Kernel Projects for Linux

Web Version of First Edition

Gary Nutt
University of Colorado

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the author was aware of the claim, the designations have been printed in initial caps or in all caps.

Creative Commons
Attribution-Noncommercial-No Derivative Work 3.0 License
You are free to share (to copy, distribute and transmit the work), under the conditions that (1) you must attribute the work to Gary Nutt (but not in any way that suggests that Gary Nutt endorses your use of the work), (2) you may not use this work for commercial purposes, and (3) you may not alter, transform, or build upon this work.

Preface to the First Edition

To the Student

Experience has shown that the best way to learn the details of how an operating system (OS) works is to experiment with it – to read, modify, and enhance the code. However OS software, by its nature, must be carefully constructed. This is because it directly controls the hardware used by all the processes and threads that execute on it. As a result, experimenting with OS code can be difficult, since an experimental version of the OS may disable the test machine. This laboratory manual provides you with a learning path for studying the Linux kernel with as little risk as possible. By learning about the Linux kernel this way, you will develop a technique by which you can learn and experiment with other kernels as well. Consider this learning path to be a graduated set of exercises. First you will learn to inspect various aspects of the OS internal state without changing any code. Second, you will extend the OS by writing new code to read (but not write) kernel data structures. Third, you will reimplement existing data structures. Finally, you will design and add your own functions and data structures to the Linux kernel.

The Linux kernel is written in the C programming language. Therefore you need to relatively proficient in using C before you can study the kernel. If you know C++, you will not have difficulty understanding the source code, though when you add or change parts of the kernel code you will not be able to use objects.

This manual is designed as a companion to a general OS textbook. It consists of two parts. Part 1 offers an overview of the Linux design. If you are using this manual at the beginning of your first OS course, you might discover the Part 1 discusses several topics that are new to you. However, work your way through it to get a feeling for how Linux is built. It gives the "big picture," but not many details. Then go back to Part 1 as needed as you work the exercises.

Part 2 consists of a dozen laboratory exercises that help you to learn to use Linux Each exercise is a sself-contained unit consisting of the following sections:

- Introduction
- Problem Statement
- Attacking the Problem

The exercises link the general concepts and Linux details. Each begins with an introduction that explains Linux concepts and details relevant to the exercise. The introduction explains how the generic concepts that you will have learned in lecture and textbooks are realized in Linux. The next part of the exercise presents the problem on which you will be working. It includes detailed Linux-specific information that you will need to solve the problem. Sometimes, a quick review of the pertinent parts of Part 1 will help you to frame the work of the exercises before you dive into the details.

Your school's laboratory probably will have already been set up as a Linux lab. For you to solve most of the exercises in this manual, the laboratory administrator will provide you with a copy of the Linux source code and superuser permission to create new versions of the OS. Do not abuse your privilege as a superuser! You need this privilege to modify the kernel, but you must not use it for other purposes. This manual includes a

CD-ROM, containing the Linux source code that you can use to install Linux on your own computer.

Good luck on your study of operating systems. I hope this exercise manual is a valuable learning tool in seeing how OS concepts appear in Linux.

To the Instructor

Today, abstraction is the basis of most software that is written – in the classroom or in practice. Students are taught to think of software solutions in terms of objects, components, threads, messages, and so on. This perspective teaches them to leverage the power of the hardware to solve increasingly more complex tasks. In this way, they reduce programming time while reusing lower level abstractions. At the bottom of all these abstractions is the operating system – processes and (and sometimes threads) and resources. Application software and middleware use these OS abstractions to create their own higher level abstractions. These range from accounting packages, spreadsheets, and missile trackers, to windows, databases, objects, components, messages, and continuous media streams.

This trend toward the heavy use of abstraction prompts some to argue that operating systems are no longer worthy of serious study, since they are largely transparent to the application programmers working at higher layers of abstraction. However the OS is still fundamental because its design and implementation are the basis of the design and implementation of all the other abstractions. Programmers will always be able to write better middleware and application programs if they understand how operating systems work. Moreover, the need for people who understand basic OS technology remains, whether they are to write drivers for new devices, to create new microkernel servers, or to provide new systems capable of efficiently handling evolving requirements such as continuous media.

Typically an OS instructor has to decide whether an OS course should focus on issues and theory or provide an environment in which students can experiment with OS code. The 1991 (and draft 2001) IEEE/ACM undergraduate course recommendation describes a course with a substantial amount of time spent on issues, but also with a significant laboratory component. Even though the trend is toward courses based on conceptual materials, students and instructors seem to agree that hands-on experience is invaluable in learning about operating systems. Many courses attempt to follow the IEEE/ACM lead by dividing the course into lecture and lab components, with the lecture component focusing on issues and theory and the laboratory component providing some form of hands-on exercises.

The IEEE/ACM recommendation supports the idea that the laboratory component should allow student to learn how to use the OS mechanisms specifically by focusing on the OS application programming interface (API) as the primary mechaism for experimentation. The philosophy behind this approach is that students must learn how to use an OS before they can really understand how to *design* one. This philosophy drives a companion book on programming Windows NT via the Win32 API [Nutt, 1999], and the concepts and issues in [Nutt, 2004].

However, in a recent (late 1998) survey of 78 universities conducted by Addison Wesley, 43 indicated that they teach OS *internals* in the introductory OS course. Of these 43, 26 use a variant of UNIX as the target OS: 13 use Linux, 10 use an unspecified version of UNIX, and 3 use MINIX. Eight said they used some other OS as the subject system (such as Nachos), and the remaining 9 did not say what OS they use. The survey clearly showed that a significant fraction of the community teaches OS internals as a component of the introductory OS class, despite the IEEE/ACM recommendation, and despite the heavy use of conceptual OS textbooks. It also showed that most of these courses use two books: A traditional OS theory book (such as [Silberschatz and Galvin, 1998] or [Nutt, 2004]) and a reference book (such as [Stevens, 1993], [McKusick, et al., 1996], or [Beck, et al, 1998]). Of course, no single-term undergraduate course can possibly cover all the material in both a theory book and a book describing an entire OS. The lack of a good lab manual forces the instructor to have students buy a supplementary book that contains *much* more information than they will have time to learn in a single academic term. Further, the instructor will have to learn all the material in both books, as well as learn the the subject OS, derive a suitable set of exercises, and provide some form of guidance through the OS reference materials so the students can solve the exercises.

This textbook is a laboratory manual of Linux internals exercises. It complements an OS theory book by providing a dozen specific lab exercises on Linux internals that illustrate how theoretical concepts are implemented in Linux. The instructor need not become a "complete" Linux kernel expert or derive a set of exercises (either with full documentation for the exercise or with pointers into appropriate sections in a supplementary reference book). Instead, the instructor, lab assistant, and students can use this manual as a self-contained source of background information and exercises to study how concepts are implemented. Thus the less expensive lab manual replaces a general reference book, while providing focused information for a set of kernel internals exercises. For the student who wants to understand related information that is not required in order to solve the exercise, