured a lot of the problems but it was still felt to lack performance. So, in 1993, the Second Extended File system, or `EXT2`, was added. It is this file system that is described in detail later on in this chapter.

An important development took place when the EXT file system was added into Linux. The real file systems were separated from the operating system and system services by an interface layer known as the Virtual File system, or VFS. VFS allows Linux to support many, often very different, file systems, each presenting a common software interface to the VFS. All of the details of the Linux file systems are translated by software so that all file systems appear identical to the rest of the Linux kernel and to programs running in the system. Linux's Virtual File system layer allows you to transparently mount the many different file systems at the same time.

The Linux Virtual File system is implemented so that access to its files is as fast and efficient as possible. It must also make sure that the files and their data are kept

[1] Well, not knowingly, although I have been bitten by operating systems with more lawyers than Linux has developers

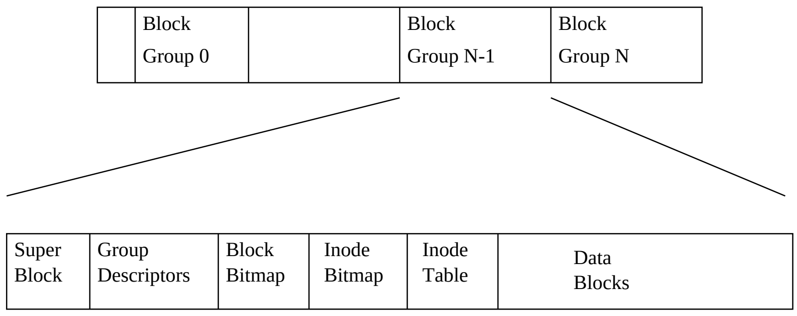

Figure 9.1: Physical Layout of the EXT2 File system

correctly. These two requirements can be at odds with each other. The Linux VFS caches information in memory from each file system as it is mounted and used. A lot of care must be taken to update the file system correctly as data within these caches is modified as files and directories are created, written to and deleted. If you could see the file system's data structures within the running kernel, you would be able to see data blocks being read and written by the file system. Data structures, describing the files and directories being accessed would be created and destroyed and all the time the device drivers would be working away, fetching and saving data. The most important of these caches is the Buffer Cache, which is integrated into the way that the individual file systems access their underlying block devices. As blocks are accessed they are put into the Buffer Cache and kept on various queues depending on their states. The Buffer Cache not only caches data buffers, it also helps manage the asynchronous interface with the block device drivers.

9.1 The Second Extended File system (EXT2)

The Second Extended File system was devised (by Rémy Card) as an extensible and powerful file system for Linux. It is also the most successful file system so far in the Linux community and is the basis for all of the currently shipping Linux distributions. The EXT2 file system, like a lot of the file systems, is built on the premise that the data held in files is kept in data blocks. These data blocks are all of the same length and, although that length can vary between different EXT2 file systems the block size of a particular EXT2 file system is set when it is created (using mke2fs). Every file's size is rounded up to an integral number of blocks. If the block size is 1024 bytes, then a file of 1025 bytes will occupy two 1024 byte blocks. Unfortunately this means that on average you waste half a block per file. Usually in computing you trade off CPU usage for memory and disk space utilisation. In this case Linux, along with most operating systems, trades off a relatively inefficient disk usage in order to reduce the workload on the CPU. Not all of the blocks in the file system hold data, some must be used to contain the information that describes the structure of the file system. EXT2 defines the file system topology by describing each file in the system with an inode data structure. An inode describes which blocks the data within a file occupies as well as the access rights of the file, the file's modification times and the type of the file. Every file in the EXT2 file system is described by a single inode and each inode has a single unique number identifying it. The inodes for the file system are all kept together in inode tables. EXT2 directories are simply special

See fs/ext2/*

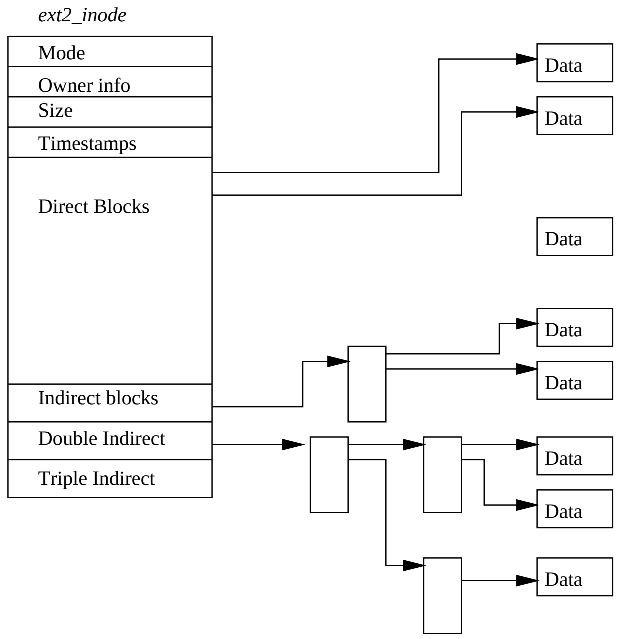

Figure 9.2: EXT2 Inode

files (themselves described by inodes) which contain pointers to the inodes of their directory entries.

Figure 9.1 shows the layout of the EXT2 file system as occupying a series of blocks in a block structured device. So far as each file system is concerned, block devices are just a series of blocks that can be read and written. A file system does not need to concern itself with where on the physical media a block should be put, that is the job of the device's driver. Whenever a file system needs to read information or data from the block device containing it, it requests that its supporting device driver reads an integral number of blocks. The EXT2 file system divides the logical partition that it occupies into Block Groups. Each group duplicates information critical to the integrity of the file system as well as holding real files and directories as blocks of information and data. This duplication is neccessary should a disaster occur and the file system need recovering. The subsections describe in more detail the contents of each Block Group.

9.1.1 The EXT2 Inode

In the `EXT2` file system, the inode is the basic building block; every file and directory in the file system is described by one and only one inode. The EXT2 inodes for each Block Group are kept in the inode table together with a bitmap that allows the system to keep track of allocated and unallocated inodes. Figure 9.2 shows the format of an EXT2 inode, amongst other information, it contains the following fields:

See `include/linux/-ext2_fs_i.h`

mode This holds two pieces of information; what this inode describes and the permissions that users have to it. For EXT2, an inode can describe one of file, directory, symbolic link, block device, character device or FIFO.

Owner Information The user and group identifiers of the owners of this file or directory. This allows the file system to correctly allow the right sort of accesses,

Size The size of the file in bytes,

Timestamps The time that the inode was created and the last time that it was modified,

Datablocks Pointers to the blocks that contain the data that this inode is describing. The first twelve are pointers to the physical blocks containing the data described by this inode and the last three pointers contain more and more levels of indirection. For example, the double indirect blocks pointer points at a block of pointers to blocks of pointers to data blocks. This means that files less than or equal to twelve data blocks in length are more quickly accessed than larger files.

You should note that EXT2 inodes can describe special device files. These are not real files but handles that programs can use to access devices. All of the device files in `/dev` are there to allow programs to access Linux's devices. For example the `mount` program takes as an argument the device file that it wishes to mount.

9.1.2 The EXT2 Superblock

The Superblock contains a description of the basic size and shape of this file system. The information within it allows the file system manager to use and maintain the file system. Usually only the Superblock in Block Group 0 is read when the file system is mounted but each Block Group contains a duplicate copy in case of file system corruption. Amongst other information it holds the:

See `include/linux/ext2_fs_sb.h`

Magic Number This allows the mounting software to check that this is indeed the Superblock for an EXT2 file system. For the current version of `EXT2` this is *0xEF53*.

Revision Level The major and minor revision levels allow the mounting code to determine whether or not this file system supports features that are only available in particular revisions of the file system. There are also feature compatibility fields which help the mounting code to determine which new features can safely be used on this file system,

Mount Count and Maximum Mount Count Together these allow the system to determine if the file system should be fully checked. The mount count is incremented each time the file system is mounted and when it equals the maximum mount count the warning message "maximal mount count reached, running e2fsck is recommended" is displayed,

Block Group Number The Block Group number that holds this copy of the Superblock,

Block Size The size of the block for this file system in bytes, for example 1024 bytes,

Blocks per Group The number of blocks in a group. Like the block size this is fixed when the file system is created,

Free Blocks The number of free blocks in the file system,

Free Inodes The number of free Inodes in the file system,

First Inode This is the inode number of the first inode in the file system. The first inode in an EXT2 root file system would be the directory entry for the '/' directory.

9.1.3 The EXT2 Group Descriptor

See `ext2_group_desc` in `include/-linux/ext2_fs.h`

Each Block Group has a data structure describing it. Like the Superblock, all the group descriptors for all of the Block Groups are duplicated in each Block Group in case of file system corruption. Each Group Descriptor contains the following information:

Blocks Bitmap The block number of the block allocation bitmap for this Block Group. This is used during block allocation and deallocation,

Inode Bitmap The block number of the inode allocation bitmap for this Block Group. This is used during inode allocation and deallocation,

Inode Table The block number of the starting block for the inode table for this Block Group. Each inode is represented by the EXT2 inode data structure described below.

Free blocks count, Free Inodes count, Used directory count

The group descriptors are placed on after another and together they make the group descriptor table. Each Blocks Group contains the entire table of group descriptors after its copy of the Superblock. Only the first copy (in Block Group 0) is actually used by the EXT2 file system. The other copies are there, like the copies of the Superblock, in case the main copy is corrupted.

9.1.4 EXT2 Directories

See `ext2_dir_entry` in `include/-linux/ext2_fs.h`

In the EXT2 file system, directories are special files that are used to create and hold access paths to the files in the file system. Figure 9.3 shows the layout of a directory entry in memory. A directory file is a list of directory entries, each one containing the following information:

inode The inode for this directory entry. This is an index into the array of inodes held in the Inode Table of the Block Group. In figure 9.3, the directory entry for the file called `file` has a reference to inode number `i1`,

name length The length of this directory entry in bytes,

name The name of this directory entry.

The first two entries for every directory are always the standard "." and ".." entries meaning "this directory" and "the parent directory" respectively.

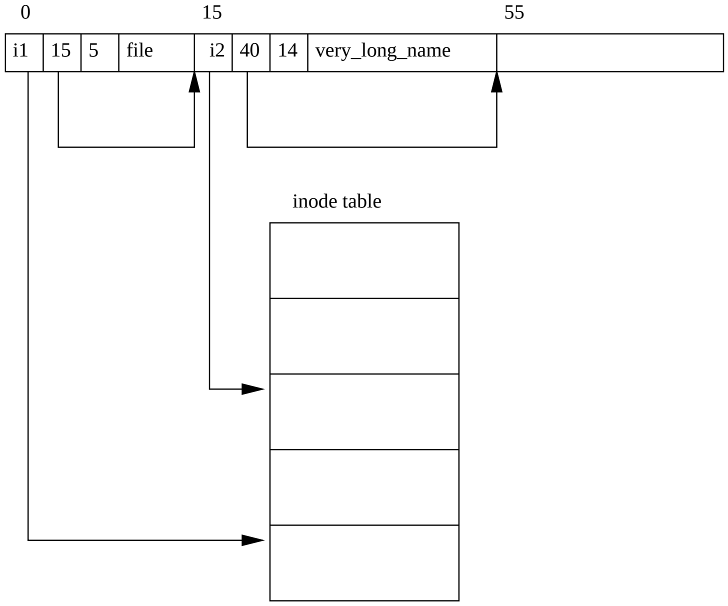

Figure 9.3: EXT2 Directory

9.1.5 Finding a File in an EXT2 File System

A Linux filename has the same format as all Unix™ filenames have. It is a series of directory names separated by forward slashes ("/") and ending in the file's name. One example filename would be `/home/rusling/.cshrc` where `/home` and `/rusling` are directory names and the file's name is `.cshrc`. Like all other Unix™ systems, Linux does not care about the format of the filename itself; it can be any length and consist of any of the printable characters. To find the inode representing this file within an `EXT2` file system the system must parse the filename a directory at a time until we get to the file itself.

The first inode we need is the inode for the root of the file system and we find its number in the file system's superblock. To read an EXT2 inode we must look for it in the inode table of the appropriate Block Group. If, for example, the root inode number is 42, then we need the 42nd inode from the inode table of Block Group 0. The root inode is for an EXT2 directory, in other words the mode of the root inode describes it as a directory and it's data blocks contain EXT2 directory entries.

`home` is just one of the many directory entries and this directory entry gives us the number of the inode describing the `/home` directory. We have to read this directory (by first reading its inode and then reading the directory entries from the data blocks described by its inode) to find the `rusling` entry which gives us the number of the inode describing the `/home/rusling` directory. Finally we read the directory entries pointed at by the inode describing the `/home/rusling` directory to find the inode number of the `.cshrc` file and from this we get the data blocks containing the information in the file.

9.1.6 Changing the Size of a File in an EXT2 File System

One common problem with a file system is its tendency to fragment. The blocks that hold the file's data get spread all over the file system and this makes sequentially accessing the data blocks of a file more and more inefficient the further apart the data blocks are. The EXT2 file system tries to overcome this by allocating the new blocks for a file physically close to its current data blocks or at least in the same Block Group as its current data blocks. Only when this fails does it allocate data blocks in another Block Group.

Whenever a process attempts to write data into a file the Linux file system checks to see if the data has gone off the end of the file's last allocated block. If it has, then it must allocate a new data block for this file. Until the allocation is complete, the process cannot run; it must wait for the file system to allocate a new data block and write the rest of the data to it before it can continue. The first thing that the EXT2 block allocation routines do is to lock the EXT2 Superblock for this file system. Allocating and deallocating changes fields within the superblock, and the Linux file system cannot allow more than one process to do this at the same time. If another process needs to allocate more data blocks, it will have to wait until this process has finished. Processes waiting for the superblock are suspended, unable to run, until control of the superblock is relinquished by its current user. Access to the superblock is granted on a first come, first served basis and once a process has control of the superblock, it keeps control until it has finished. Having locked the superblock, the process checks that there are enough free blocks left in this file system. If there are not enough free blocks, then this attempt to allocate more will fail and the process will relinquish control of this file system's superblock.

See `ext2_new_block()` in `fs/ext2/balloc.c`

If there are enough free blocks in the file system, the process tries to allocate one. If the EXT2 file system has been built to preallocate data blocks then we may be able to take one of those. The preallocated blocks do not actually exist, they are just reserved within the allocated block bitmap. The VFS inode representing the file that we are trying to allocate a new data block for has two EXT2 specific fields, `prealloc_block` and `prealloc_count`, which are the block number of the first preallocated data block and how many of them there are, respectively. If there were no preallocated blocks or block preallocation is not enabled, the EXT2 file system must allocate a new block. The EXT2 file system first looks to see if the data block after the last data block in the file is free. Logically, this is the most efficient block to allocate as it makes sequential accesses much quicker. If this block is not free, then the search widens and it looks for a data block within 64 blocks of the of the ideal block. This block, although not ideal is at least fairly close and within the same Block Group as the other data blocks belonging to this file.

If even that block is not free, the process starts looking in all of the other Block Groups in turn until it finds some free blocks. The block allocation code looks for a cluster of eight free data blocks somewhere in one of the Block Groups. If it cannot find eight together, it will settle for less. If block preallocation is wanted and enabled it will update `prealloc_block` and `prealloc_count` accordingly.

Wherever it finds the free block, the block allocation code updates the Block Group's block bitmap and allocates a data buffer in the buffer cache. That data buffer is uniquely identified by the file system's supporting device identifier and the block number of the allocated block. The data in the buffer is zero'd and the buffer is marked as "dirty" to show that it's contents have not been written to the physical

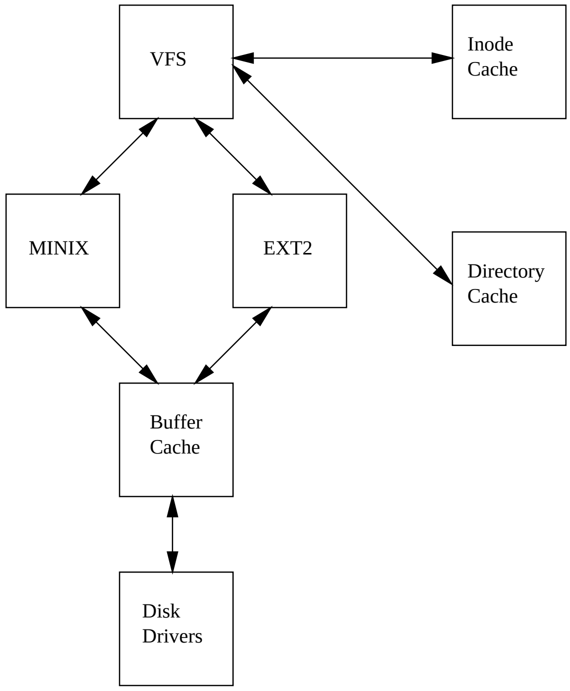

Figure 9.4: A Logical Diagram of the Virtual File System

disk. Finally, the superblock itself is marked as “dirty” to show that it has been changed and it is unlocked. If there were any processes waiting for the superblock, the first one in the queue is allowed to run again and will gain exclusive control of the superblock for its file operations. The process's data is written to the new data block and, if that data block is filled, the entire process is repeated and another data block allocated.

9.2 The Virtual File System (VFS)

Figure 9.4 shows the relationship between the Linux kernel's Virtual File System and it's real file systems. The virtual file system must manage all of the different file systems that are mounted at any given time. To do this it maintains data structures that describe the whole (virtual) file system and the real, mounted, file systems. Rather confusingly, the VFS describes the system's files in terms of superblocks and inodes in much the same way as the EXT2 file system uses superblocks and inodes. Like the EXT2 inodes, the VFS inodes describe files and directories within the system; the contents and topology of the Virtual File System. From now on, to avoid confusion, I will write about VFS inodes and VFS superblocks to distinquish them from EXT2 inodes and superblocks.

See `fs/*`

As each file system is initialised, it registers itself with the VFS. This happens as the operating system initialises itself at system boot time. The real file systems are either built into the kernel itself or are built as loadable modules. File System modules are loaded as the system needs them, so, for example, if the `VFAT` file system is implemented as a kernel module, then it is only loaded when a `VFAT` file system is mounted. When a block device based file system is mounted, and this includes the root file system, the VFS must read its superblock. Each file system type's superblock read routine must work out the file system's topology and map that information onto a VFS superblock data structure. The VFS keeps a list of the mounted file systems in the system together with their VFS superblocks. Each VFS superblock contains information and pointers to routines that perform particular functions. So, for example, the superblock representing a mounted EXT2 file system contains a pointer to the EXT2 specific inode reading routine. This EXT2 inode read routine, like all of the file system specific inode read routines, fills out the fields in a VFS inode. Each VFS superblock contains a pointer to the first VFS inode on the file system. For the root file system, this is the inode that represents the ``/'' directory. This mapping of information is very efficient for the EXT2 file system but moderately less so for other file systems.

See `fs/inode.c`

As the system's processes access directories and files, system routines are called that traverse the VFS inodes in the system. For example, typing ls for a directory or cat for a file cause the the Virtual File System to search through the VFS inodes that represent the file system. As every file and directory on the system is represented by a VFS inode, then a number of inodes will be being repeatedly accessed. These inodes are kept in the inode cache which makes access to them quicker. If an inode is not in the inode cache, then a file system specific routine must be called in order to read the appropriate inode. The action of reading the inode causes it to be put into the inode cache and further accesses to the inode keep it in the cache. The less used VFS inodes get removed from the cache.

See `fs/buffer.c`

All of the Linux file systems use a common buffer cache to cache data buffers from the underlying devices to help speed up access by all of the file systems to the physical devices holding the file systems. This buffer cache is independent of the file systems and is integrated into the mechanisms that the Linux kernel uses to allocate and read and write data buffers. It has the distinct advantage of making the Linux file systems independent from the underlying media and from the device drivers that support them. All block structured devices register themselves with the Linux kernel and present a uniform, block based, usually asynchronous interface. Even relatively complex block devices such as SCSI devices do this. As the real file systems read data from the underlying physical disks, this results in requests to the block device drivers to read physical blocks from the device that they control. Integrated into this block device interface is the buffer cache. As blocks are read by the file systems they are saved in the global buffer cache shared by all of the file systems and the Linux kernel. Buffers within it are identified by their block number and a unique identifier for the device that read it. So, if the same data is needed often, it will be retrieved from the buffer cache rather than read from the disk, which would take somewhat longer. Some devices support read ahead where data blocks are speculatively read just in case they are needed.

See `fs/dcache.c`

The VFS also keeps a cache of directory lookups so that the inodes for frequently used directories can be quickly found. As an experiment, try listing a directory that you have not listed recently. The first time you list it, you may notice a slight pause but the second time you list its contents the result is immediate. The directory cache does not store the inodes for the directories itself; these should be in the inode cache, the directory cache simply stores the mapping between the full directory names and their inode numbers.

9.2.1 The VFS Superblock

Every mounted file system is represented by a VFS superblock; amongst other information, the VFS superblock contains the:

See `include/-linux/fs.h`

Device This is the device identifier for the block device that this file system is contained in. For example, `/dev/hda1`, the first IDE hard disk in the system has a device identifier of *0x301*,

Inode pointers The `mounted` inode pointer points at the first inode in this file system. The `covered` inode pointer points at the inode representing the directory that this file system is mounted on. The root file system's VFS superblock does not have a `covered` pointer,

Blocksize The block size in bytes of this file system, for example 1024 bytes,

Superblock operations A pointer to a set of superblock routines for this file system. Amongst other things, these routines are used by the VFS to read and write inodes and superblocks.

File System type A pointer to the mounted file system's `file_system_type` data structure,

File System specific A pointer to information needed by this file system,

9.2.2 The VFS Inode

Like the EXT2 file system, every file, directory and so on in the VFS is represented by one and only one VFS inode. The information in each VFS inode is built from information in the underlying file system by file system specific routines. VFS inodes exist only in the kernel's memory and are kept in the VFS inode cache as long as they are useful to the system. Amongst other information, VFS inodes contain the following fields:

See `include/-linux/fs.h`

device This is the device identifer of the device holding the file or whatever that this VFS inode represents,

inode number This is the number of the inode and is unique within this file system. The combination of `device` and `inode number` is unique within the Virtual File System,

mode Like EXT2 this field describes what this VFS inode represents as well as access rights to it,

user ids The owner identifiers,

times The creation, modification and write times,

block size The size of a block for this file in bytes, for example 1024 bytes,

inode operations A pointer to a block of routine addresses. These routines are specific to the file system and they perform operations for this inode, for example, truncate the file that is represented by this inode.

count The number of system components currently using this VFS inode. A count of zero means that the inode is free to be discarded or reused,

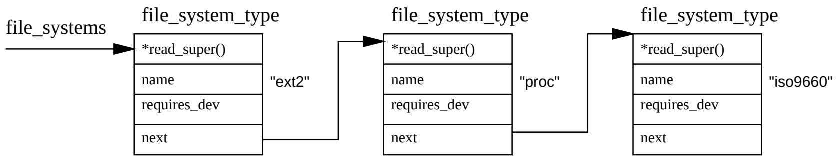

Figure 9.5: Registered File Systems

lock This field is used to lock the VFS inode, for example, when it is being read from the file system,

dirty Indicates whether this VFS inode has been written to, if so the underlying file system will need modifying,

file system specific information

9.2.3 Registering the File Systems

See `sys_setup()` in `fs/-filesystems.c`

When you build the Linux kernel you are asked if you want each of the supported file systems. When the kernel is built, the file system startup code contains calls to the initialisation routines of all of the built in file systems. Linux file systems may also be built as modules and, in this case, they may be demand loaded as they are needed or loaded by hand using insmod. Whenever a file system module is loaded it registers itself with the kernel and unregisters itself when it is unloaded. Each file system's initialisation routine registers itself with the Virtual File System and is represented by a `file_system_type` data structure which contains the name of the file system and a pointer to its VFS superblock read routine. Figure 9.5 shows that the `file_system_type` data structures are put into a list pointed at by the `file_systems` pointer. Each `file_system_type` data structure contains the following information:

See `file_system_type` in `include/-linux/fs.h`

Superblock read routine This routine is called by the VFS when an instance of the file system is mounted,

File System name The name of this file system, for example `ext2`,

Device needed Does this file system need a device to support? Not all file system need a device to hold them. The `/proc` file system, for example, does not require a block device,

You can see which file systems are registered by looking in at `/proc/filesystems`. For example:

```
      ext2
nodev proc
      iso9660
```

9.2.4 Mounting a File System

When the superuser attempts to mount a file system, the Linux kernel must first validate the arguments passed in the system call. Although mount does some basic checking, it does not know which file systems this kernel has been built to support or that the proposed mount point actually exists. Consider the following mount command:

```
$ mount -t iso9660 -o ro /dev/cdrom /mnt/cdrom
```

This mount command will pass the kernel three pieces of information; the name of the file system, the physical block device that contains the file system and, thirdly, where in the existing file system topology the new file system is to be mounted.

The first thing that the Virtual File System must do is to find the file system. To do this it searches through the list of known file systems by looking at each `file_system_type` data structure in the list pointed at by `file_systems`. If it finds a matching name it now knows that this file system type is supported by this kernel and it has the address of the file system specific routine for reading this file system's superblock. If it cannot find a matching file system name then all is not lost if the kernel is built to demand load kernel modules (see Chapter 12). In this case the kernel will request that the kernel daemon loads the appropriate file system module before continuing as before.

See `do_mount()` in `fs/super.c`

See `get_fs_type()` in `fs/super.c`

Next if the physical device passed by mount is not already mounted, it must find the VFS inode of the directory that is to be the new file system's mount point. This VFS inode may be in the inode cache or it might have to be read from the block device supporting the file system of the mount point. Once the inode has been found it is checked to see that it is a directory and that there is not already some other file system mounted there. The same directory cannot be used as a mount point for more than one file system.

At this point the VFS mount code must allocate a VFS superblock and pass it the mount information to the superblock read routine for this file system. All of the system's VFS superblocks are kept in the `super_blocks` vector of `super_block` data structures and one must be allocated for this mount. The superblock read routine must fill out the VFS superblock fields based on information that it reads from the physical device. For the EXT2 file system this mapping or translation of information is quite easy, it simply reads the EXT2 superblock and fills out the VFS superblock from there. For other file systems, such as the MS DOS file system, it is not quite such an easy task. Whatever the file system, filling out the VFS superblock means that the file system must read whatever describes it from the block device that supports it. If the block device cannot be read from or if it does not contain this type of file system then the mount command will fail.

Each mounted file system is described by a `vfsmount` data structure; see figure 9.6. These are queued on a list pointed at by `vfsmntlist`. Another pointer, `vfsmnttail` points at the last entry in the list and the `mru_vfsmnt` pointer points at the most recently used file system. Each `vfsmount` structure contains the device number of the block device holding the file system, the directory where this file system is mounted and a pointer to the VFS superblock allocated when this file system was mounted. In turn the VFS superblock points at the `file_system_type` data structure for this sort of file system and to the root inode for this file system. This inode is kept resident in the VFS inode cache all of the time that this file system is loaded.

See `add_vfsmnt()` in `fs/super.c`

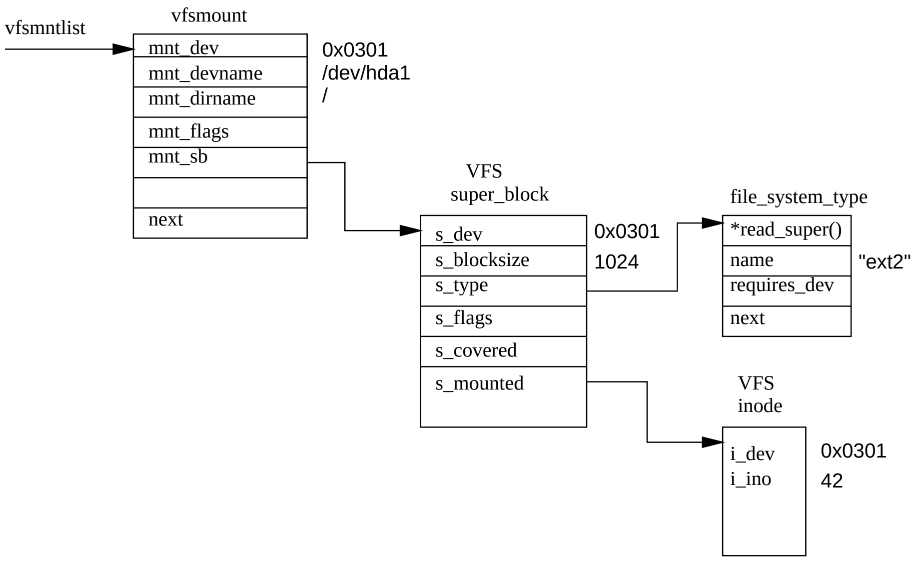

Figure 9.6: A Mounted File System

9.2.5 Finding a File in the Virtual File System

To find the VFS inode of a file in the Virtual File System, VFS must resolve the name a directory at a time, looking up the VFS inode representing each of the intermediate directories in the name. Each directory lookup involves calling the file system specific lookup whose address is held in the VFS inode representing the parent directory. This works because we always have the VFS inode of the root of each file system available and pointed at by the VFS superblock for that system. Each time an inode is looked up by the real file system it checks the directory cache for the directory. If there is no entry in the directory cache, the real file system gets the VFS inode either from the underlying file system or from the inode cache.

9.2.6 Creating a File in the Virtual File System

9.2.7 Unmounting a File System

See `do_umount()` in `fs/super.c`

The workshop manual for my MG usually describes assembly as the reverse of disassembly and the reverse is more or less true for unmounting a file system. A file system cannot be unmounted if something in the system is using one of its files. So, for example, you cannot umount `/mnt/cdrom` if a process is using that directory or any of its children. If anything is using the file system to be unmounted there may be VFS inodes from it in the VFS inode cache, and the code checks for this by looking through the list of inodes looking for inodes owned by the device that this file system occupies. If the VFS superblock for the mounted file system is dirty, that is it has been modified, then it must be written back to the file system on disk. Once it has been written to disk, the memory occupied by the VFS superblock is returned to the kernel's free pool of memory. Finally the `vfsmount` data structure for this mount is unlinked from `vfsmntlist` and freed.

See `remove_vfsmnt()` in `fs/super.c`

9.2.8 The VFS Inode Cache

As the mounted file systems are navigated, their VFS inodes are being continually read and, in some cases, written. The Virtual File System maintains an inode cache to speed up accesses to all of the mounted file systems. Every time a VFS inode is read from the inode cache the system saves an access to a physical device. See fs/inode.c

The VFS inode cache is implmented as a hash table whose entries are pointers to lists of VFS inodes that have the same hash value. The hash value of an inode is calculated from its inode number and from the device identifier for the underlying physical device containing the file system. Whenever the Virtual File System needs to access an inode, it first looks in the VFS inode cache. To find an inode in the cache, the system first calculates its hash value and then uses it as an index into the inode hash table. This gives it a pointer to a list of inodes with the same hash value. It then reads each inode in turn until it finds one with both the same inode number and the same device identifier as the one that it is searching for.

If it can find the inode in the cache, its count is incremented to show that it has another user and the file system access continues. Otherwise a free VFS inode must be found so that the file system can read the inode from memory. VFS has a number of choices about how to get a free inode. If the system may allocate more VFS inodes then this is what it does; it allocates kernel pages and breaks them up into new, free, inodes and puts them into the inode list. All of the system's VFS inodes are in a list pointed at by `first_inode` as well as in the inode hash table. If the system already has all of the inodes that it is allowed to have, it must find an inode that is a good candidate to be reused. Good candidates are inodes with a usage count of zero; this indicates that the system is not currently using them. Really important VFS inodes, for example the root inodes of file systems always have a usage count greater than zero and so are never candidates for reuse. Once a candidate for reuse has been located it is cleaned up. The VFS inode might be dirty and in this case it needs to be written back to the file system or it might be locked and in this case the system must wait for it to be unlocked before continuing. The candidate VFS inode must be cleaned up before it can be reused.

However the new VFS inode is found, a file system specific routine must be called to fill it out from information read from the underlying real file system. Whilst it is being filled out, the new VFS inode has a usage count of one and is locked so that nothing else accesses it until it contains valid information.

To get the VFS inode that is actually needed, the file system may need to access several other inodes. This happens when you read a directory; only the inode for the final directory is needed but the inodes for the intermediate directories must also be read. As the VFS inode cache is used and filled up, the less used inodes will be discarded and the more used inodes will remain in the cache.

9.2.9 The Directory Cache

To speed up accesses to commonly used directories, the VFS maintains a cache of directory entries. As directories are looked up by the real file systems their details are added into the directory cache. The next time the same directory is looked up, for example to list it or open a file within it, then it will be found in the directory cache. Only short directory entries (up to 15 characters long) are cached but this See fs/dcache.c

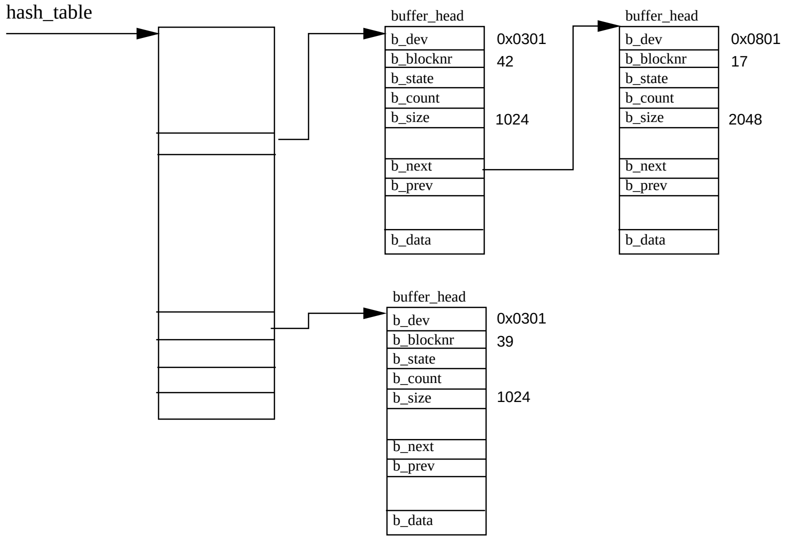

Figure 9.7: The Buffer Cache

is reasonable as the shorter directory names are the most commonly used ones. For example, `/usr/X11R6/bin` is very commonly accessed when the X server is running.

The directory cache consists of a hash table, each entry of which points at a list of directory cache entries that have the same hash value. The hash function uses the device number of the device holding the file system and the directory's name to calculate the offset, or index, into the hash table. It allows cached directory entries to be quickly found. It is no use having a cache when lookups within the cache take too long to find entries, or even not to find them.

In an effort to keep the caches valid and up to date the VFS keeps lists of Least Recently Used (LRU) directory cache entries. When a directory entry is first put into the cache, which is when it is first looked up, it is added onto the end of the first level LRU list. In a full cache this will displace an existing entry from the front of the LRU list. As the directory entry is accessed again it is promoted to the back of the second LRU cache list. Again, this may displace a cached level two directory entry at the front of the level two LRU cache list. This displacing of entries at the front of the level one and level two LRU lists is fine. The only reason that entries are at the front of the lists is that they have not been recently accessed. If they had, they would be nearer the back of the lists. The entries in the second level LRU cache list are safer than entries in the level one LRU cache list. This is the intention as these entries have not only been looked up but also they have been repeatedly referenced.

REVIEW NOTE: *Do we need a diagram for this?*

9.3 The Buffer Cache

As the mounted file systems are used they generate a lot of requests to the block devices to read and write data blocks. All block data read and write requests are given to the device drivers in the form of `buffer_head` data structures via standard kernel routine calls. These give all of the information that the block device drivers need; the device identifier uniquely identifies the device and the block number tells the driver which block to read. All block devices are viewed as linear collections of blocks of the same size. To speed up access to the physical block devices, Linux maintains a cache of block buffers. All of the block buffers in the system are kept somewhere in this buffer cache, even the new, unused buffers. This cache is shared between all of the physical block devices; at any one time there are many block buffers in the cache, belonging to any one of the system's block devices and often in many different states. If valid data is available from the buffer cache this saves the system an access to a physical device. Any block buffer that has been used to read data from a block device or to write data to it goes into the buffer cache. Over time it may be removed from the cache to make way for a more deserving buffer or it may remain in the cache as it is frequently accessed.

Block buffers within the cache are uniquely identfied by the owning device identifier and the block number of the buffer. The buffer cache is composed of two functional parts. The first part is the lists of free block buffers. There is one list per supported buffer size and the system's free block buffers are queued onto these lists when they are first created or when they have been discarded. The currently supported buffer sizes are 512, 1024, 2048, 4096 and 8192 bytes. The second functional part is the cache itself. This is a hash table which is a vector of pointers to chains of buffers that have the same hash index. The hash index is generated from the owning device identifier and the block number of the data block. Figure 9.7 shows the hash table together with a few entries. Block buffers are either in one of the free lists or they are in the buffer cache. When they are in the buffer cache they are also queued onto Least Recently Used (LRU) lists. There is an LRU list for each buffer type and these are used by the system to perform work on buffers of a type, for example, writing buffers with new data in them out to disk. The buffer's type reflects its state and Linux currently supports the following types:

clean Unused, new buffers,

locked Buffers that are locked, waiting to be written,

dirty Dirty buffers. These contain new, valid data, and will be written but so far have not been scheduled to write,

shared Shared buffers,

unshared Buffers that were once shared but which are now not shared,

Whenever a file system needs to read a buffer from its underlying physical device, it trys to get a block from the buffer cache. If it cannot get a buffer from the buffer cache, then it will get a clean one from the appropriate sized free list and this new buffer will go into the buffer cache. If the buffer that it needed is in the buffer cache, then it may or may not be up to date. If it is not up to date or if it is a new block buffer, the file system must request that the device driver read the appropriate block of data from the disk.

Like all caches, the buffer cache must be maintained so that it runs efficiently and fairly allocates cache entries between the block devices using the buffer cache. Linux uses the `bdflush` kernel daemon to perform a lot of housekeeping duties on the cache but some happen automatically as a result of the cache being used.

9.3.1 The bdflush Kernel Daemon

See bdflush() in fs/buffer.c

The bdflush kernel daemon is a simple kernel daemon that provides a dynamic response to the system having too many dirty buffers; buffers that contain data that must be written out to disk at some time. It is started as a kernel thread at system startup time and, rather confusingly, it calls itself "kflushd" and that is the name that you will see if you use the ps command to show the processes in the system. Mostly this daemon sleeps waiting for the number of dirty buffers in the system to grow too large. As buffers are allocated and discarded the number of dirty buffers in the system is checked. If there are too many as a percentage of the total number of buffers in the system then bdflush is woken up. The default threshold is 60% but, if the system is desperate for buffers, bdflush will be woken up anyway. This value can be seen and changed using the update command:

```
# update -d

bdflush version 1.4
0:     60 Max fraction of LRU list to examine for dirty blocks
1:    500 Max number of dirty blocks to write each time bdflush activated
2:     64 Num of clean buffers to be loaded onto free list by refill_freelist
3:    256 Dirty block threshold for activating bdflush in refill_freelist
4:     15 Percentage of cache to scan for free clusters
5:   3000 Time for data buffers to age before flushing
6:    500 Time for non-data (dir, bitmap, etc) buffers to age before flushing
7:   1884 Time buffer cache load average constant
8:      2 LAV ratio (used to determine threshold for buffer fratricide).
```

All of the dirty buffers are linked into the BUF_DIRTY LRU list whenever they are made dirty by having data written to them and bdflush tries to write a reasonable number of them out to their owning disks. Again this number can be seen and controlled by the update command and the default is 500 (see above).

9.3.2 The update Process

See sys_bdflush() in fs/buffer.c

The update command is more than just a command; it is also a daemon. When run as superuser (during system initialisation) it will periodically flush all of the older dirty buffers out to disk. It does this by calling a system service routine that does more or less the same thing as bdflush. Whenever a dirty buffer is finished with, it is tagged with the system time that it should be written out to its owning disk. Every time that update runs it looks at all of the dirty buffers in the system looking for ones with an expired flush time. Every expired buffer is written out to disk.

9.4 The /proc File System

The /proc file system really shows the power of the Linux Virtual File System. It does not really exist (yet another of Linux's conjuring tricks), neither the /proc directory nor its subdirectories and its files actually exist. So how can you cat

`/proc/devices`? The /proc file system, like a real file system, registers itself with the Virtual File System. However, when the VFS makes calls to it requesting inodes as its files and directories are opened, the `/proc` file system creates those files and directories from information within the kernel. For example, the kernel's `/proc/devices` file is generated from the kernel's data structures describing its devices.

The `/proc` file system presents a user readable window into the kernel's inner workings. Several Linux subsystems, such as Linux kernel modules described in chapter 12, create entries in the the `/proc` file system.

9.5 Device Special Files

Linux, like all versions of Unix™ presents its hardware devices as special files. So, for example, /dev/null is the null device. A device file does not use any data space in the file system, it is only an access point to the device driver. The EXT2 file system and the Linux VFS both implement device files as special types of inode. There are two types of device file; character and block special files. Within the kernel itself, the device drivers implement file semantices: you can open them, close them and so on. Character devices allow I/O operations in character mode and block devices require that all I/O is via the buffer cache. When an I/O request is made to a device file, it is forwarded to the appropriate device driver within the system. Often this is not a real device driver but a pseudo-device driver for some subsystem such as the SCSI device driver layer. Device files are referenced by a major number, which identifies the device type, and a minor type, which identifies the unit, or instance of that major type. For example, the IDE disks on the first IDE controller in the system have a major number of 3 and the first partition of an IDE disk would have a minor number of 1. So, `ls -l` of `/dev/hda1` gives:

see `/include/linux/major.h` for all of Linux's major device numbers.

```
$ brw-rw----   1 root    disk       3,    1  Nov 24  15:09 /dev/hda1
```

Within the kernel, every device is uniquely described by a `kdev_t` data type, this is two bytes long, the first byte containing the minor device number and the second byte holding the major device number. The IDE device above is held within the kernel as *0x0301*. An EXT2 inode that represents a block or character device keeps the device's major and minor numbers in its first direct block pointer. When it is read by the VFS, the VFS inode data structure representing it has its `i_rdev` field set to the correct device identifier.

See `include/linux/kdev_t.h`

Chapter 10

Networks

Networking and Linux are terms that are almost synonymous. In a very real sense Linux is a product of the Internet or World Wide Web (WWW). Its developers and users use the web to exchange information ideas, code, and Linux itself is often used to support the networking needs of organizations. This chapter describes how Linux supports the network protocols known collectively as TCP/IP.

The TCP/IP protocols were designed to support communications between computers connected to the ARPANET, an American research network funded by the US government. The ARPANET pioneered networking concepts such as packet switching and protocol layering where one protocol uses the services of another. ARPANET was retired in 1988 but its successors (NSF[1] NET and the Internet) have grown even larger. What is now known as the World Wide Web grew from the ARPANET and is itself supported by the TCP/IP protocols. Unix™ was extensively used on the ARPANET and the first released networking version of Unix™ was 4.3 BSD. Linux's networking implementation is modeled on 4.3 BSD in that it supports BSD sockets (with some extensions) and the full range of TCP/IP networking. This programming interface was chosen because of its popularity and to help applications be portable between Linux and other Unix™ platforms.

10.1 An Overview of TCP/IP Networking

This section gives an overview of the main principles of TCP/IP networking. It is not meant to be an exhaustive description, for that I suggest that you read [10, Comer].

In an IP network every machine is assigned an IP address, this is a 32 bit number that uniquely identifies the machine. The WWW is a very large, and growing, IP network and every machine that is connected to it has to have a unique IP address assigned to it. IP addresses are represented by four numbers separated by dots, for

[1] National Science Foundation

example, `16.42.0.9`. This IP address is actually in two parts, the *network* address and the *host* address. The sizes of these parts may vary (there are several classes of IP addresses) but using `16.42.0.9` as an example, the network address would be `16.42` and the host address `0.9`. The host address is further subdivided into a *subnetwork* and a *host* address. Again, using `16.42.0.9` as an example, the subnetwork address would be `16.42.0` and the host address 16.42.0.9. This subdivision of the IP address allows organizations to subdivide their networks. For example, `16.42` could be the network address of the ACME Computer Company; `16.42.0` would be subnet `0` and `16.42.1` would be subnet `1`. These subnets might be in separate buildings, perhaps connected by leased telephone lines or even microwave links. IP addresses are assigned by the network administrator and having IP subnetworks is a good way of distributing the administration of the network. IP subnet administrators are free to allocate IP addresses within their IP subnetworks.

Generally though, IP addresses are somewhat hard to remember. Names are much easier. `linux.acme.com` is much easier to remember than `16.42.0.9` but there must be some mechanism to convert the network names into an IP address. These names can be statically specified in the `/etc/hosts` file or Linux can ask a Distributed Name Server (DNS server) to resolve the name for it. In this case the local host must know the IP address of one or more DNS servers and these are specified in `/etc/resolv.conf`.

Whenever you connect to another machine, say when reading a web page, its IP address is used to exchange data with that machine. This data is contained in IP packets each of which have an IP header containing the IP addresses of the source and destination machine's IP addresses, a checksum and other useful information. The checksum is derived from the data in the IP packet and allows the receiver of IP packets to tell if the IP packet was corrupted during transmission, perhaps by a noisy telephone line. The data transmitted by an application may have been broken down into smaller packets which are easier to handle. The size of the IP data packets varies depending on the connection media; ethernet packets are generally bigger than PPP packets. The destination host must reassemble the data packets before giving the data to the receiving application. You can see this fragmentation and reassembly of data graphically if you access a web page containing a lot of graphical images via a moderately slow serial link.

Hosts connected to the same IP subnet can send IP packets directly to each other, all other IP packets will be sent to a special host, a gateway. Gateways (or routers) are connected to more than one IP subnet and they will resend IP packets received on one subnet, but destined for another onwards. For example, if subnets `16.42.1.0` and `16.42.0.0` are connected together by a gateway then any packets sent from subnet `0` to subnet `1` would have to be directed to the gateway so that it could route them. The local host builds up routing tables which allow it to route IP packets to the correct machine. For every IP destination there is an entry in the routing tables which tells Linux which host to send IP packets to in order that they reach their destination. These routing tables are dynamic and change over time as applications use the network and as the network topology changes.

The IP protocol is a transport layer that is used by other protocols to carry their data. The Transmission Control Protocol (TCP) is a reliable end to end protocol that uses IP to transmit and receive its own packets. Just as IP packets have their own header, TCP has its own header. TCP is a connection based protocol where two networking

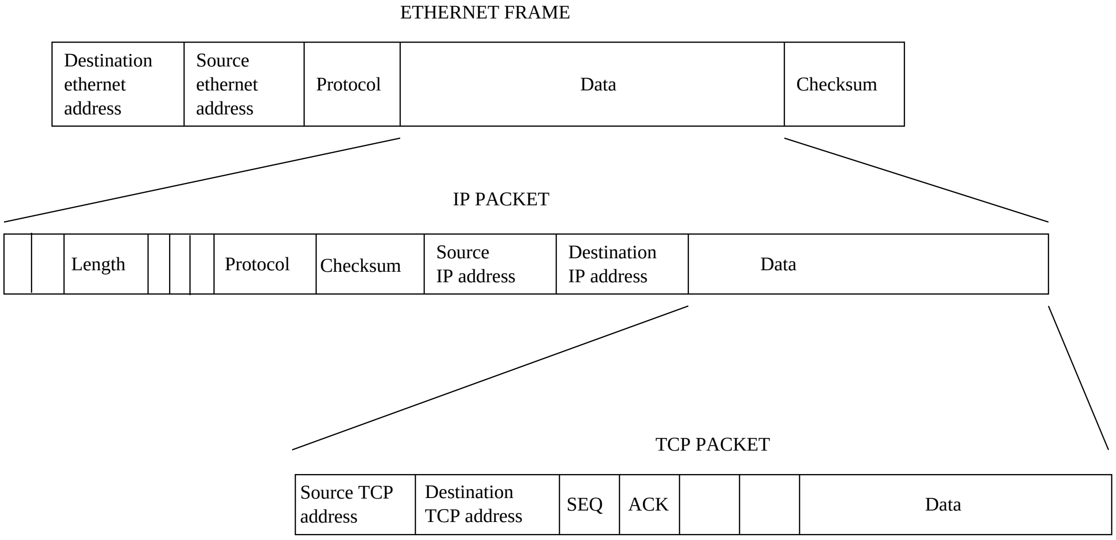

Figure 10.1: TCP/IP Protocol Layers

applications are connected by a single, virtual connection even though there may be many subnetworks, gateways and routers between them. TCP reliably transmits and receives data between the two applications and guarantees that there will be no lost or duplicated data. When TCP transmits its packet using IP, the data contained within the IP packet is the TCP packet itself. The IP layer on each communicating host is responsible for transmitting and receiving IP packets. User Datagram Protocol (UDP) also uses the IP layer to transport its packets, unlike TCP, UDP is not a reliable protocol but offers a datagram service. This use of IP by other protocols means that when IP packets are received the receiving IP layer must know which upper protocol layer to give the data contained in this IP packet to. To facilitate this every IP packet header has a byte containing a protocol identifier. When TCP asks the IP layer to transmit an IP packet , that IP packet's header states that it contains a TCP packet. The receiving IP layer uses that protocol identifier to decide which layer to pass the received data up to, in this case the TCP layer. When applications communicate via TCP/IP they must specify not only the target's IP address but also the *port* address of the application. A port address uniquely identifies an application and standard network applications use standard port addresses; for example, web servers use port 80. These registered port addresses can be seen in `/etc/services`.

This layering of protocols does not stop with TCP, UDP and IP. The IP protocol layer itself uses many different physical media to transport IP packets to other IP hosts. These media may themselves add their own protocol headers. One such example is the ethernet layer, but PPP and SLIP are others. An ethernet network allows many hosts to be simultaneously connected to a single physical cable. Every transmitted ethernet frame can be seen by all connected hosts and so every ethernet device has a unique address. Any ethernet frame transmitted to that address will be received by the addressed host but ignored by all the other hosts connected to the network. These unique addresses are built into each ethernet device when they are manufactured and it is usually kept in an SROM[2] on the ethernet card. Ethernet addresses are 6 bytes long, an example would be `08-00-2b-00-49-A4`. Some ethernet

[2]Synchronous Read Only Memory

addresses are reserved for multicast purposes and ethernet frames sent with these destination addresses will be received by all hosts on the network. As ethernet frames can carry many different protocols (as data) they, like IP packets, contain a protocol identifier in their headers. This allows the ethernet layer to correctly receive IP packets and to pass them onto the IP layer.

In order to send an IP packet via a multi-connection protocol such as ethernet, the IP layer must find the ethernet address of the IP host. This is because IP addresses are simply an addressing concept, the ethernet devices themselves have their own physical addresses. IP addresses on the other hand can be assigned and reassigned by network administrators at will but the network hardware responds only to ethernet frames with its own physical address or to special multicast addresses which all machines must receive. Linux uses the Address Resolution Protocol (or ARP) to allow machines to translate IP addresses into real hardware addresses such as ethernet addresses. A host wishing to know the hardware address associated with an IP address sends an ARP request packet containing the IP address that it wishes translating to all nodes on the network by sending it to a multicast address. The target host that owns the IP address, responds with an ARP reply that contains its physical hardware address. ARP is not just restricted to ethernet devices, it can resolve IP addresses for other physical media, for example FDDI. Those network devices that cannot ARP are marked so that Linux does not attempt to ARP. There is also the reverse function, Reverse ARP or RARP, which translates phsyical network addresses into IP addresses. This is used by gateways, which respond to ARP requests on behalf of IP addresses that are in the remote network.

10.2 The Linux TCP/IP Networking Layers

Just like the network protocols themselves, Figure 10.2 shows that Linux implements the internet protocol address family as a series of connected layers of software. BSD sockets are supported by a generic socket management software concerned only with BSD sockets. Supporting this is the INET socket layer, this manages the communication end points for the IP based protocols TCP and UDP. UDP (User Datagram Protocol) is a connectionless protocol whereas TCP (Transmission Control Protocol) is a reliable end to end protocol. When UDP packets are transmitted, Linux neither knows nor cares if they arrive safely at their destination. TCP packets are numbered and both ends of the TCP connection make sure that transmitted data is received correctly. The IP layer contains code implementing the Internet Protocol. This code prepends IP headers to transmitted data and understands how to route incoming IP packets to either the TCP or UDP layers. Underneath the IP layer, supporting all of Linux's networking are the network devices, for example PPP and ethernet. Network devices do not always represent physical devices; some like the loopback device are purely software devices. Unlike standard Linux devices that are created via the `mknod` command, network devices appear only if the underlying software has found and initialized them. You will only see `/dev/eth0` when you have built a kernel with the appropriate ethernet device driver in it. The ARP protocol sits between the IP layer and the protocols that support ARPing for addresses.

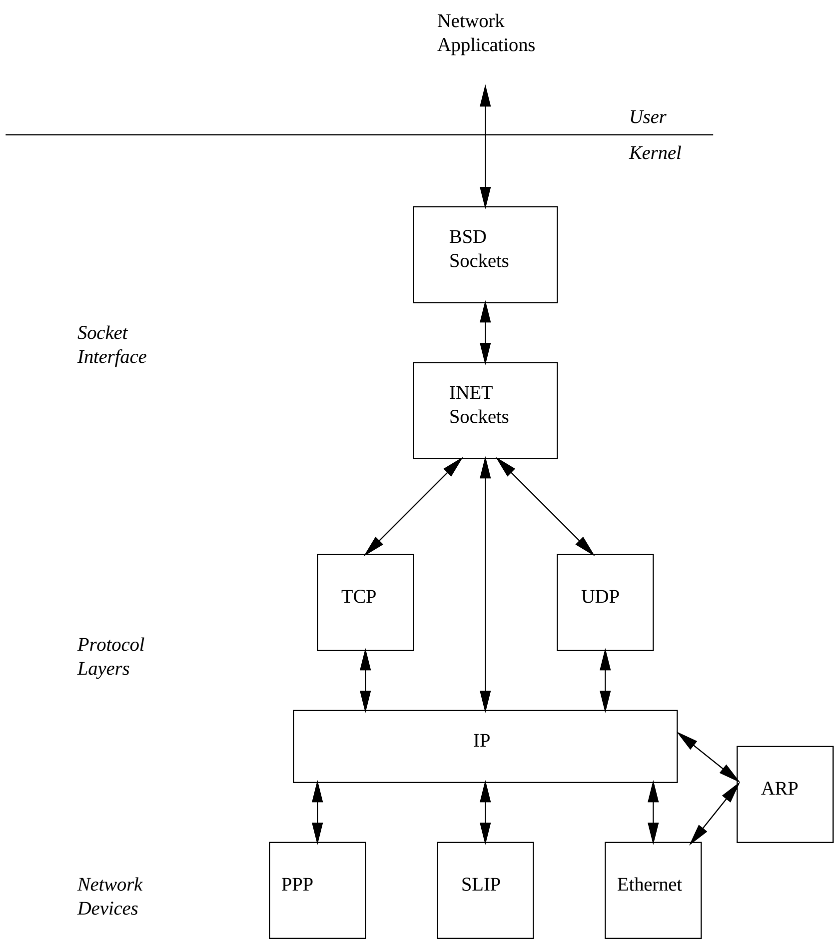

Figure 10.2: Linux Networking Layers

10.3 The BSD Socket Interface

This is a general interface which not only supports various forms of networking but is also an inter-process communications mechanism. A socket describes one end of a communications link, two communicating processes would each have a socket describing their end of the communication link between them. Sockets could be thought of as a special case of pipes but, unlike pipes, sockets have no limit on the amount of data that they can contain. Linux supports several classes of socket and these are known as *address families*. This is because each class has its own method of addressing its communications. Linux supports the following socket address families or domains:

UNIX	Unix domain sockets,
INET	The Internet address family supports communications via TCP/IP protocols
AX25	Amateur radio X25
IPX	Novell IPX
APPLETALK	Appletalk DDP
X25	X25

There are several socket types and these represent the type of service that supports the connection. Not all address families support all types of service. Linux BSD sockets support a number of socket types:

Stream These sockets provide reliable two way sequenced data streams with a guarantee that data cannot be lost, corrupted or duplicated in transit. Stream sockets are supported by the TCP protocol of the Internet (INET) address family.

Datagram These sockets also provide two way data transfer but, unlike stream sockets, there is no guarantee that the messages will arrive. Even if they do arrive there is no guarantee that they will arrive in order or even not be duplicated or corrupted. This type of socket is supported by the UDP protocol of the Internet address family.

Raw This allows processes direct (hence "raw") access to the underlying protocols. It is, for example, possible to open a raw socket to an ethernet device and see raw IP data traffic.

Reliable Delivered Messages These are very like datagram sockets but the data is guaranteed to arrive.

Sequenced Packets These are like stream sockets except that the data packet sizes are fixed.

Packet This is not a standard BSD socket type, it is a Linux specific extension that allows processes to access packets directly at the device level.

Processes that communicate using sockets use a client server model. A server provides a service and clients make use of that service. One example would be a Web Server, which provides web pages and a web client, or browser, which reads those pages. A server using sockets, first creates a socket and then binds a name to it. The format of this name is dependent on the socket's address family and it is, in effect, the local address of the server. The socket's name or address is specified using the `sockaddr` data structure. An INET socket would have an IP port address bound to it. The registered port numbers can be seen in /etc/services; for example, the port number for a web server is 80. Having bound an address to the socket, the server then listens for incoming connection requests specifying the bound address. The originator of the request, the client, creates a socket and makes a connection request on it, specifying the target address of the server. For an INET socket the address of the server is its IP address and its port number. These incoming requests must find their way up through the various protocol layers and then wait on the server's listening socket. Once the server has received the incoming request it either accepts or rejects it. If the incoming request is to be accepted, the server must create a new socket to accept it on. Once a socket has been used for listening for incoming connection requests it cannot be used to support a connection. With the connection established both ends are free to send and receive data. Finally, when the connection is no longer needed it can be shutdown. Care is taken to ensure that data packets in transit are correctly dealt with.

The exact meaning of operations on a BSD socket depends on its underlying address family. Setting up TCP/IP connections is very different from setting up an amateur radio X.25 connection. Like the virtual filesystem, Linux abstracts the socket interface with the BSD socket layer being concerned with the BSD socket interface to the application programs which is in turn supported by independent address family specific software. At kernel initialization time, the address families built into the kernel register themselves with the BSD socket interface. Later on, as applications create and use BSD sockets, an association is made between the BSD socket and its supporting address family. This association is made via cross-linking data structures and tables of address family specific support routines. For example there is an address family specific socket creation routine which the BSD socket interface uses when an application creates a new socket.

When the kernel is configured, a number of address families and protocols are built into the `protocols` vector. Each is represented by its name, for example "INET" and the address of its initialization routine. When the socket interface is initialized at boot time each protocol's initialization routine is called. For the socket address families this results in them registering a set of protocol operations. This is a set of routines, each of which performs a a particular operation specific to that address family. The registered protocol operations are kept in the `pops` vector, a vector of pointers to `proto_ops` data structures. The `proto_ops` data structure consists of the address family type and a set of pointers to socket operation routines specific to a particular address family. The `pops` vector is indexed by the address family identifier, for example the Internet address family identifier (AF_INET is 2).

See `include/-linux/net.h`

10.4 The INET Socket Layer

The INET socket layer supports the internet address family which contains the TCP/IP protocols. As discussed above, these protocols are layered, one protocol using the services of another. Linux's TCP/IP code and data structures reflect this layering. Its interface with the BSD socket layer is through the set of Internet address family socket operations which it registers with the BSD socket layer during network initialization. These are kept in the `pops` vector along with the other registered address families. The BSD socket layer calls the INET layer socket support

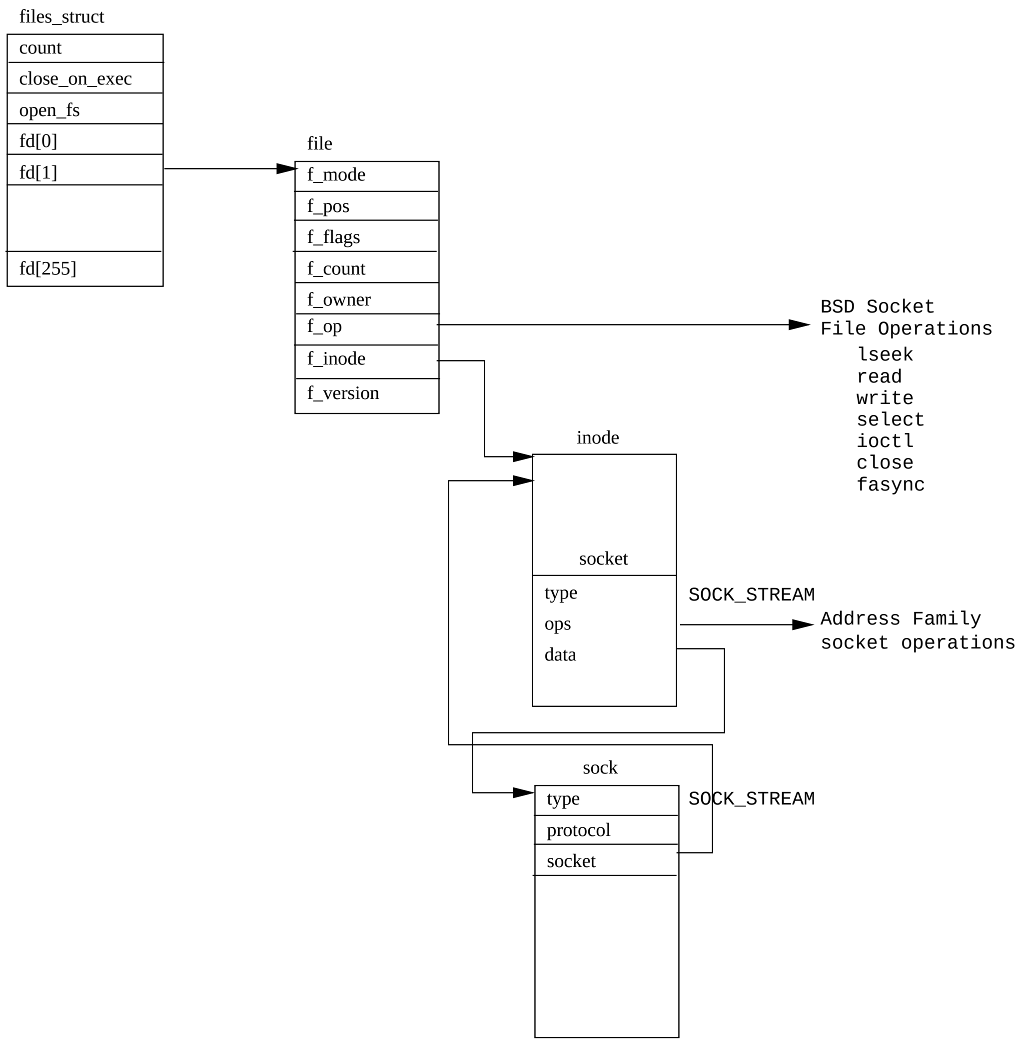

Figure 10.3: Linux BSD Socket Data Structures

routines from the registered INET `proto_ops` data structure to perform work for it. For example a BSD socket create request that gives the address family as INET will use the underlying INET socket create function. The BSD socket layer passes the `socket` data structure representing the BSD socket to the INET layer in each of these operations. Rather than clutter the BSD `socket` wiht TCP/IP specific information, the INET socket layer uses its own data structure, the `sock` which it links to the BSD `socket` data structure. This linkage can be seen in Figure 10.3. It links the `sock` data structure to the BSD `socket` data structure using the `data` pointer in the BSD `socket`. This means that subsequent INET socket calls can easily retrieve the `sock` data structure. The `sock` data structure's protocol operations pointer is also set up at creation time and it depends on the protocol requested. If TCP is requested, then the `sock` data structure's protocol operations pointer will point to the set of TCP protocol operations needed for a TCP connection.

See `include/-net/sock.h`

10.4.1 Creating a BSD Socket

The system call to create a new socket passes identifiers for its address family, socket type and protocol. Firstly the requested address family is used to search the `pops` vector for a matching address family. It may be that a particular address family is implemented as a kernel module and, in this case, the `kerneld` daemon must load the module before we can continue. A new `socket` data structure is allocated to represent the BSD socket. Actually the `socket` data structure is physically part of the VFS `inode` data structure and allocating a socket really means allocating a VFS `inode`. This may seem strange unless you consider that sockets can be operated on in just the same way that ordinairy files can. As all files are represented by a VFS `inode` data structure, then in order to support file operations, BSD sockets must also be represented by a VFS `inode` data structure.

See `sys_socket()` in `net/socket.c`

The newly created BSD `socket` data structure contains a pointer to the address family specific socket routines and this is set to the `proto_ops` data structure retrieved from the `pops` vector. Its type is set to the sccket type requested; one of SOCK_STREAM, SOCK_DGRAM and so on. The address family specific creation routine is called using the address kept in the `proto_ops` data structure.

A free file descriptor is allocated from the current processes `fd` vector and the `file` data structure that it points at is initialized. This includes setting the file operations pointer to point to the set of BSD socket file operations supported by the BSD socket interface. Any future operations will be directed to the socket interface and it will in turn pass them to the supporting address family by calling its address family operation routines.

10.4.2 Binding an Address to an INET BSD Socket

In order to be able to listen for incoming internet connection requests, each server must create an INET BSD socket and bind its address to it. The bind operation is mostly handled within the INET socket layer with some support from the underlying TCP and UDP protocol layers. The socket having an address bound to cannot be being used for any other communication. This means that the `socket`'s state must be `TCP_CLOSE`. The `sockaddr` pass to the bind operation contains the IP address to be bound to and, optionally, a port number. Normally the IP address bound to would be one that has been assigned to a network device that supports the INET address family and whose interface is up and able to be used. You can see which network interfaces are currently active in the system by using the ifconfig command. The IP address may also be the IP broadcast address of either all 1's or all 0's. These are special addresses that mean "send to everybody"[3]. The IP address could also be specified as any IP address if the machine is acting as a transparent proxy or firewall, but only processes with superuser privileges can bind to any IP address. The IP address bound to is saved in the `sock` data structure in the `recv_addr` and `saddr` fields. These are used in hash lookups and as the sending IP address respectively. The port number is optional and if it is not specified the supporting network is asked for a free one. By convention, port numbers less than 1024 cannot be used by processes without superuser privileges. If the underlying network does allocate a port number it always allocates ones greater than 1024.

As packets are being received by the underlying network devices they must be routed to the correct INET and BSD sockets so that they can be processed. For this reason UDP and TCP maintain hash tables which are used to lookup the addresses within incoming IP messages and direct them to the correct `socket`/`sock` pair. TCP is a connection oriented protocol and so there is more information involved in processing TCP packets than there is in processing UDP packets.

UDP maintains a hash table of allocated UDP ports, the `udp_hash` table. This consists of pointers to `sock` data structures indexed by a hash function based on the port number. As the UDP hash table is much smaller than the number of permissible port numbers (`udp_hash` is only 128 or `UDP_HTABLE_SIZE` entries long) some entries in the table point to a chain of `sock` data structures linked together using each `sock`'s `next` pointer.

TCP is much more complex as it maintains several hash tables. However, TCP does not actually add the binding `sock` data stucture into its hash tables during the bind operation, it merely checks that the port number requested is not currently being used. The `sock` data structure is added to TCP's hash tables during the *listen* operation.

REVIEW NOTE: *What about the route entered?*

10.4.3 Making a Connection on an INET BSD Socket

Once a socket has been created and, provided it has not been used to listen for inbound connection requests, it can be used to make outbound connection requests. For connectionless protocols like UDP this socket operation does not do a whole lot but for connection orientated protocols like TCP it involves building a virtual circuit between two applications.

An outbound connection can only be made on an INET BSD socket that is in the right state; that is to say one that does not already have a connection established and one that is not being used for listening for inbound connections. This means that the BSD `socket` data structure must be in state `SS_UNCONNECTED`. The UDP protocol does not establish virtual connections between applications, any messages sent are datagrams, one off messages that may or may not reach their destinations. It does, however, support the *connect* BSD socket operation. A connection operation on a UDP INET BSD socket simply sets up the addresses of the remote application; its IP address and its IP port number. Additionally it sets up a cache of the routing table

[3] duh? What used for?

entry so that UDP packets sent on this BSD socket do not need to check the routing database again (unless this route becomes invalid). The cached routing information is pointed at from the `ip_route_cache` pointer in the INET `sock` data structure. If no addressing information is given, this cached routing and IP addressing information will be automatically be used for messages sent using this BSD socket. UDP moves the `sock`'s state to `TCP_ESTABLISHED`.

For a connect operation on a TCP BSD socket, TCP must build a TCP message containing the connection information and send it to IP destination given. The TCP message contains information about the connection, a unique starting message sequence number, the maximum sized message that can be managed by the initiating host, the transmit and receive window size and so on. Within TCP all messages are numbered and the initial sequence number is used as the first message number. Linux chooses a reasonably random value to avoid malicious protocol attacks. Every message transmitted by one end of the TCP connection and successfully received by the other is acknowledged to say that it arrived successfully and uncorrupted. Unacknowledges messages will be retransmitted. The transmit and receive window size is the number of outstanding messages that there can be without an acknowledgement being sent. The maximum message size is based on the network device that is being used at the initiating end of the request. If the receiving end's network device supports smaller maximum message sizes then the connection will use the minimum of the two. The application making the outbound TCP connection request must now wait for a response from the target application to accept or reject the connection request. As the TCP `sock` is now expecting incoming messages, it is added to the `tcp_listening_hash` so that incoming TCP messages can be directed to this `sock` data structure. TCP also starts timers so that the outbound connection request can be timed out if the target application does not respond to the request.

10.4.4 Listening on an INET BSD Socket

Once a socket has had an address bound to it, it may listen for incoming connection requests specifying the bound addresses. A network application can listen on a socket without first binding an address to it; in this case the INET socket layer finds an unused port number (for this protocol) and automatically binds it to the socket. The listen socket function moves the socket into state `TCP_LISTEN` and does any network specific work needed to allow incoming connections.

For UDP sockets, changing the socket's state is enough but TCP now adds the socket's `sock` data structure into two hash tables as it is now active. These are the `tcp_bound_hash` table and the `tcp_listening_hash`. Both are indexed via a hash function based on the IP port number.

Whenever an incoming TCP connection request is received for an active listening socket, TCP builds a new `sock` data structure to represent it. This `sock` data structure will become the bottom half of the TCP connection when it is eventually accepted. It also clones the incoming `sk_buff` containing the connection request and queues it onto the `receive_queue` for the listening `sock` data structure. The clone `sk_buff` contains a pointer to the newly created `sock` data structure.

10.4.5 Accepting Connection Requests

UDP does not support the concept of connections, accepting INET socket connection requests only applies to the TCP protocol as an accept operation on a listening socket causes a new `socket` data structure to be cloned from the original listening `socket`. The accept operation is then passed to the supporting protocol layer, in this case INET to accept any incoming connection requests. The INET protocol layer will fail the accept operation if the underlying protocol, say UDP, does not support connections. Otherwise the accept operation is passed through to the real protocol, in this case TCP. The accept operation can be either blocking or non-blocking. In the non-blocking case if there are no incoming connections to accept, the accept operation will fail and the newly created `socket` data structure will be thrown away. In the blocking case the network application performing the accept operation will be added to a wait queue and then suspended until a TCP connection request is received. Once a connection request has been received the `sk_buff` containing the request is discarded and the `sock` data structure is returned to the INET socket layer where it is linked to the new `socket` data structure created earlier. The file descriptor (`fd`) number of the new `socket` is returned to the network application, and the application can then use that file descriptor in socket operations on the newly created INET BSD socket.

10.5 The IP Layer

10.5.1 Socket Buffers

One of the problems of having many layers of network protocols, each one using the services of another, is that each protocol needs to add protocol headers and tails to data as it is transmitted and to remove them as it processes received data. This make passing data buffers between the protocols difficult as each layer needs to find where its particular protocol headers and tails are. One solution is to copy buffers at each layer but that would be inefficient. Instead, Linux uses socket buffers or `sk_buffs` to pass data between the protocol layers and the network device drivers. `sk_buffs` contain pointer and length fields that allow each protocol layer to manipulate the application data via standard functions or "methods".

See `include/-linux/skbuff.h`

Figure 10.4 shows the `sk_buff` data structure; each `sk_buff` has a block of data associated with it. The `sk_buff` has four data pointers, which are used to manipulate and manage the socket buffer's data:

head points to the start of the data area in memory. This is fixed when the `sk_buff` and its associated data block is allocated,

data points at the current start of the protocol data. This pointer varies depending on the protocol layer that currently owns the `sk_buff`,

tail points at the current end of the protocol data. Again, this pointer varies depending on the owning protocol layer,

end points at the end of the data area in memory. This is fixed when the `sk_buff` is allocated.

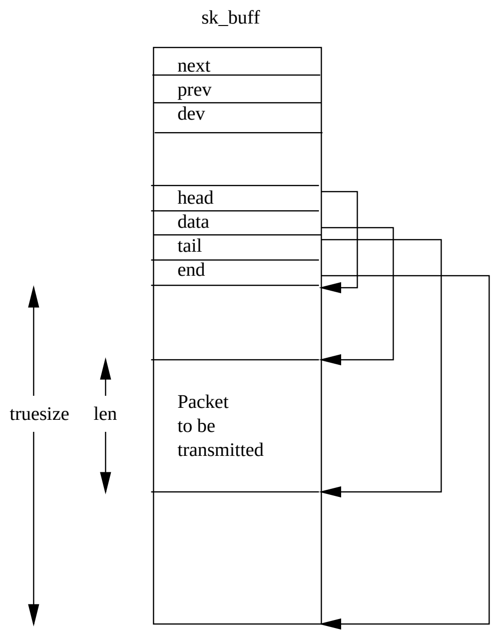

Figure 10.4: The Socket Buffer (sk_buff)

There are two length fields `len` and `truesize`, which describe the length of the current protocol packet and the total size of the data buffer respectively. The `sk_buff` handling code provides standard mechanisms for adding and removing protocol headers and tails to the application data. These safely manipulate the `data`, `tail` and `len` fields in the `sk_buff`:

push This moves the `data` pointer towards the start of the data area and increments the `len` field. This is used when adding data or protocol headers to the start of the data to be transmitted,

See `skb_push()` in `include/-linux/skbuff.h`

pull This moves the `data` pointer away from the start, towards the end of the data area and decrements the `len` field. This is used when removing data or protocol headers from the start of the data that has been received,

See `skb_pull()` in `include/-linux/skbuff.h`

put This moves the `tail` pointer towards the end of the data area and increments the `len` field. This is used when adding data or protocol information to the end of the data to be transmitted,

See `skb_put()` in `include/linux/-skbuff.h`

trim This moves the `tail` pointer towards the start of the data area and decrements the `len` field. This is used when removing data or protocol tails from the received packet.

See `skb_trim()` in `include/-linux/skbuff.h`

The `sk_buff` data structure also contains pointers that are used as it is stored in doubly linked circular lists of `sk_buff`'s during processing. There are generic `sk_buff` routines for adding `sk_buffs` to the front and back of these lists and for removing them.

10.5.2 Receiving IP Packets

Chapter 8 described how Linux's network drivers built are into the kernel and initialized. This results in a series of `device` data structures linked together in the dev_base list. Each device data structure describes its device and provides a set of callback routines that the network protocol layers call when they need the network driver to perform work. These functions are mostly concerned with transmitting data and with the network device's addresses. When a network device receives packets from its network it must convert the received data into sk_buff data structures. These received sk_buff's are added onto the backlog queue by the network drivers as they are received. If the backlog queue grows too large, then the received sk_buff's are discarded. The network bottom half is flagged as ready to run as there is work to do.

See netif_rx() in net/core/dev.c

When the network bottom half handler is run by the scheduler it processes any network packets waiting to be transmitted before processing the backlog queue of sk_buff's determining which protocol layer to pass the received packets to. As the Linux networking layers were initialized, each protocol registered itself by adding a packet_type data structure onto either the ptype_all list or into the ptype_base hash table. The packet_type data structure contains the protocol type, a pointer to a network device, a pointer to the protocol's receive data processing routine and, finally, a pointer to the next packet_type data structure in the list or hash chain. The ptype_all chain is used to snoop all packets being received from any network device and is not normally used. The ptype_base hash table is hashed by protocol identifier and is used to decide which protocol should receive the incoming network packet. The network bottom half matches the protocol types of incoming sk_buff's against one or more of the packet_type entries in either table. The protocol may match more than one entry, for example when snooping all network traffic, and in this case the sk_buff will be cloned. The sk_buff is passed to the matching protocol's handling routine.

See net_bh() in net/core/dev.c

See ip_recv() in net/ipv4/-ip_input.c

10.5.3 Sending IP Packets

Packets are transmitted by applications exchanging data or else they are generated by the network protocols as they support established connections or connections being established. Whichever way the data is generated, an sk_buff is built to contain the data and various headers are added by the protocol layers as it passes through them.

The sk_buff needs to be passed to a network device to be transmitted. First though the protocol, for example IP, needs to decide which network device to use. This depends on the best route for the packet. For computers connected by modem to a single network, say via the PPP protocol, the routing choice is easy. The packet should either be sent to the local host via the loopback device or to the gateway at the end of the PPP modem connection. For computers connected to an ethernet the choices are harder as there are many computers connected to the network.

For every IP packet transmitted, IP uses the routing tables to resolve the route for the destination IP address. Each IP destination successfully looked up in the routing tables returns a rtable data structure describing the route to use. This includes the source IP address to use, the address of the network device data structure and, sometimes, a prebuilt hardware header. This hardware header is network device specific and contains the source and destination physical addresses and other media specific information. If the network device is an ethernet device, the hardware header would be as shown in Figure 10.1 and the source and destination addresses would be physical ethernet addresses. The hardware header is cached with the route because it must be appended to each IP packet transmitted on this route and constructing

See include/-net/route.h

it takes time. The hardware header may contain physical addresses that have to be resolved using the ARP protocol. In this case the outgoing packet is stalled until the address has been resolved. Once it has been resolved and the hardware header built, the hardware header is cached so that future IP packets sent using this interface do not have to ARP.

10.5.4 Data Fragmentation

Every network device has a maximum packet size and it cannot transmit or receive a data packet bigger than this. The IP protocol allows for this and will fragment data into smaller units to fit into the packet size that the network device can handle. The IP protocol header includes a fragment field which contains a flag and the fragment offset.

See `ip_build_xmit()` in `net/ipv4/ip_output.c`

When an IP packet is ready to be transmited, IP finds the network device to send the IP packet out on. This device is found from the IP routing tables. Each `device` has a field describing its maximum transfer unit (in bytes), this is the `mtu` field. If the device's mtu is smaller than the packet size of the IP packet that is waiting to be transmitted, then the IP packet must be broken down into smaller (mtu sized) fragments. Each fragment is represented by an `sk_buff`; its IP header marked to show that it is a fragment and what offset into the data this IP packet contains. The last packet is marked as being the last IP fragment. If, during the fragmentation, IP cannot allocate an `sk_buff`, the transmit will fail.

See `ip_rcv()` in `net/ipv4/ip_input.c`

Receiving IP fragments is a little more difficult than sending them because the IP fragments can be received in any order and they must all be received before they can be reassembled. Each time an IP packet is received it is checked to see if it is an IP fragment. The first time that the fragment of a message is received, IP creates a new `ipq` data structure, and this is linked into the `ipqueue` list of IP fragments awaiting recombination. As more IP fragments are received, the correct `ipq` data structure is found and a new `ipfrag` data structure is created to describe this fragment. Each `ipq` data structure uniquely describes a fragmented IP receive frame with its source and destination IP addresses, the upper layer protocol identifier and the identifier for this IP frame. When all of the fragments have been received, they are combined into a single `sk_buff` and passed up to the next protocol level to be processed. Each `ipq` contains a timer that is restarted each time a valid fragment is received. If this timer expires, the `ipq` data structure and its `ipfrag`'s are dismantled and the message is presumed to have been lost in transit. It is then up to the higher level protocols to retransmit the message.

10.6 The Address Resolution Protocol (ARP)

See `ip_build_xmit()` in `net/ipv4/ip_output.c`

See `eth_rebuild_header()` in `net/ethernet/eth.c`

The Address Resolution Protocol's role is to provide translations of IP addresses into physical hardware addresses such as ethernet addresses. IP needs this translation just before it passes the data (in the form of an `sk_buff`) to the device driver for transmission. It performs various checks to see if this device needs a hardware header and, if it does, if the hardware header for the packet needs to be rebuilt. Linux caches hardware headers to avoid frequent rebuilding of them. If the hardware header needs rebuilding, it calls the device specific hardware header rebuilding routine. All ethernet devices use the same generic header rebuilding routine which in turn uses the ARP services to translate the destination IP address into a physical address.

The ARP protocol itself is very simple and consists of two message types, an ARP request and an ARP reply. The ARP request contains the IP address that needs translating and the reply (hopefully) contains the translated IP address, the hardware address. The ARP request is broadcast to all hosts connected to the network, so, for an ethernet network, all of the machines connected to the ethernet will see the ARP request. The machine that owns the IP address in the request will respond to the ARP request with an ARP reply containing its own physical address.

The ARP protocol layer in Linux is built around a table of `arp_table` data structures which each describe an IP to physical address translation. These entries are created as IP addresses need to be translated and removed as they become stale over time. Each `arp_table` data structure has the following fields:

last used	the time that this ARP entry was last used,
last updated	the time that this ARP entry was last updated,
flags	these describe this entry's state, if it is complete and so on,
IP address	The IP address that this entry describes
hardware address	The translated hardware address
hardware header	This is a pointer to a cached hardware header,
timer	This is a `timer_list` entry used to time out ARP requests that do not get a response,
retries	The number of times that this ARP request has been retried,
`sk_buff` queue	List of `sk_buff` entries waiting for this IP address to be resolved

The ARP table consists of a table of pointers (the `arp_tables` vector) to chains of `arp_table` entries. The entries are cached to speed up access to them, each entry is found by taking the last two bytes of its IP address to generate an index into the table and then following the chain of entries until the correct one is found. Linux also caches prebuilt hardware headers off the `arp_table` entries in the form of `hh_cache` data structures.

When an IP address translation is requested and there is no corresponding `arp_table` entry, ARP must send an ARP request message. It creates a new `arp_table` entry in the table and queues the `sk_buff` containing the network packet that needs the address translation on the `sk_buff` queue of the new entry. It sends out an ARP request and sets the ARP expiry timer running. If there is no response then ARP will retry the request a number of times and if there is still no response ARP will remove the `arp_table` entry. Any `sk_buff` data structures queued waiting for the IP address to be translated will be notified and it is up to the protocol layer that is transmitting them to cope with this failure. UDP does not care about lost packets but TCP will attempt to retransmit on an established TCP link. If the owner of the IP address responds with its hardware address, the `arp_table` entry is marked as complete and any queued `sk_buff`'s will be removed from the queue and will go on to be transmitted. The hardware address is written into the hardware header of each `sk_buff`.

The ARP protocol layer must also respond to ARP requests that specfy its IP address. It registers its protocol type (`ETH_P_ARP`), generating a `packet_type` data structure. This means that it will be passed all ARP packets that are received by the network devices. As well as ARP replies, this includes ARP requests. It generates an ARP reply using the hardware address kept in the receiving device's `device` data structure.

Network topologies can change over time and IP addresses can be reassigned to different hardware addresses. For example, some dial up services assign an IP address as each connection is established. In order that the ARP table contains up to date entries, ARP runs a periodic timer which looks through all of the `arp_table` entries to see which have timed out. It is very careful not to remove entries that contain one or more cached hardware headers. Removing these entries is dangerous as other data structures rely on them. Some `arp_table` entries are permanent and these are marked so that they will not be deallocated. The ARP table cannot be allowed to grow too large; each `arp_table` entry consumes some kernel memory. Whenever the a new entry needs to be allocated and the ARP table has reached its maximum size the table is pruned by searching out the oldest entries and removing them.

10.7 IP Routing

The IP routing function determines where to send IP packets destined for a particular IP address. There are many choices to be made when transmitting IP packets. Can the destination be reached at all? If it can be reached, which network device should be used to transmit it? If there is more than one network device that could be used to reach the destination, which is the better one? The IP routing database maintains information that gives answers to these questions. There are two databases, the most important being the Forwarding Information Database. This is an exhaustive list of known IP destinations and their best routes. A smaller and much faster database, the *route cache* is used for quick lookups of routes for IP destinations. Like all caches, it must contain only the frequently accessed routes; its contents are derived from the Forwarding Information Database.

Routes are added and deleted via IOCTL requests to the BSD socket interface. These are passed onto the protocol to process. The INET protocol layer only allows processes with superuser privileges to add and delete IP routes. These routes can be fixed or they can be dynamic and change over time. Most systems use fixed routes unless they themselves are routers. Routers run routing protocols which constantly check on the availability of routes to all known IP destinations. Systems that are not routers are known as end systems. The routing protocols are implemented as daemons, for example GATED, and they also add and delete routes via the IOCTL BSD socket interface.

10.7.1 The Route Cache

Whenever an IP route is looked up, the route cache is first checked for a matching route. If there is no matching route in the route cache the Forwarding Information Database is searched for a route. If no route can be found there, the IP packet will fail to be sent and the application notified. If a route is in the Forwarding Information Database and not in the route cache, then a new entry is generated and added into the route cache for this route. The route cache is a table (`ip_rt_hash_table`) that contains pointers to chains of `rtable` data structures. The index into the route table is a hash function based on the least significant two bytes of the IP address. These are the two bytes most likely to be different between destinations and provide the

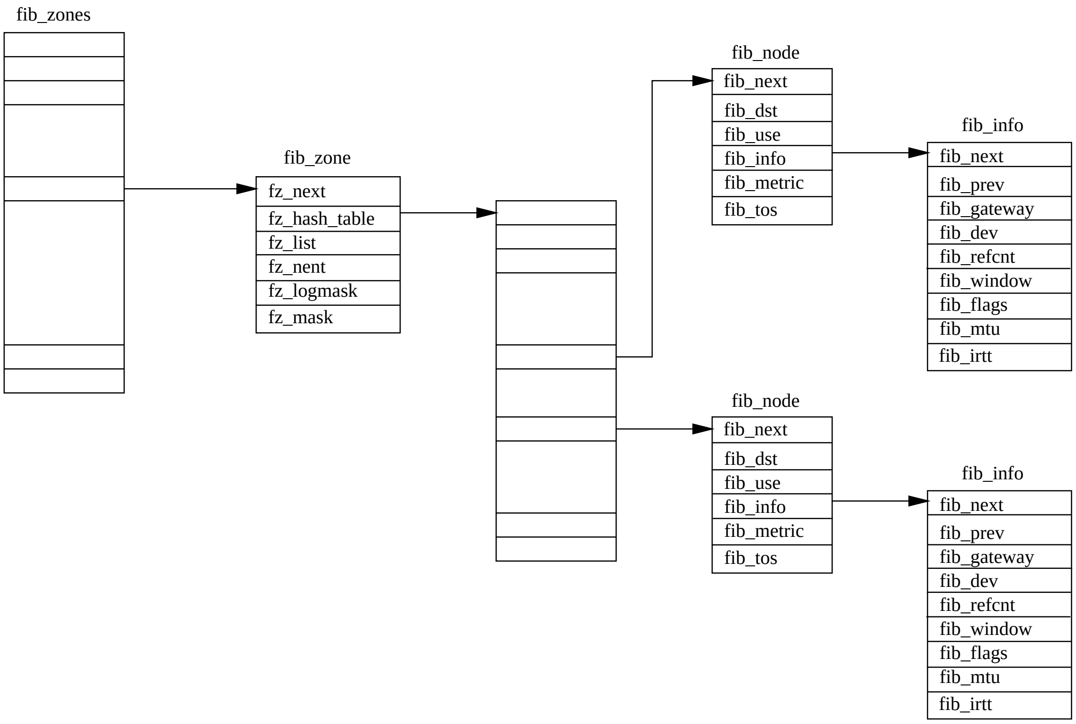

Figure 10.5: The Forwarding Information Database

best spread of hash values. Each `rtable` entry contains information about the route; the destination IP address, the network `device` to use to reach that IP address, the maximum size of message that can be used and so on. It also has a reference count, a usage count and a timestamp of the last time that they were used (in `jiffies`). The reference count is incremented each time the route is used to show the number of network connections using this route. It is decremented as applications stop using the route. The usage count is incremented each time the route is looked up and is used to order the `rtable` entry in its chain of hash entries. The last used timestamp for all of the entries in the route cache is periodically checked to see if the `rtable` is too old . If the route has not been recently used, it is discarded from the route cache. If routes are kept in the route cache they are ordered so that the most used entries are at the front of the hash chains. This means that finding them will be quicker when routes are looked up.

See `ip_rt_check_expire()` in `net/ipv4/route.c`

10.7.2 The Forwarding Information Database

The forwarding information database (shown in Figure 10.5 contains IP's view of the routes available to this system at this time. It is quite a complicated data structure and, although it is reasonably efficiently arranged, it is not a quick database to consult. In particular it would be very slow to look up destinations in this database for every IP packet transmitted. This is the reason that the route cache exists: to speed up IP packet transmission using known good routes. The route cache is derived from the forwarding database and represents its commonly used entries.

Each IP subnet is represented by a `fib_zone` data structure. All of these are pointed at from the `fib_zones` hash table. The hash index is derived from the IP subnet mask. All routes to the same subnet are described by pairs of `fib_node` and `fib_info` data structures queued onto the `fz_list` of each `fib_zone` data structure. If the number of routes in this subnet grows large, a hash table is generated to make finding the `fib_node` data structures easier.

Several routes may exist to the same IP subnet and these routes can go through one of several gateways. The IP routing layer does not allow more than one route to a subnet using the same gateway. In other words, if there are several routes to a subnet, then each route is guaranteed to use a different gateway. Associated with each route is its *metric*. This is a measure of how advantagious this route is. A route's metric is, essentially, the number of IP subnets that it must hop across before it reaches the destination subnet. The higher the metric, the worse the route.

Chapter 11

Kernel Mechanisms

This chapter describes some of the general tasks and mechanisms that the Linux kernel needs to supply so that other parts of the kernel work effectively together.

11.1 Bottom Half Handling

There are often times in a kernel when you do not want to do work at this moment. A good example of this is during interrupt processing. When the interrupt was asserted, the processor stopped what it was doing and the operating system delivered the interrupt to the appropriate device driver. Device drivers should not spend too much time handling interrupts as, during this time, nothing else in the system can run. There is often some work that could just as well be done later on. Linux's bottom half handlers were invented so that device drivers and other parts of the Linux kernel could queue work to be done later on. Figure 11.1 shows the kernel data structures associated with bottom half handling. There can be up to 32 different

See `include/linux/-interrupt.h`

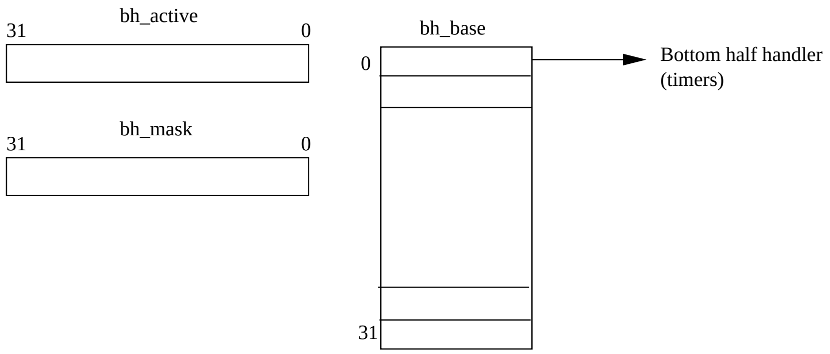

Figure 11.1: Bottom Half Handling Data Structures

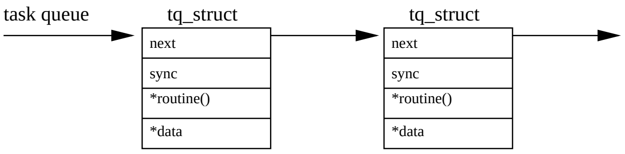

Figure 11.2: A Task Queue

bottom half handlers; `bh_base` is a vector of pointers to each of the kernel's bottom half handling routines. `bh_active` and `bh_mask` have their bits set according to what handlers have been installed and are active. If bit N of `bh_mask` is set then the Nth element of `bh_base` contains the address of a bottom half routine. If bit N of `bh_active` is set then the N'th bottom half handler routine should be called as soon as the scheduler deems reasonable. These indices are statically defined; the timer bottom half handler is the highest priority (index 0), the console bottom half handler is next in priority (index 1) and so on. Typically the bottom half handling routines have lists of tasks associated with them. For example, the *immediate* bottom half handler works its way through the immediate tasks queue (`tq_immediate`) which contains tasks that need to be performed immediately.

Some of the kernel's bottom half handers are device specific, but others are more generic:

TIMER This handler is marked as active each time the system's periodic timer interrupts and is used to drive the kernel's timer queue mechanisms,

CONSOLE This handler is used to process console messages,

TQUEUE This handler is used to process `tty` messages,

NET This handler handles general network processing,

IMMEDIATE This is a generic handler used by several device drivers to queue work to be done later.

Whenever a device driver, or some other part of the kernel, needs to schedule work to be done later, it adds work to the appropriate system queue, for example the timer queue, and then signals the kernel that some bottom half handling needs to be done. It does this by setting the appropriate bit in `bh_active`. Bit 8 is set if the driver has queued something on the immediate queue and wishes the immediate bottom half handler to run and process it. The `bh_active` bitmask is checked at the end of each system call, just before control is returned to the calling process. If it has any bits set, the bottom half handler routines that are active are called. Bit 0 is checked first, then 1 and so on until bit 31. The bit in `bh_active` is cleared as each bottom half handling routine is called. `bh_active` is transient; it only has meaning between calls to the scheduler and is a way of not calling bottom half handling routines when there is no work for them to do.

See `do_bottom_half()` in `kernel/softirq.c`

11.2 Task Queues

Task queues are the kernel's way of deferring work until later. Linux has a generic mechanism for queuing work on queues and for processing them later. Task queues

See `include/linux/tqueue.h`

are often used in conjunction with bottom half handlers; the timer task queue is processed when the timer queue bottom half handler runs. A task queue is a simple data structure, see figure 11.2 which consists of a singly linked list of `tq_struct` data structures each of which contains the address of a routine and a pointer to some data. The routine will be called when the element on the task queue is processed and it will be passed a pointer to the data.

Anything in the kernel, for example a device driver, can create and use task queues but there are three task queues created and managed by the kernel:

timer This queue is used to queue work that will be done as soon after the next system clock tick as is possible. Each clock tick, this queue is checked to see if it contains any entries and, if it does, the timer queue bottom half handler is made active. The timer queue bottom half handler is processed, along with all the other bottom half handlers, when the scheduler next runs. This queue should not be confused with system timers, which are a much more sophisticated mechanism.

immediate This queue is also processed when the scheduler processes the active bottom half handlers. The immediate bottom half handler is not as high in priority as the timer queue bottom half handler and so these tasks will be run later.

scheduler This task queue is processed directly by the scheduler. It is used to support other task queues in the system and, in this case, the task to be run will be a routine that processes a task queue, say for a device driver.

When task queues are processed, the pointer to the first element in the queue is removed from the queue and replaced with a null pointer. In fact, this removal is an atomic operation, one that cannot be interrupted. Then each element in the queue has its handling routine called in turn. The elements in the queue are often statically allocated data. However there is no inherent mechanism for discarding allocated memory. The task queue processing routine simply moves onto the next element in the list. It is the job of the task itself to ensure that it properly cleans up any allocated kernel memory.

11.3 Timers

An operating system needs to be able to schedule an activity sometime in the future. A mechanism is needed whereby activities can be scheduled to run at some relatively precise time. Any microprocessor that wishes to support an operating system must have a programmable interval timer that periodically interrupts the processor. This periodic interrupt is known as a system clock tick and it acts like a metronome, orchestrating the system's activities. Linux has a very simple view of what time it is; it measures time in clock ticks since the system booted. All system times are based on this measurement, which is known as `jiffies` after the globally available variable of the same name.

Linux has two types of system timers, both queue routines to be called at some system time but they are slightly different in their implementations. Figure 11.3 shows both mechanisms. The first, the old timer mechanism, has a static array of 32 pointers to `timer_struct` data structures and a mask of active timers, `timer_active`.

See `include/-linux/timer.h`

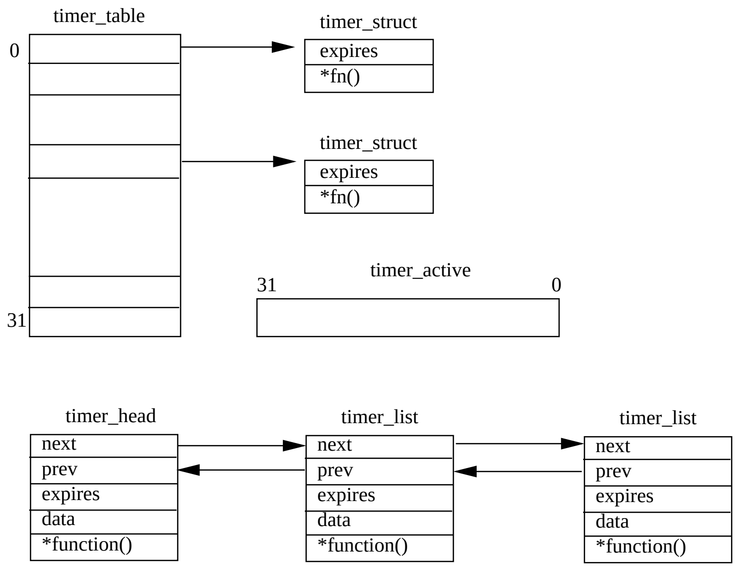

Figure 11.3: System Timers

Where the timers go in the timer table is statically defined (rather like the bottom half handler table `bh_base`). Entries are added into this table mostly at system initialization time. The second, newer, mechanism uses a linked list of `timer_list` data structures held in ascending expiry time order.

Both methods use the time in `jiffies` as an expiry time so that a timer that wished to run in 5s would have to convert 5s to units of `jiffies` and add that to the current system time to get the system time in `jiffies` when the timer should expire. Every system clock tick the timer bottom half handler is marked as active so that the when the scheduler next runs, the timer queues will be processed. The timer bottom half handler processes both types of system timer. For the old system timers the `timer_active` bit mask is check for bits that are set. If the expiry time for an active timer has expired (expiry time is less than the current system `jiffies`), its timer routine is called and its active bit is cleared. For new system timers, the entries in the linked list of `timer_list` data structures are checked. Every expired timer is removed from the list and its routine is called. The new timer mechanism has the advantage of being able to pass an argument to the timer routine.

See `timer_bh()` in `kernel/sched.c`

See `run_old_timers()` in `kernel/sched.c`

See `run_timer_list()` in `kernel/sched.c`

11.4 Wait Queues

There are many times when a process must wait for a system resource. For example a process may need the VFS inode describing a directory in the file system and that inode may not be in the buffer cache. In this case the process must wait for that inode to be fetched from the physical media containing the file system before it can carry on.

See `include/-linux/wait.h`

The Linux kernel uses a simple data structure, a wait queue (see figure 11.4), which consists of a pointer to the processes `task_struct` and a pointer to the next element in the wait queue.

When processes are added to the end of a wait queue they can either be inter-

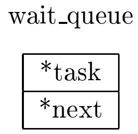

Figure 11.4: Wait Queue

ruptible or uninterruptible. Interruptible processes may be interrupted by events such as timers expiring or signals being delivered whilst they are waiting on a wait queue. The waiting processes state will reflect this and either be `INTERRUPTIBLE` or `UNINTERRUPTIBLE`. As this process can not now continue to run, the scheduler is run and, when it selects a new process to run, the waiting process will be suspended. [1]

When the wait queue is processed, the state of every process in the wait queue is set to `RUNNING`. If the process has been removed from the run queue, it is put back onto the run queue. The next time the scheduler runs, the processes that are on the wait queue are now candidates to be run as they are now no longer waiting. When a process on the wait queue is scheduled the first thing that it will do is remove itself from the wait queue. Wait queues can be used to synchronize access to system resources and they are used by Linux in its implementation of semaphores (see below).

11.5 Buzz Locks

These are better known as spin locks and they are a primitive way of protecting a data structure or piece of code. They only allow one process at a time to be within a critical region of code. They are used in Linux to restrict access to fields in data structures, using a single integer field as a lock. Each process wishing to enter the region attempts to change the lock's initial value from 0 to 1. If its current value is 1, the process tries again, spinning in a tight loop of code. The access to the memory location holding the lock must be atomic, the action of reading its value, checking that it is 0 and then changing it to 1 cannot be interrupted by any other process. Most CPU architectures provide support for this via special instructions but you can also implement buzz locks using uncached main memory.

When the owning process leaves the critical region of code it decrements the buzz lock, returning its value to 0. Any processes spinning on the lock will now read it as 0, the first one to do this will increment it to 1 and enter the critical region.

11.6 Semaphores

Semaphores are used to protect critical regions of code or data structures. Remember that each access of a critical piece of data such as a VFS inode describing a directory is made by kernel code running on behalf of a process. It would be very dangerous to allow one process to alter a critical data structure that is being used by another process. One way to achieve this would be to use a buzz lock around the critical piece of data is being accessed but this is a simplistic approach that would not give very good system performance. Instead Linux uses semaphores to allow just one process

[1] REVIEW NOTE: *What is to stop a task in state `INTERRUPTIBLE` being made to run the next time the scheduler runs? Processes in a wait queue should never run until they are woken up.*

at a time to access critical regions of code and data; all other processes wishing to access this resource will be made to wait until it becomes free. The waiting processes are suspended, other processes in the system can continue to run as normal.

See `include/-asm/semaphore.h`

A Linux `semaphore` data structure contains the following information:

count This field keeps track of the count of processes wishing to use this resource. A positive value means that the resource is available. A negative or zero value means that processes are waiting for it. An initial value of 1 means that one and only one process at a time can use this resource. When processes want this resource they decrement the count and when they have finished with this resource they increment the count,

waking This is the count of processes waiting for this resource which is also the number of process waiting to be woken up when this resource becomes free,

wait queue When processes are waiting for this resource they are put onto this wait queue,

lock A buzz lock used when accessing the `waking` field.

Suppose the initial count for a semaphore is 1, the first process to come along will see that the count is positive and decrement it by 1, making it 0. The process now "owns" the critical piece of code or resource that is being protected by the semaphore. When the process leaves the critical region it increments the semphore's count. The most optimal case is where there are no other processes contending for ownership of the critical region. Linux has implemented semaphores to work efficiently for this, the most common, case.

If another process wishes to enter the critical region whilst it is owned by a process it too will decrement the count. As the count is now negative (-1) the process cannot enter the critical region. Instead it must wait until the owning process exits it. Linux makes the waiting process sleep until the owning process wakes it on exiting the critical region. The waiting process adds itself to the semaphore's wait queue and sits in a loop checking the value of the `waking` field and calling the scheduler until `waking` is non-zero.

The owner of the critical region increments the semaphore's count and if it is less than or equal to zero then there are processes sleeping, waiting for this resource. In the optimal case the semaphore's count would have been returned to its initial value of 1 and no further work would be neccessary. The owning process increments the waking counter and wakes up the process sleeping on the semaphore's wait queue. When the waiting process wakes up, the waking counter is now 1 and it knows that it it may now enter the critical region. It decrements the waking counter, returning it to a value of zero, and continues. All access to the waking field of semaphore are protected by a buzz lock using the semaphore's lock.

Chapter 12

Modules

This chapter describes how the Linux kernel can dynamically load functions, for example filesystems, only when they are needed.

Linux is a monolithic kernel; that is, it is one, single, large program where all the functional components of the kernel have access to all of its internal data structures and routines. The alternative is to have a micro-kernel structure where the functional pieces of the kernel are broken out into separate units with strict communication mechanisms between them. This makes adding new components into the kernel via the configuration process rather time consuming. Say you wanted to use a SCSI driver for an NCR 810 SCSI and you had not built it into the kernel. You would have to configure and then build a new kernel before you could use the NCR 810. There is an alternative, Linux allows you to dynamically load and unload components of the operating system as you need them. Linux modules are lumps of code that can be dynamically linked into the kernel at any point after the system has booted. They can be unlinked from the kernel and removed when they are no longer needed. Mostly Linux kernel modules are device drivers, pseudo-device drivers such as network drivers, or file-systems.

You can either load and unload Linux kernel modules explicitly using the insmod and rmmod commands or the kernel itself can demand that the kernel daemon (`kerneld`) loads and unloads the modules as they are needed. Dynamically loading code as it is needed is attractive as it keeps the kernel size to a minimum and makes the kernel very flexible. My current Intel kernel uses modules extensively and is only 406Kbytes long. I only occasionally use `VFAT` file systems and so I build my Linux kernel to automatically load the `VFAT` file system module as I mount a `VFAT` partition. When I have unmounted the `VFAT` partition the system detects that I no longer need the `VFAT` file system module and removes it from the system. Modules can also be useful for trying out new kernel code without having to rebuild and reboot the kernel every time you try it out. Nothing, though, is for free and there is a slight performance and memory penalty associated with kernel modules. There is a little more code that a loadable module must provide and this and the extra data structures take a little more memory. There is also a level of indirection introduced that makes accesses of kernel resources slightly less efficient for modules.

Once a Linux module has been loaded it is as much a part of the kernel as any normal kernel code. It has the same rights and responsibilities as any kernel code; in other words, Linux kernel modules can crash the kernel just like all kernel code or device drivers can.

So that modules can use the kernel resources that they need, they must be able to find them. Say a module needs to call `kmalloc()`, the kernel memory allocation routine. At the time that it is built, a module does not know where in memory `kmalloc()` is, so when the module is loaded, the kernel must fix up all of the module's references to `kmalloc()` before the module can work. The kernel keeps a list of all of the kernel's resources in the kernel symbol table so that it can resolve references to those resources from the modules as they are loaded. Linux allows module stacking, this is where one module requires the services of another module. For example, the `VFAT` file system module requires the services of the `FAT` file system module as the `VFAT` file system is more or less a set of extensions to the `FAT` file system. One module requiring services or resources from another module is very similar to the situation where a module requires services and resources from the kernel itself. Only here the required services are in another, previously loaded module. As each module is loaded, the kernel modifies the kernel symbol table, adding to it all of the resources or symbols exported by the newly loaded module. This means that, when the next module is loaded, it has access to the services of the already loaded modules.

When an attempt is made to unload a module, the kernel needs to know that the module is unused and it needs some way of notifying the module that it is about to be unloaded. That way the module will be able to free up any system resources that it has allocated, for example kernel memory or interrupts, before it is removed from the kernel. When the module is unloaded, the kernel removes any symbols that that module exported into the kernel symbol table.

Apart from the ability of a loaded module to crash the operating system by being badly written, it presents another danger. What happens if you load a module built for an earlier or later kernel than the one that you are now running? This may cause a problem if, say, the module makes a call to a kernel routine and supplies the wrong arguments. The kernel can optionally protect against this by making rigorous version checks on the module as it is loaded.

12.1 Loading a Module

There are two ways that a kernel module can be loaded. The first way is to use the insmod command to manually insert the it into the kernel. The second, and much more clever way, is to load the module as it is needed; this is known as demand loading. When the kernel discovers the need for a module, for example when the user mounts a file system that is not in the kernel, the kernel will request that the kernel daemon (`kerneld`) attempts to load the appropriate module.

`kerneld` is in the modules package along with insmod, lsmod and rmmod.

The kernel daemon is a normal user process albeit with super user privileges. When it is started up, usually at system boot time, it opens up an Inter-Process Communication (IPC) channel to the kernel. This link is used by the kernel to send messages to the `kerneld` asking for various tasks to be performed. `Kerneld`'s major function

See `include/-linux/kerneld.h`

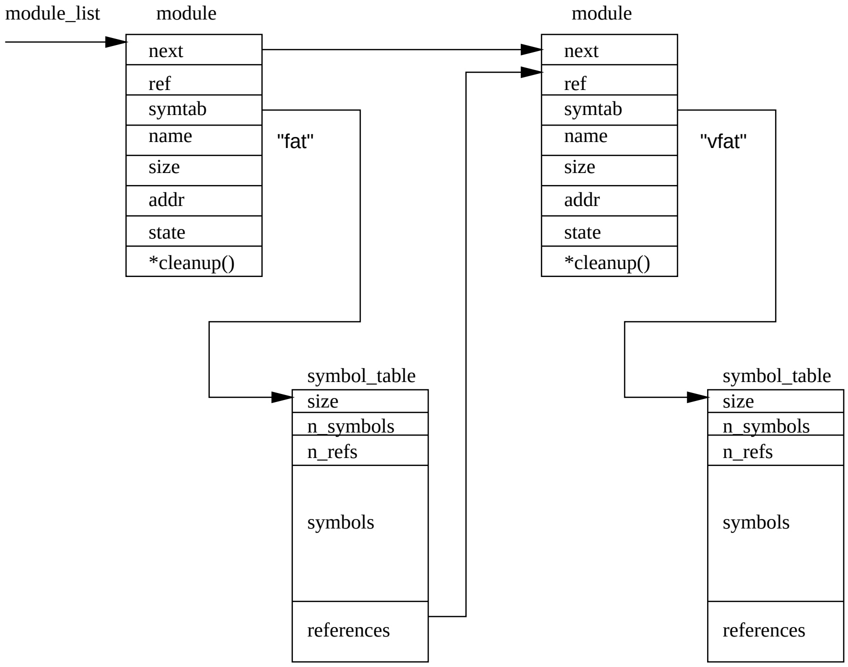

Figure 12.1: The List of Kernel Modules

is to load and unload kernel modules but it is also capable of other tasks such as starting up the PPP link over serial line when it is needed and closing it down when it is not. `Kerneld` does not perform these tasks itself, it runs the neccessary programs such as insmod to do the work. `Kerneld` is just an agent of the kernel, scheduling work on its behalf.

The insmod utility must find the requested kernel module that it is to load. Demand loaded kernel modules are normally kept in `/lib/modules/kernel-version`. The kernel modules are linked object files just like other programs in the system except that they are linked as a relocatable images. That is, images that are not linked to run from a particular address. They can be either `a.out` or `elf` format object files. insmod makes a privileged system call to find the kernel's exported symbols. These are kept in pairs containing the symbol's name and its value, for example its address. The kernel's exported symbol table is held in the first `module` data structure in the list of modules maintained by the kernel and pointed at by the `module_list` pointer. Only specifically entered symbols are added into the table, which is built when the kernel is compiled and linked, not *every* symbol in the kernel is exported to its modules. An example symbol is ``` ``request_irq'' ``` which is the kernel routine that must be called when a driver wishes to take control of a particular system interrupt. In my current kernel, this has a value of *0x0010cd30*. You can easily see the exported kernel symbols and their values by looking at `/proc/ksyms` or by using the ksyms utility. The ksyms utility can either show you all of the exported kernel symbols or only those symbols exported by loaded modules. insmod reads the module into its virtual memory and fixes up its unresolved references to kernel routines and resources using the exported symbols from the kernel. This fixing up takes the form of patching the module image in memory. insmod physically writes the address of the symbol into the appropriate place in the module.

See `sys_get_kernel_syms()` in `kernel/module.c`

See `include/linux/module.h`

See sys_-create_module() in kernel/-module.c.

When insmod has fixed up the module's references to exported kernel symbols, it asks the kernel for enough space to hold the new kernel, again using a privileged system call. The kernel allocates a new module data structure and enough kernel memory to hold the new module and puts it at the end of the kernel modules list. The new module is marked as UNINITIALIZED. Figure 12.1 shows the list of kernel modules after two modules, VFAT and VFAT have been loaded into the kernel. Not shown in the diagram is the first module on the list, which is a pseudo-module that is only there to hold the kernel's exported symbol table. You can use the command lsmod to list all of the loaded kernel modules and their interdependencies. lsmod simply reformats /proc/modules which is built from the list of kernel module data structures. The memory that the kernel allocates for it is mapped into the insmod process's address space so that it can access it. insmod copies the module into the allocated space and relocates it so that it will run from the kernel address that it has been allocated. This must happen as the module cannot expect to be loaded at the same address twice let alone into the same address in two different Linux systems. Again, this relocation involves patching the module image with the appropriate addresses.

See sys_init_-module() in kernel/-module.c.

The new module also exports symbols to the kernel and insmod builds a table of these exported images. Every kernel module must contain module initialization and module cleanup routines and these symbols are deliberately not exported but insmod must know the addresses of them so that it can pass them to the kernel. All being well, insmod is now ready to initialize the module and it makes a privileged system call passing the kernel the addresses of the module's initialization and cleanup routines.

When a new module is added into the kernel, it must update the kernel's set of symbols and modify the modules that are being used by the new module. Modules that have other modules dependent on them must maintain a list of references at the end of their symbol table and pointed at by their module data structure. Figure 12.1 shows that the VFAT file system module is dependent on the FAT file system module. So, the FAT module contains a reference to the VFAT module; the reference was added when the VFAT module was loaded. The kernel calls the modules initialization routine and, if it is successful it carries on installing the module. The module's cleanup routine address is stored in it's module data structure and it will be called by the kernel when that module is unloaded. Finally, the module's state is set to RUNNING.

12.2 Unloading a Module

Modules can be removed using the rmmod command but demand loaded modules are automatically removed from the system by kerneld when they are no longer being used. Every time its idle timer expires, kerneld makes a system call requesting that all unused demand loaded modules are removed from the system. The timer's value is set when you start kerneld; my kerneld checks every 180 seconds. So, for example, if you mount an iso9660 CD ROM and your iso9660 filesystem is a loadable module, then shortly after the CD ROM is unmounted, the iso9660 module will be removed from the kernel.

A module cannot be unloaded so long as other components of the kernel are depending on it. For example, you cannot unload the VFAT module if you have one or more VFAT file systems mounted. If you look at the output of lsmod, you will see that each module has a count associated with it. For example:

```
Module:        #pages:  Used by:
msdos              5                1
vfat               4                1 (autoclean)
fat                6    [vfat msdos]  2 (autoclean)
```

The count is the number of kernel entities that are dependent on this module. In the above example, the `vfat` and `msdos` modules are both dependent on the `fat` module and so it has a count of 2. Both the `vfat` and `msdos` modules have 1 dependent, which is a mounted file system. If I were to load another `VFAT` file system then the `vfat` module's count would become 2. A module's count is held in the first longword of its image.

This field is slightly overloaded as it also holds the `AUTOCLEAN` and `VISITED` flags. Both of these flags are used for demand loaded modules. These modules are marked as `AUTOCLEAN` so that the system can recognize which ones it may automatically unload. The `VISITED` flag marks the module as in use by one or more other system components; it is set whenever another component makes use of the module. Each time the system is asked by `kerneld` to remove unused demand loaded modules it looks through all of the modules in the system for likely candidates. It only looks at modules marked as `AUTOCLEAN` and in the state `RUNNING`. If the candidate has its `VISITED` flag cleared then it will remove the module, otherwise it will clear the `VISITED` flag and go on to look at the next module in the system.

Assuming that a module can be unloaded, its cleanup routine is called to allow it to free up the kernel resources that it has allocated. The `module` data structure is marked as `DELETED` and it is unlinked from the list of kernel modules. Any other modules that it is dependent on have their reference lists modified so that they no longer have it as a dependent. All of the kernel memory that the module needed is deallocated.

See `sys_delete_module()` in `kernel/module.c`

Chapter 13

Processors

Linux runs on a number of processors; this chapter gives a brief outline of each of them.

13.1 X86

TBD

13.2 ARM

The ARM processor implements a low power, high performance 32 bit RISC architecture. It is being widely used in embedded devices such as mobile phones and PDAs (Personal Data Assistants). It has 31 32 bit registers with 16 visible in any mode. Its instructions are simple load and store instructions (load a value from memory, perform an operation and store the result back into memory). One interesting feature it has is that every instruction is conditional. For example, you can test the value of a register and, until you next test for the same condition, you can conditionally execute instructions as and when you like. Another interesting feature is that you can perform arithmetic and shift operations on values as you load them. It operates in several modes, including a system mode that can be entered from user mode via a SWI (software interrupt).

It is a synthasisable core and ARM (the company) does not itself manufacture processors. Instead the ARM partners (companies such as Intel or LSI for example) implement the ARM architecture in silicon. It allows other processors to be tightly coupled via a co-processor interface and it has several memory management unit variations. These range from simple memory protection schemes to complex page hierarchies.

13.3 Alpha AXP Processor

The Alpha AXP architecture is a 64-bit load/store RISC architecture designed with speed in mind. All registers are 64 bits in length; 32 integer registers and 32 floating point registers. Integer register 31 and floating point register 31 are used for null operations. A read from them generates a zero value and a write to them has no effect. All instructions are 32 bits long and memory operations are either reads or writes. The architecture allows different implementations so long as the implementations follow the architecture.

There are no instructions that operate directly on values stored in memory; all data manipulation is done between registers. So, if you want to increment a counter in memory, you first read it into a register, then modify it and write it out. The instructions only interact with each other by one instruction writing to a register or memory location and another register reading that register or memory location. One interesting feature of Alpha AXP is that there are instructions that can generate flags, such as testing if two registers are equal, the result is not stored in a processor status register, but is instead stored in a third register. This may seem strange at first, but removing this dependency from a status register means that it is much easier to build a CPU which can issue multiple instructions every cycle. Instructions on unrelated registers do not have to wait for each other to execute as they would if there were a single status register. The lack of direct operations on memory and the large number of registers also help issue multiple instructions.

The Alpha AXP architecture uses a set of subroutines, called privileged architecture library code (PALcode). PALcode is specific to the operating system, the CPU implementation of the Alpha AXP architecture and to the system hardware. These subroutines provide operating system primitives for context switching, interrupts, exceptions and memory management. These subroutines can be invoked by hardware or by CALL_PAL instructions. PALcode is written in standard Alpha AXP assembler with some implementation specific extensions to provide direct access to low level hardware functions, for example internal processor registers. PALcode is executed in PALmode, a privileged mode that stops some system events happening and allows the PALcode complete control of the physical system hardware.

Chapter 14

The Linux Kernel Sources

This chapter describes where in the Linux kernel sources you should start looking for particular kernel functions.

This book does not depend on a knowledge of the 'C' programming language or require that you have the Linux kernel sources available in order to understand how the Linux kernel works. That said, it is a fruitful exercise to look at the kernel sources to get an in-depth understanding of the Linux operating system. This chapter gives an overview of the kernel sources; how they are arranged and where you might start to look for particular code.

Where to Get The Linux Kernel Sources

All of the major Linux distributions (Craftworks, Debian, Slackware, Red Hat etcetera) include the kernel sources in them. Usually the Linux kernel that got installed on your Linux system was built from those sources. By their very nature these sources tend to be a little out of date so you may want to get the latest sources from one of the web sites mentioned in chapter B. They are kept on `ftp://ftp.cs.helsinki.fi` and all of the other web sites shadow them. This makes the Helsinki web site the most up to date, but sites like MIT and Sunsite are never very far behind.

If you do not have access to the web, there are many CD ROM vendors who offer snapshots of the world's major web sites at a very reasonable cost. Some even offer a subscription service with quarterly or even monthly updates. Your local Linux User Group is also a good source of sources.

The Linux kernel sources have a very simple numbering system. Any even number kernel (for example `2.0.30`) is a stable, released, kernel and any odd numbered kernel (for example `2.1.42` is a development kernel. This book is based on the stable `2.0.30` source tree. Development kernels have all of the latest features and support all of the latest devices. Although they can be unstable, which may not be exactly what you want it, is important that the Linux community tries the latest kernels. That way they are tested for the whole community. Remember that it is *always* worth backing up your system thoroughly if you do try out non-production kernels.

Changes to the kernel sources are distributed as `patch` files. The `patch` utility is used to apply a series of edits to a set of source files. So, for example, if you have the 2.0.29 kernel source tree and you wanted to move to the 2.0.30 source tree, you would obtain the 2.0.30 patch file and apply the patches (edits) to that source tree:

```
$ cd /usr/src/linux
$ patch -p1 < patch-2.0.30
```

This saves copying whole source trees, perhaps over slow serial connections. A good source of kernel patches (official and unofficial) is the `http://www.linuxhq.com` web site.

How The Kernel Sources Are Arranged

At the very top level of the source tree `/usr/src/linux` you will see a number of directories:

arch The `arch` subdirectory contains all of the architecture specific kernel code. It has further subdirectories, one per supported architecture, for example `i386` and `alpha`.

include The `include` subdirectory contains most of the include files needed to build the kernel code. It too has further subdirectories including one for every architecture supported. The `include/asm` subdirectory is a soft link to the real include directory needed for this architecture, for example `include/asm-i386`. To change architectures you need to edit the kernel makefile and rerun the Linux kernel configuration program.

init This directory contains the initialization code for the kernel and it is a very good place to start looking at how the kernel works.

mm This directory contains all of the memory management code. The architecture specific memory management code lives down in `arch/*/mm/`, for example `arch/i386/mm/fault.c`.

drivers All of the system's device drivers live in this directory. They are further sub-divided into classes of device driver, for example `block`.

ipc This directory contains the kernels inter-process communications code.

modules This is simply a directory used to hold built modules.

fs All of the file system code. This is further sub-divided into directories, one per supported file system, for example `vfat` and `ext2`.

kernel The main kernel code. Again, the architecture specific kernel code is in `arch/*/kernel`.

net The kernel's networking code.

lib This directory contains the kernel's library code. The architecture specific library code can be found in `arch/*/lib/`.

scripts This directory contains the scripts (for example `awk` and `tk` scripts) that are used when the kernel is configured.

Where to Start Looking

A large complex program like the Linux kernel can be rather daunting to look at. It is rather like a large ball of string with no end showing. Looking at one part of the kernel often leads to looking at several other related files and before long you have forgotten what you were looking for. The next subsections give you a hint as to where in the source tree the best place to look is for a given subject.

System Startup and Initialization

On an Intel based system, the kernel starts when either loadlin.exe or LILO has loaded the kernel into memory and passed control to it. Look in `arch/i386/-kernel/head.S` for this part. `Head.S` does some architecture specific setup and then jumps to the `main()` routine in `init/main.c`.

Memory Management

This code is mostly in `mm` but the architecture specific code is in `arch/*/mm`. The page fault handling code is in `mm/memory.c` and the memory mapping and page cache code is in `mm/filemap.c`. The buffer cache is implemented in `mm/buffer.c` and the swap cache in `mm/swap_state.c` and `mm/swapfile.c`.

Kernel

Most of the relevent generic code is in `kernel` with the architecture specific code in `arch/*/kernel`. The scheduler is in `kernel/sched.c` and the fork code is in `kernel/fork.c`. The bottom half handling code is in `include/linux/interrupt.h`. The `task_struct` data structure can be found in `include/linux/sched.h`.

PCI

The PCI pseudo driver is in `drivers/pci/pci.c` with the system wide definitions in `include/linux/pci.h`. Each architecture has some specific PCI BIOS code, Alpha AXP's is in `arch/alpha/kernel/bios32.c`.

Interprocess Communication

This is all in `ipc`. All System V IPC objects include an `ipc_perm` data structure and this can be found in `include/linux/ipc.h`. System V messages are implemented in `ipc/msg.c`, shared memory in `ipc/shm.c` and semaphores in `ipc/sem.c`. Pipes are implemented in `ipc/pipe.c`.

Interrupt Handling

The kernel's interrupt handling code is almost all microprocessor (and often platform) specific. The Intel interrupt handling code is in `arch/i386/kernel/irq.c` and its definitions in `include/asm-i386/irq.h`.

Device Drivers

Most of the lines of the Linux kernel's source code are in its device drivers. All of Linux's device driver sources are held in `drivers` but these are further broken out by type:

/block block device drivers such as ide (in `ide.c`). If you want to look at how all of the devices that could possibly contain file systems are initialized then you should look at `device_setup()` in `drivers/block/genhd.c`. It not only initializes the hard disks but also the network as you need a network to mount `nfs` file systems. Block devices include both IDE and SCSI based devices.

/char This the place to look for character based devices such as `ttys`, serial ports and mice.

/cdrom All of the CDROM code for Linux. It is here that the special CDROM devices (such as Soundblaster CDROM) can be found. Note that the ide CD driver is `ide-cd.c` in `drivers/block` and that the SCSI CD driver is in `scsi.c` in `drivers/scsi`.

/pci This are the sources for the PCI pseudo-driver. A good place to look at how the PCI subsystem is mapped and initialized. The Alpha AXP PCI fixup code is also worth looking at in `arch/alpha/kernel/bios32.c`.

/scsi This is where to find all of the SCSI code as well as all of the drivers for the scsi devices supported by Linux.

/net This is where to look to find the network device drivers such as the DECChip 21040 PCI ethernet driver which is in `tulip.c`.

/sound This is where all of the sound card drivers are.

File Systems

The sources for the `EXT2` file system are all in the `fs/ext2/` directory with data structure definitions in `include/linux/ext2_fs.h, ext2_fs_i.h` and `ext2_fs_sb.h`. The Virtual File System data structures are described in `include/linux/fs.h` and the code is in `fs/*`. The buffer cache is implemented in `fs/buffer.c` along with the `update` kernel daemon.

Network

The networking code is kept in `net` with most of the include files in `include/net`. The BSD socket code is in `net/socket.c` and the IP version 4 INET socket code is in `net/ipv4/af_inet.c`. The generic protocol support code (including the `sk_buff` handling routines) is in `net/core` with the TCP/IP networking code in `net/ipv4`. The network device drivers are in `drivers/net`.

Modules

The kernel module code is partially in the kernel and partially in the `modules` package. The kernel code is all in `kernel/modules.c` with the data structures and kernel demon `kerneld` messages in `include/linux/module.h` and `include/linux/kerneld.h` respectively. You may want to look at the structure of an `ELF` object file in `include/linux/elf.h`.

Appendix A

Linux Data Structures

This appendix lists the major data structures that Linux uses and which are described in this book. They have been edited slightly to fit the paper.

block_dev_struct

block_dev_struct data structures are used to register block devices as available for use by the buffer cache. They are held together in the blk_dev vector.

See include/linux/blkdev.h

```
struct blk_dev_struct {
    void (*request_fn)(void);
    struct request * current_request;
    struct request   plug;
    struct tq_struct plug_tq;
};
```

buffer_head

The buffer_head data structure holds information about a block buffer in the buffer cache.

See include/linux/fs.h

```
/* bh state bits */
#define BH_Uptodate  0   /* 1 if the buffer contains valid data      */
#define BH_Dirty     1   /* 1 if the buffer is dirty                 */
#define BH_Lock      2   /* 1 if the buffer is locked                */
#define BH_Req       3   /* 0 if the buffer has been invalidated     */
#define BH_Touched   4   /* 1 if the buffer has been touched (aging) */
#define BH_Has_aged  5   /* 1 if the buffer has been aged (aging)    */
#define BH_Protected 6   /* 1 if the buffer is protected             */
#define BH_FreeOnIO  7   /* 1 to discard the buffer_head after IO    */

struct buffer_head {
```

```
  /* First cache line: */
  unsigned long      b_blocknr;    /* block number                   */
  kdev_t             b_dev;        /* device (B_FREE = free)         */
  kdev_t             b_rdev;       /* Real device                    */
  unsigned long      b_rsector;    /* Real buffer location on disk   */
  struct buffer_head *b_next;      /* Hash queue list                */
  struct buffer_head *b_this_page; /* circular list of buffers in one
                                      page                           */

  /* Second cache line: */
  unsigned long      b_state;      /* buffer state bitmap (above)    */
  struct buffer_head *b_next_free;
  unsigned int       b_count;      /* users using this block         */
  unsigned long      b_size;       /* block size                     */

  /* Non-performance-critical data follows. */
  char               *b_data;      /* pointer to data block          */
  unsigned int       b_list;       /* List that this buffer appears  */
  unsigned long      b_flushtime;  /* Time when this (dirty) buffer
                                    * should be written              */
  unsigned long      b_lru_time;   /* Time when this buffer was
                                    * last used.                     */
  struct wait_queue  *b_wait;
  struct buffer_head *b_prev;      /* doubly linked hash list        */
  struct buffer_head *b_prev_free; /* doubly linked list of buffers  */
  struct buffer_head *b_reqnext;   /* request queue                  */
};
```

device

See include/linux/netdevice.h

Every network device in the system is represented by a `device` data structure.

```
struct device
{

  /*
   * This is the first field of the "visible" part of this structure
   * (i.e. as seen by users in the "Space.c" file).  It is the name
   * the interface.
   */
  char                    *name;

  /* I/O specific fields                                           */
  unsigned long           rmem_end;        /* shmem "recv" end     */
  unsigned long           rmem_start;      /* shmem "recv" start   */
  unsigned long           mem_end;         /* shared mem end       */
  unsigned long           mem_start;       /* shared mem start     */
  unsigned long           base_addr;       /* device I/O address   */
  unsigned char           irq;             /* device IRQ number    */

  /* Low-level status flags. */
  volatile unsigned char  start,           /* start an operation   */
                          interrupt;       /* interrupt arrived    */
  unsigned long           tbusy;           /* transmitter busy     */
```

```
struct device           *next;

/* The device initialization function. Called only once.         */
int                     (*init)(struct device *dev);

/* Some hardware also needs these fields, but they are not part of
   the usual set specified in Space.c. */
unsigned char           if_port;        /* Selectable AUI,TP,   */
unsigned char           dma;            /* DMA channel          */

struct enet_statistics* (*get_stats)(struct device *dev);

/*
 * This marks the end of the "visible" part of the structure. All
 * fields hereafter are internal to the system, and may change at
 * will (read: may be cleaned up at will).
 */

/* These may be needed for future network-power-down code.       */
unsigned long           trans_start;    /* Time (jiffies) of
                                           last transmit        */
unsigned long           last_rx;        /* Time of last Rx      */
unsigned short          flags;          /* interface flags (BSD)*/
unsigned short          family;         /* address family ID    */
unsigned short          metric;         /* routing metric       */
unsigned short          mtu;            /* MTU value            */
unsigned short          type;           /* hardware type        */
unsigned short          hard_header_len; /* hardware hdr len     */
void                    *priv;          /* private data         */

/* Interface address info. */
unsigned char           broadcast[MAX_ADDR_LEN];
unsigned char           pad;
unsigned char           dev_addr[MAX_ADDR_LEN];
unsigned char           addr_len;       /* hardware addr len    */
unsigned long           pa_addr;        /* protocol address     */
unsigned long           pa_brdaddr;     /* protocol broadcast addr*/
unsigned long           pa_dstaddr;     /* protocol P-P other addr*/
unsigned long           pa_mask;        /* protocol netmask     */
unsigned short          pa_alen;        /* protocol address len */

struct dev_mc_list      *mc_list;       /* M'cast mac addrs     */
int                     mc_count;       /* No installed mcasts  */

struct ip_mc_list       *ip_mc_list;    /* IP m'cast filter chain */
__u32                   tx_queue_len;   /* Max frames per queue   */

/* For load balancing driver pair support */
unsigned long           pkt_queue;      /* Packets queued       */
struct device           *slave;         /* Slave device         */
struct net_alias_info   *alias_info;    /* main dev alias info  */
struct net_alias        *my_alias;      /* alias devs           */

/* Pointer to the interface buffers. */
```

```
    struct sk_buff_head     buffs[DEV_NUMBUFFS];

    /* Pointers to interface service routines. */
    int                     (*open)(struct device *dev);
    int                     (*stop)(struct device *dev);
    int                     (*hard_start_xmit) (struct sk_buff *skb,
                                                struct device *dev);
    int                     (*hard_header) (struct sk_buff *skb,
                                            struct device *dev,
                                            unsigned short type,
                                            void *daddr,
                                            void *saddr,
                                            unsigned len);
    int                     (*rebuild_header)(void *eth,
                                            struct device *dev,
                                            unsigned long raddr,
                                            struct sk_buff *skb);
    void                    (*set_multicast_list)(struct device *dev);
    int                     (*set_mac_address)(struct device *dev,
                                            void *addr);
    int                     (*do_ioctl)(struct device *dev,
                                            struct ifreq *ifr,
                                            int cmd);
    int                     (*set_config)(struct device *dev,
                                            struct ifmap *map);
    void                    (*header_cache_bind)(struct hh_cache **hhp,
                                            struct device *dev,
                                            unsigned short htype,
                                            __u32 daddr);
    void                    (*header_cache_update)(struct hh_cache *hh,
                                            struct device *dev,
                                            unsigned char *  haddr);
    int                     (*change_mtu)(struct device *dev,
                                            int new_mtu);
    struct iw_statistics*   (*get_wireless_stats)(struct device *dev);
  };
```

device_struct

`device_struct` data structures are used to register character and block devices (they hold its name and the set of file operations that can be used for this device). Each valid member of the `chrdevs` and `blkdevs` vectors represents a character or block device respectively.

See `fs/devices.c`

```
struct device_struct {
    const char * name;
    struct file_operations * fops;
};
```

file

See `include/linux/fs.h`

Each open file, socket etcetera is represented by a `file` data structure.

```
struct file {
  mode_t f_mode;
  loff_t f_pos;
  unsigned short f_flags;
  unsigned short f_count;
  unsigned long f_reada, f_ramax, f_raend, f_ralen, f_rawin;
  struct file *f_next, *f_prev;
  int f_owner;         /* pid or -pgrp where SIGIO should be sent */
  struct inode * f_inode;
  struct file_operations * f_op;
  unsigned long f_version;
  void *private_data;  /* needed for tty driver, and maybe others */
};
```

files_struct

See include/linux/sched.h

The `files_struct` data structure describes the files that a process has open.

```
struct files_struct {
  int count;
  fd_set close_on_exec;
  fd_set open_fds;
  struct file * fd[NR_OPEN];
};
```

fs_struct

See include/linux/sched.h

```
struct fs_struct {
  int count;
  unsigned short umask;
  struct inode * root, * pwd;
};
```

gendisk

See include/linux/genhd.h

The `gendisk` data structure holds information about a hard disk. They are used during initialization when the disks are found and then probed for partitions.

```
struct hd_struct {
    long start_sect;
    long nr_sects;
};

struct gendisk {
    int major;               /* major number of driver */
    const char *major_name;  /* name of major driver */
    int minor_shift;         /* number of times minor is shifted to
                                get real minor */
    int max_p;               /* maximum partitions per device */
    int max_nr;              /* maximum number of real devices */

    void (*init)(struct gendisk *);
                             /* Initialization called before we
```

```
                                   do our thing */
        struct hd_struct *part;  /* partition table */
        int *sizes;              /* device size in blocks, copied to
                                    blk_size[] */
        int nr_real;             /* number of real devices */

        void *real_devices;      /* internal use */
        struct gendisk *next;
    };
```

inode

See include/linux/fs.h

The VFS inode data structure holds information about a file or directory on disk.

```
    struct inode {
        kdev_t                       i_dev;
        unsigned long                i_ino;
        umode_t                      i_mode;
        nlink_t                      i_nlink;
        uid_t                        i_uid;
        gid_t                        i_gid;
        kdev_t                       i_rdev;
        off_t                        i_size;
        time_t                       i_atime;
        time_t                       i_mtime;
        time_t                       i_ctime;
        unsigned long                i_blksize;
        unsigned long                i_blocks;
        unsigned long                i_version;
        unsigned long                i_nrpages;
        struct semaphore             i_sem;
        struct inode_operations      *i_op;
        struct super_block           *i_sb;
        struct wait_queue            *i_wait;
        struct file_lock             *i_flock;
        struct vm_area_struct        *i_mmap;
        struct page                  *i_pages;
        struct dquot                 *i_dquot[MAXQUOTAS];
        struct inode                 *i_next, *i_prev;
        struct inode                 *i_hash_next, *i_hash_prev;
        struct inode                 *i_bound_to, *i_bound_by;
        struct inode                 *i_mount;
        unsigned short               i_count;
        unsigned short               i_flags;
        unsigned char                i_lock;
        unsigned char                i_dirt;
        unsigned char                i_pipe;
        unsigned char                i_sock;
        unsigned char                i_seek;
        unsigned char                i_update;
        unsigned short               i_writecount;
        union {
            struct pipe_inode_info   pipe_i;
            struct minix_inode_info  minix_i;
```

```
        struct ext_inode_info    ext_i;
        struct ext2_inode_info   ext2_i;
        struct hpfs_inode_info   hpfs_i;
        struct msdos_inode_info  msdos_i;
        struct umsdos_inode_info umsdos_i;
        struct iso_inode_info    isofs_i;
        struct nfs_inode_info    nfs_i;
        struct xiafs_inode_info  xiafs_i;
        struct sysv_inode_info   sysv_i;
        struct affs_inode_info   affs_i;
        struct ufs_inode_info    ufs_i;
        struct socket            socket_i;
        void                     *generic_ip;
    } u;
};
```

ipc_perm

The `ipc_perm` data structure describes the access permissions of a System V IPC object .

See `include/linux/ipc.h`

```
struct ipc_perm
{
  key_t  key;
  ushort uid;   /* owner euid and egid */
  ushort gid;
  ushort cuid;  /* creator euid and egid */
  ushort cgid;
  ushort mode;  /* access modes see mode flags below */
  ushort seq;   /* sequence number */
};
```

irqaction

The `irqaction` data structure is used to describe the system's interrupt handlers.

See `include/linux/interrupt.h`

```
struct irqaction {
  void (*handler)(int, void *, struct pt_regs *);
  unsigned long flags;
  unsigned long mask;
  const char *name;
  void *dev_id;
  struct irqaction *next;
};
```

linux_binfmt

Each binary file format that Linux understands is represented by a `linux_binfmt` data structure.

See `include/linux/binfmts.h`

```
struct linux_binfmt {
  struct linux_binfmt * next;
  long *use_count;
```

```
  int (*load_binary)(struct linux_binprm *, struct  pt_regs * regs);
  int (*load_shlib)(int fd);
  int (*core_dump)(long signr, struct pt_regs * regs);
};
```

mem_map_t

See include/linux/mm.h

The mem_map_t data structure (also known as page) is used to hold information about each page of physical memory.

```
typedef struct page {
  /* these must be first (free area handling) */
  struct page        *next;
  struct page        *prev;
  struct inode       *inode;
  unsigned long      offset;
  struct page        *next_hash;
  atomic_t           count;
  unsigned           flags;     /* atomic flags, some possibly
                                   updated asynchronously */
  unsigned           dirty:16,
                     age:8;
  struct wait_queue  *wait;
  struct page        *prev_hash;
  struct buffer_head *buffers;
  unsigned long      swap_unlock_entry;
  unsigned long      map_nr;    /* page->map_nr == page - mem_map */
} mem_map_t;
```

mm_struct

See include/linux/sched.h

The mm_struct data structure is used to describe the virtual memory of a task or process.

```
struct mm_struct {
  int count;
  pgd_t * pgd;
  unsigned long context;
  unsigned long start_code, end_code, start_data, end_data;
  unsigned long start_brk, brk, start_stack, start_mmap;
  unsigned long arg_start, arg_end, env_start, env_end;
  unsigned long rss, total_vm, locked_vm;
  unsigned long def_flags;
  struct vm_area_struct * mmap;
  struct vm_area_struct * mmap_avl;
  struct semaphore mmap_sem;
};
```

pci_bus

See include/linux/pci.h

Every PCI bus in the system is represented by a pci_bus data structure.

```
struct pci_bus {
  struct pci_bus  *parent;     /* parent bus this bridge is on */
  struct pci_bus  *children;   /* chain of P2P bridges on this bus */
  struct pci_bus  *next;       /* chain of all PCI buses */

  struct pci_dev  *self;       /* bridge device as seen by parent */
  struct pci_dev  *devices;    /* devices behind this bridge */

  void    *sysdata;            /* hook for sys-specific extension */

  unsigned char  number;       /* bus number */
  unsigned char  primary;      /* number of primary bridge */
  unsigned char  secondary;    /* number of secondary bridge */
  unsigned char  subordinate;  /* max number of subordinate buses */
};
```

pci_dev

Every PCI device in the system, including PCI-PCI and PCI-ISA bridge devices is represented by a `pci_dev` data structure.

See `include/linux/pci.h`

```
/*
 * There is one pci_dev structure for each slot-number/function-number
 * combination:
 */
struct pci_dev {
  struct pci_bus  *bus;      /* bus this device is on */
  struct pci_dev  *sibling;  /* next device on this bus */
  struct pci_dev  *next;     /* chain of all devices */

  void    *sysdata;          /* hook for sys-specific extension */

  unsigned int  devfn;       /* encoded device & function index */
  unsigned short  vendor;
  unsigned short  device;
  unsigned int  class;       /* 3 bytes: (base,sub,prog-if) */
  unsigned int  master : 1;  /* set if device is master capable */
  /*
   * In theory, the irq level can be read from configuration
   * space and all would be fine.  However, old PCI chips don't
   * support these registers and return 0 instead.  For example,
   * the Vision864-P rev 0 chip can uses INTA, but returns 0 in
   * the interrupt line and pin registers.  pci_init()
   * initializes this field with the value at PCI_INTERRUPT_LINE
   * and it is the job of pcibios_fixup() to change it if
   * necessary.  The field must not be 0 unless the device
   * cannot generate interrupts at all.
   */
  unsigned char  irq;        /* irq generated by this device */
};
```

request

See include/linux/blkdev.h

`request` data structures are used to make requests to the block devices in the system. The requests are always to read or write blocks of data to or from the buffer cache.

```
struct request {
    volatile int rq_status;
#define RQ_INACTIVE            (-1)
#define RQ_ACTIVE              1
#define RQ_SCSI_BUSY           0xffff
#define RQ_SCSI_DONE           0xfffe
#define RQ_SCSI_DISCONNECTING  0xffe0

    kdev_t rq_dev;
    int cmd;        /* READ or WRITE */
    int errors;
    unsigned long sector;
    unsigned long nr_sectors;
    unsigned long current_nr_sectors;
    char * buffer;
    struct semaphore * sem;
    struct buffer_head * bh;
    struct buffer_head * bhtail;
    struct request * next;
};
```

rtable

See include/net/route.h

Each `rtable` data structure holds information about the route to take in order to send packets to an IP host. `rtable` data structures are used within the IP route cache.

```
struct rtable
{
    struct rtable     *rt_next;
    __u32             rt_dst;
    __u32             rt_src;
    __u32             rt_gateway;
    atomic_t          rt_refcnt;
    atomic_t          rt_use;
    unsigned long     rt_window;
    atomic_t          rt_lastuse;
    struct hh_cache   *rt_hh;
    struct device     *rt_dev;
    unsigned short    rt_flags;
    unsigned short    rt_mtu;
    unsigned short    rt_irtt;
    unsigned char     rt_tos;
};
```

semaphore

See include/asm/semaphore.h

Semaphores are used to protect critical data structures and regions of code. y

```
struct semaphore {
    int count;
    int waking;
    int lock ;                /* to make waking testing atomic */
    struct wait_queue *wait;
};
```

sk_buff

The `sk_buff` data structure is used to describe network data as it moves between the layers of protocol.

See include/linux/skbuff.h

```
struct sk_buff
{
  struct sk_buff      *next;       /* Next buffer in list                    */
  struct sk_buff      *prev;       /* Previous buffer in list                */
  struct sk_buff_head *list;       /* List we are on                         */
  int                 magic_debug_cookie;
  struct sk_buff      *link3;      /* Link for IP protocol level buffer chains */
  struct sock         *sk;         /* Socket we are owned by                 */
  unsigned long       when;        /* used to compute rtt's                  */
  struct timeval      stamp;       /* Time we arrived                        */
  struct device       *dev;        /* Device we arrived on/are leaving by    */
  union
  {
      struct tcphdr   *th;
      struct ethhdr   *eth;
      struct iphdr    *iph;
      struct udphdr   *uh;
      unsigned char   *raw;
      /* for passing file handles in a unix domain socket */
      void            *filp;
  } h;

  union
  {
      /* As yet incomplete physical layer views */
      unsigned char   *raw;
      struct ethhdr   *ethernet;
  } mac;

  struct iphdr        *ip_hdr;     /* For IPPROTO_RAW                        */
  unsigned long       len;         /* Length of actual data                  */
  unsigned long       csum;        /* Checksum                               */
  __u32               saddr;       /* IP source address                      */
  __u32               daddr;       /* IP target address                      */
  __u32               raddr;       /* IP next hop address                    */
  __u32               seq;         /* TCP sequence number                    */
  __u32               end_seq;     /* seq [+ fin] [+ syn] + datalen          */
  __u32               ack_seq;     /* TCP ack sequence number                */
  unsigned char       proto_priv[16];
  volatile char       acked,       /* Are we acked ?                         */
                      used,        /* Are we in use ?                        */
                      free,        /* How to free this buffer                */
```

```
                    arp;         /* Has IP/ARP resolution finished       */
  unsigned char     tries,       /* Times tried                          */
                    lock,        /* Are we locked ?                      */
                    localroute,  /* Local routing asserted for this frame */
                    pkt_type,    /* Packet class                         */
                    pkt_bridged, /* Tracker for bridging                 */
                    ip_summed;   /* Driver fed us an IP checksum         */
#define PACKET_HOST         0      /* To us                                */
#define PACKET_BROADCAST    1      /* To all                               */
#define PACKET_MULTICAST    2      /* To group                             */
#define PACKET_OTHERHOST    3      /* To someone else                      */
  unsigned short    users;       /* User count - see datagram.c,tcp.c    */
  unsigned short    protocol;    /* Packet protocol from driver.         */
  unsigned int      truesize;    /* Buffer size                          */
  atomic_t          count;       /* reference count                      */
  struct sk_buff    *data_skb;   /* Link to the actual data skb          */
  unsigned char     *head;       /* Head of buffer                       */
  unsigned char     *data;       /* Data head pointer                    */
  unsigned char     *tail;       /* Tail pointer                         */
  unsigned char     *end;        /* End pointer                          */
  void              (*destructor)(struct sk_buff *); /* Destruct function */
  __u16             redirport;   /* Redirect port                        */
};
```

sock

Each sock data structure holds protocol specific information about a BSD socket. For example, for an INET (Internet Address Domain) socket this data structure would hold all of the TCP/IP and UDP/IP specific information.

See include/linux/net.h

```
struct sock
{
    /* This must be first. */
    struct sock            *sklist_next;
    struct sock            *sklist_prev;

    struct options         *opt;
    atomic_t               wmem_alloc;
    atomic_t               rmem_alloc;
    unsigned long          allocation;       /* Allocation mode */
    __u32                  write_seq;
    __u32                  sent_seq;
    __u32                  acked_seq;
    __u32                  copied_seq;
    __u32                  rcv_ack_seq;
    unsigned short         rcv_ack_cnt;      /* count of same ack */
    __u32                  window_seq;
    __u32                  fin_seq;
    __u32                  urg_seq;
    __u32                  urg_data;
    __u32                  syn_seq;
    int                    users;            /* user count */
  /*
   *    Not all are volatile, but some are, so we
```

```
 *     might as well say they all are.
 */
  volatile char           dead,
                          urginline,
                          intr,
                          blog,
                          done,
                          reuse,
                          keepopen,
                          linger,
                          delay_acks,
                          destroy,
                          ack_timed,
                          no_check,
                          zapped,
                          broadcast,
                          nonagle,
                          bsdism;
  unsigned long           lingertime;
  int                     proc;

  struct sock             *next;
  struct sock             **pprev;
  struct sock             *bind_next;
  struct sock             **bind_pprev;
  struct sock             *pair;
  int                     hashent;
  struct sock             *prev;
  struct sk_buff          *volatile send_head;
  struct sk_buff          *volatile send_next;
  struct sk_buff          *volatile send_tail;
  struct sk_buff_head     back_log;
  struct sk_buff          *partial;
  struct timer_list       partial_timer;
  long                    retransmits;
  struct sk_buff_head     write_queue,
                          receive_queue;
  struct proto            *prot;
  struct wait_queue       **sleep;
  __u32                   daddr;
  __u32                   saddr;          /* Sending source */
  __u32                   rcv_saddr;      /* Bound address */
  unsigned short          max_unacked;
  unsigned short          window;
  __u32                   lastwin_seq;    /* sequence number when we last
                                             updated the window we offer */
  __u32                   high_seq;       /* sequence number when we did
                                             current fast retransmit */
  volatile unsigned long  ato;            /* ack timeout */
  volatile unsigned long  lrcvtime;       /* jiffies at last data rcv */
  volatile unsigned long  idletime;       /* jiffies at last rcv */
  unsigned int            bytes_rcv;
/*
 *    mss is min(mtu, max_window)
```

```
 */
    unsigned short          mtu;              /* mss negotiated in the syn's */
    volatile unsigned short mss;              /* current eff. mss - can change */
    volatile unsigned short user_mss;         /* mss requested by user in ioctl */
    volatile unsigned short max_window;
    unsigned long           window_clamp;
    unsigned int            ssthresh;
    unsigned short          num;
    volatile unsigned short cong_window;
    volatile unsigned short cong_count;
    volatile unsigned short packets_out;
    volatile unsigned short shutdown;
    volatile unsigned long  rtt;
    volatile unsigned long  mdev;
    volatile unsigned long  rto;

    volatile unsigned short backoff;
    int                     err, err_soft;    /* Soft holds errors that don't
                                                 cause failure but are the cause
                                                 of a persistent failure not
                                                 just 'timed out' */
    unsigned char           protocol;
    volatile unsigned char  state;
    unsigned char           ack_backlog;
    unsigned char           max_ack_backlog;
    unsigned char           priority;
    unsigned char           debug;
    int                     rcvbuf;
    int                     sndbuf;
    unsigned short          type;
    unsigned char           localroute;       /* Route locally only */
/*
 *    This is where all the private (optional) areas that don't
 *    overlap will eventually live.
 */
    union
    {
          struct unix_opt   af_unix;
#if defined(CONFIG_ATALK) || defined(CONFIG_ATALK_MODULE)
        struct atalk_sock   af_at;
#endif
#if defined(CONFIG_IPX) || defined(CONFIG_IPX_MODULE)
        struct ipx_opt      af_ipx;
#endif
#ifdef CONFIG_INET
        struct inet_packet_opt  af_packet;
#ifdef CONFIG_NUTCP
        struct tcp_opt      af_tcp;
#endif
#endif
    } protinfo;
/*
 *    IP 'private area'
 */
```

```
    int                     ip_ttl;          /* TTL setting */
    int                     ip_tos;          /* TOS */
    struct tcphdr           dummy_th;
    struct timer_list       keepalive_timer;  /* TCP keepalive hack */
    struct timer_list       retransmit_timer; /* TCP retransmit timer */
    struct timer_list       delack_timer;     /* TCP delayed ack timer */
    int                     ip_xmit_timeout;  /* Why the timeout is running */
    struct rtable           *ip_route_cache;  /* Cached output route */
    unsigned char           ip_hdrincl;       /* Include headers ? */
#ifdef CONFIG_IP_MULTICAST
    int                     ip_mc_ttl;        /* Multicasting TTL */
    int                     ip_mc_loop;       /* Loopback */
    char                    ip_mc_name[MAX_ADDR_LEN]; /* Multicast device name */
    struct ip_mc_socklist   *ip_mc_list;      /* Group array */
#endif

/*
 *    This part is used for the timeout functions (timer.c).
 */
    int                      timeout;         /* What are we waiting for? */
    struct timer_list        timer;           /* This is the TIME_WAIT/receive
                                               * timer when we are doing IP
                                               */
    struct timeval           stamp;
 /*
  *    Identd
  */
    struct socket            *socket;
  /*
   *    Callbacks
   */
    void                     (*state_change)(struct sock *sk);
    void                     (*data_ready)(struct sock *sk,int bytes);
    void                     (*write_space)(struct sock *sk);
    void                     (*error_report)(struct sock *sk);

};
```

socket

Each `socket` data structure holds information about a BSD socket. It does not exist independently; it is, instead, part of the VFS `inode` data structure.

See include/linux/net.h

```
struct socket {
  short               type;         /* SOCK_STREAM, ...             */
  socket_state        state;
  long                flags;
  struct proto_ops    *ops;         /* protocols do most everything */
  void                *data;        /* protocol data                */
  struct socket       *conn;        /* server socket connected to   */
  struct socket       *iconn;       /* incomplete client conn.s     */
  struct socket       *next;
  struct wait_queue   **wait;       /* ptr to place to wait on      */
  struct inode        *inode;
```

```
  struct fasync_struct *fasync_list; /* Asynchronous wake up list    */
  struct file          *file;        /* File back pointer for gc     */
};
```

task_struct

See include/linux/sched.h

Each task_struct data structure describes a process or task in the system.

```
struct task_struct {
/* these are hardcoded - don't touch */
  volatile long       state;          /* -1 unrunnable, 0 runnable, >0 stopped */
  long                counter;
  long                priority;
  unsigned            long signal;
  unsigned            long blocked;   /* bitmap of masked signals */
  unsigned            long flags;     /* per process flags, defined below */
  int errno;
  long                debugreg[8];    /* Hardware debugging registers */
  struct exec_domain  *exec_domain;
/* various fields */
  struct linux_binfmt  *binfmt;
  struct task_struct  *next_task, *prev_task;
  struct task_struct  *next_run,  *prev_run;
  unsigned long       saved_kernel_stack;
  unsigned long       kernel_stack_page;
  int                 exit_code, exit_signal;
  /* ??? */
  unsigned long       personality;
  int                 dumpable:1;
  int                 did_exec:1;
  int                 pid;
  int                 pgrp;
  int                 tty_old_pgrp;
  int                 session;
  /* boolean value for session group leader */
  int                 leader;
  int                 groups[NGROUPS];
  /*
   * pointers to (original) parent process, youngest child, younger sibling,
   * older sibling, respectively.  (p->father can be replaced with
   * p->p_pptr->pid)
   */
  struct task_struct  *p_opptr, *p_pptr, *p_cptr,
                      *p_ysptr, *p_osptr;
  struct wait_queue   *wait_chldexit;
  unsigned short      uid,euid,suid,fsuid;
  unsigned short      gid,egid,sgid,fsgid;
  unsigned long       timeout, policy, rt_priority;
  unsigned long       it_real_value, it_prof_value, it_virt_value;
  unsigned long       it_real_incr, it_prof_incr, it_virt_incr;
  struct timer_list   real_timer;
  long                utime, stime, cutime, cstime, start_time;
/* mm fault and swap info: this can arguably be seen as either
   mm-specific or thread-specific */
```

```
  unsigned long       min_flt, maj_flt, nswap, cmin_flt, cmaj_flt, cnswap;
  int swappable:1;
  unsigned long       swap_address;
  unsigned long       old_maj_flt;    /* old value of maj_flt */
  unsigned long       dec_flt;        /* page fault count of the last time */
  unsigned long       swap_cnt;       /* number of pages to swap on next pass */
/* limits */
  struct rlimit       rlim[RLIM_NLIMITS];
  unsigned short      used_math;
  char                comm[16];
/* file system info */
  int                 link_count;
  struct tty_struct   *tty;           /* NULL if no tty */
/* ipc stuff */
  struct sem_undo     *semundo;
  struct sem_queue    *semsleeping;
/* ldt for this task - used by Wine.  If NULL, default_ldt is used */
  struct desc_struct *ldt;
/* tss for this task */
  struct thread_struct tss;
/* filesystem information */
  struct fs_struct    *fs;
/* open file information */
  struct files_struct  *files;
/* memory management info */
  struct mm_struct    *mm;
/* signal handlers */
  struct signal_struct *sig;
#ifdef __SMP__
  int                 processor;
  int                 last_processor;
  int                 lock_depth;     /* Lock depth.
                                         We can context switch in and out
                                         of holding a syscall kernel lock... */
#endif
};
```

timer_list

See include/linux/timer.h

`timer_list` data structure's are used to implement real time timers for processes.

```
struct timer_list {
  struct timer_list *next;
  struct timer_list *prev;
  unsigned long expires;
  unsigned long data;
  void (*function)(unsigned long);
};
```

tq_struct

Each task queue (`tq_struct`) data structure holds information about work that has been queued. This is usually a task needed by a device driver but which does not

See include/linux/tqueue.h

have to be done immediately.

```
struct tq_struct {
    struct tq_struct *next;   /* linked list of active bh's */
    int sync;                 /* must be initialized to zero */
    void (*routine)(void *);  /* function to call */
    void *data;               /* argument to function */
};
```

vm_area_struct

See include/linux/mm.h

Each vm_area_struct data structure describes an area of virtual memory for a process.

```
struct vm_area_struct {
  struct mm_struct * vm_mm;  /* VM area parameters */
  unsigned long vm_start;
  unsigned long vm_end;
  pgprot_t vm_page_prot;
  unsigned short vm_flags;
/* AVL tree of VM areas per task, sorted by address */
  short vm_avl_height;
  struct vm_area_struct * vm_avl_left;
  struct vm_area_struct * vm_avl_right;
/* linked list of VM areas per task, sorted by address */
  struct vm_area_struct * vm_next;
/* for areas with inode, the circular list inode->i_mmap */
/* for shm areas, the circular list of attaches */
/* otherwise unused */
  struct vm_area_struct * vm_next_share;
  struct vm_area_struct * vm_prev_share;
/* more */
  struct vm_operations_struct * vm_ops;
  unsigned long vm_offset;
  struct inode * vm_inode;
  unsigned long vm_pte;      /* shared mem */
};
```

Appendix B

Useful Web and FTP Sites

The following World Wide Web and ftp sites are useful:

http://www.azstarnet.com/~axplinux This is David Mosberger-Tang's Alpha AXP Linux web site and it is the place to go for all of the Alpha AXP HOW-TOs. It also has a large number of pointers to Linux and Alpha AXP specific information such as CPU data sheets.

http://www.redhat.com/ Red Hat's web site. This has a lot of useful pointers.

ftp://sunsite.unc.edu This is the major site for a lot of free software. The Linux specific software is held in *pub/Linux*.

http://www.intel.com Intel's web site and a good place to look for Intel chip information.

http://www.ssc.com/lj/index.html The Linux Journal is a very good Linux magazine and well worth the yearly subscription for its excellent articles.

http://www.blackdown.org/java-linux.html This is the primary site for information on Java on Linux.

ftp://tsx-11.mit.edu/~ftp/pub/linux MIT's Linux ftp site.

ftp://ftp.cs.helsinki.fi/pub/Software/Linux/Kernel Linus's kernel sources.

http://www.linux.org.uk The UK Linux User Group.

http://sunsite.unc.edu/mdw/linux.html Home page for the Linux Documentation Project,

http://www.digital.com Digital Equipment Corporation's main web page.

http://altavista.digital.com DIGITAL's Altavista search engine. A very good place to search for information within the web and news groups.

http://www.linuxhq.com The Linux HQ web site holds up to date official and unoffical patches as well as advice and web pointers that help you get the best set of kernel sources possible for your system.

http://www.amd.com The AMD web site.

http://www.cyrix.com Cyrix's web site.

http://www.arm.com ARM's web site.

Appendix C

Linux Documentation Project Manifesto

This is the Linux Documentation Project "Manifesto"

Last Revision 21 September 1998, by Michael K. Johnson

This file describes the goals and current status of the Linux Documentation Project, including names of projects, volunteers, FTP sites, and so on.

C.1 Overview

The Linux Documentation Project is working on developing good, reliable docs for the Linux operating system. The overall goal of the LDP is to collaborate in taking care of all of the issues of Linux documentation, ranging from online docs (man pages, texinfo docs, and so on) to printed manuals covering topics such as installing, using, and running Linux. The LDP is essentially a loose team of volunteers with little central organization; anyone who is interested in helping is welcome to join in the effort. We feel that working together and agreeing on the direction and scope of Linux documentation is the best way to go, to reduce problems with conflicting efforts—two people writing two books on the same aspect of Linux wastes someone's time along the way.

The LDP is set out to produce the canonical set of Linux online and printed documentation. Because our docs will be freely available (like software licensed under the terms of the GNU GPL) and distributed on the net, we are able to easily update the documentation to stay on top of the many changes in the Linux world. If you are interested in publishing any of the LDP works, see the section "Publishing LDP Manuals", below.

C.2 Getting Involved

Send mail to `linux-howto@metalab.unc.edu`

Of course, you'll also need to get in touch with the coordinator of whatever LDP projects you're interested in working on; see the next section.

C.3 Current Projects

For a list of current projects, see the LDP Homepage at `http://sunsite.unc.edu/LDP/ldp.html`. The best way to get involved with one of these projects is to pick up the current version of the manual and send revisions, editions, or suggestions to the coordinator. You probably want to coordinate with the author before sending revisions so that you know you are working together.

C.4 FTP sites for LDP works

LDP works can be found on sunsite.unc.edu in the directory /pub/Linux/docs. LDP manuals are found in /pub/Linux/docs/LDP, HOWTOs and other documentation found in /pub/Linux/docs/HOWTO.

C.5 Documentation Conventions

Here are the conventions that are currently used by LDP manuals. If you are interested in writing another manual using different conventions, please let us know of your plans first.

The *man pages* – the Unix standard for online manuals – are created with the Unix standard nroff man (or BSD mdoc) macros.

The *guides* – full books produced by the LDP – have historically been done in LaTeX, as their primary goal has been to *printed* documentation. However, guide authors have been moving towards SGML with the DocBook DTD, because it allows them to create more different kinds of output, both printed and on-line. If you use LaTeX, we have a style file you can use to keep your printed look consistent with other LDP documents, and we suggest that you use it.

The *HOWTO* documents are all required to be in SGML format. Currently, they use the *linuxdoc* DTD, which is quite simple. There is a move afoot to switch to the DocBook DTD over time.

LDP documents must be freely redistributable without fees paid to the authors. It is not required that the text be modifiable, but it is encouraged. You can come up with your own license terms that satisfy this constraint, or you can use a previously prepared license. The LDP provides a boilerplate license that you can use, some people like to use the GPL, and others write their own.

The copyright for each manual should be in the name of the head writer or coordinator for the project. "The Linux Documentation Project" isn't a formal entity and shouldn't be used to copyright the docs.

C.6 Copyright and License

Here is a "boilerplate" license you may apply to your work. It has not been reviewed by a lawyer; feel free to have your own lawyer review it (or your modification of it) for its applicability to your own desires. Remember that in order for your document to be part of the LDP, you must allow unlimited reproduction and distribution without fee.

This manual may be reproduced and distributed in whole or in part, without fee, subject to the following conditions:

- The copyright notice above and this permission notice must be preserved complete on all complete or partial copies.
- Any translation or derived work must be approved by the author in writing before distribution.
- If you distribute this work in part, instructions for obtaining the complete version of this manual must be included, and a means for obtaining a complete version provided.
- Small portions may be reproduced as illustrations for reviews or quotes in other works without this permission notice if proper citation is given.

Exceptions to these rules may be granted for academic purposes: Write to the author and ask. These restrictions are here to protect us as authors, not to restrict you as learners and educators.

All source code in this document is placed under the GNU General Public License, available via anonymous FTP from `prep.ai.mit.edu:/pub/gnu/COPYING`.

C.7 Publishing LDP Manuals

If you're a publishing company interested in distributing any of the LDP manuals, read on.

By the license requirements given previously, anyone is allowed to publish and distribute verbatim copies of the Linux Documentation Project manuals. You don't need our explicit permission for this. However, if you would like to distribute a translation or derivative work based on any of the LDP manuals, you may need to obtain permission from the author, in writing, before doing so, if the license requires that.

You may, of course, sell the LDP manuals for profit. We encourage you to do so. Keep in mind, however, that because the LDP manuals are freely distributable, anyone may photocopy or distribute printed copies free of charge, if they wish to do so.

We do not require to be paid royalties for any profit earned from selling LDP manuals. However, we would like to suggest that if you do sell LDP manuals for profit, that you either offer the author royalties, or donate a portion of your earnings to the author, the LDP as a whole, or to the Linux development community. You may also wish to send one or more free copies of the LDP manuals that you are distributing to the authors. Your show of support for the LDP and the Linux community will be very much appreciated.

We would like to be informed of any plans to publish or distribute LDP manuals, just so we know how they're becoming available. If you are publishing or planning to publish any LDP manuals, please send mail to `ldp-l@linux.org.au`. It's nice to know who's doing what.

We encourage Linux software distributors to distribute the LDP manuals (such as the Installation and Getting Started Guide) with their software. The LDP manuals are intended to be used as the "official" Linux documentation, and we are glad to see mail-order distributors bundling the LDP manuals with the software. As the LDP manuals mature, hopefully they will fulfill this goal more and more adequately.

Appendix D

The GNU General Public License

Printed below is the GNU General Public License (the *GPL* or *copyleft*), under which Linux is licensed. It is reproduced here to clear up some of the confusion about Linux's copyright status—Linux is *not* shareware, and it is *not* in the public domain. The bulk of the Linux kernel is copyright ©1993 by Linus Torvalds, and other software and parts of the kernel are copyrighted by their authors. Thus, Linux *is* copyrighted, however, you may redistribute it under the terms of the GPL printed below.

GNU GENERAL PUBLIC LICENSE
Version 2, June 1991

D.1 Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software–to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

D.2 Terms and Conditions for Copying, Distribution, and Modification

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the

absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

   Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

   In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this

License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED

IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

D.3 Appendix: How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the “copyright” line and a pointer to where the full notice is found.

> ⟨*one line to give the program's name and a brief idea of what it does.*⟩
> Copyright ©19yy ⟨*name of author*⟩
>
> This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.
>
> This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
>
> You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author Gnomovision
comes with ABSOLUTELY NO WARRANTY; for details type 'show w'.  This
is free software, and you are welcome to redistribute it under certain
conditions; type 'show c' for details.
```

The hypothetical commands 'show w' and 'show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than 'show w' and 'show c'; they could even be mouse-clicks or menu items–whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

> Yoyodyne, Inc., hereby disclaims all copyright interest in the program 'Gnomovision' (which makes passes at compilers) written by James Hacker.
>
> ⟨*signature of Ty Coon*⟩, 1 April 1989 Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Glossary

Argument Functions and routines are passed arguments to process.

ARP Address Resolution Protocol. Used to translate IP addresses into physical hardware addresses.

Ascii American Standard Code for Information Interchange. Each letter of the alphabet is represented by an 8 bit code. Ascii is most often used to store written characters.

Bit A single bit of data that represents either 1 or 0 (on or off).

Bottom Half Handler Handlers for work queued within the kernel.

Byte 8 bits of data,

C A high level programming language. Most of the Linux kernel is written in C.

CISC Complex Instruction Set Computer. The opposite of *RISC*, a processor which supports a large number of often complex assembly instructions. The X86 architecture is a CISC architecture.

CPU Central Processing Unit. The main engine of the computer, see also *microprocessor* and *processor*.

Data Structure This is a set of data in memory comprised of fields,

Device Driver The software controlling a particular device, for example the NCR 810 device driver controls the NCR 810 SCSI device.

DMA Direct Memory Access.

ELF Executable and Linkable Format. This object file format designed by the Unix System Laboratories is now firmly established as the most commonly used format in Linux.

EIDE Extended IDE.

Executable image A structured file containing machine instructions and data. This file can be loaded into a process's virtual memory and executed. See also *program*.

Function A piece of software that performs an action. For example, returning the bigger of two numbers.

IDE Integrated Disk Electronics.

Image See *executable image*.

IP Internet Protocol.

IPC Interprocess Communiction.

Interface A standard way of calling routines and passing data structures. For example, the interface between two layers of code might be expressed in terms of routines that pass and return a particular data structure. Linux's VFS is a good example of an interface.

IRQ Interrupt Request Queue.

ISA Industry Standard Architecture. This is a standard, although now rather dated, data bus interface for system components such as floppy disk drivers.

Kernel Module A dynamically loaded kernel function such as a filesystem or a device driver.

Kilobyte A thousand bytes of data, often written as **K**byte,

Megabyte A million bytes of data, often written as **M**byte,

Microprocessor A very integrated *CPU*. Most modern CPUs are *Microprocessors*.

Module A file containing CPU instructions in the form of either assembly language instructions or a high level language like C.

Object file A file containing machine code and data that has not yet been linked with other object files or libraries to become an *executable image*.

Page Physical memory is divided up into equal sized pages.

Pointer A location in memory that contains the address of another location in memory,

Process This is an entity which can execute *programs*. A process could be thought of as a *program* in action.

Processor Short for Microprocessor, equivalent to *CPU*.

PCI Peripheral Component Interconnect. A standard describing how the peripheral components of a computer system may be connected together.

Peripheral An intelligent processor that does work on behalf of the system's CPU. For example, an IDE controller chip,

Program A coherent set of CPU instructions that performs a task, such as printing "hello world". See also *executable image*.

Protocol A protocol is a networking *language* used to transfer application data between two cooperating processes or network layers.

Register A location within a chip, used to store information or instructions.

Register File The set of registers in a processor.

RISC Reduced Instruction Set Computer. The opposite of *CISC*, that is a processor with a small number of assembly instructions, each of which performs simple operations. The ARM and Alpha processors are both RISC architectures.

Routine Similar to a function except that, strictly speaking, routines do not return values.

SCSI Small Computer Systems Interface.

Shell This is a program which acts as an interface between the operating system and a human user. Also called a *command shell*, the most commonly used shell in Linux is the bash shell.

SMP Symmetrical multiprocessing. Systems with more than one processor which fairly share the work amongst those processors.

Socket A socket represents one end of a network connection, Linux supports the BSD Socket interface.

Software CPU instructions (both assembler and high level languages like C) and data. Mostly interchangable with *Program*.

System V A variant of Unix™ produced in 1983, which included, amongst other things, *System V IPC mechanisms*.

TCP Transmission Control Protocol.

Task Queue A mechanism for deferring work in the Linux kernel.

UDP User Datagram Protocol.

Virtual memory A hardware and software mechanism for making the physical memory in a system appear larger than it actually is.

Bibliography

[1] Richard L. Sites. *Alpha Architecture Reference Manual* Digital Press

[2] Matt Welsh and Lar Kaufman. *Running Linux* O'Reilly & Associates, Inc, ISBN 1-56592-100-3

[3] PCI Special Interest Group *PCI Local Bus Specification*

[4] PCI Special Interest Group *PCI BIOS ROM Specification*

[5] PCI Special Interest Group *PCI to PCI Bridge Architecture Specification*

[6] Intel *Peripheral Components* Intel 296467, ISBN 1-55512-207-8

[7] Brian W. Kernighan and Dennis M. Richie *The C Programming Language* Prentice Hall, ISBN 0-13-110362-8

[8] Steven Levy *Hackers* Penguin, ISBN 0-14-023269-9

[9] Intel *Intel486 Processor Family: Programmer's Reference Manual* Intel

[10] Comer D. E. *Interworking with TCP/IP, Volume 1 - Principles, Protocols and Architecture* Prentice Hall International Inc

[11] David Jagger *ARM Architectural Reference Manual* Prentice Hall, ISBN 0-13-736299-4

Index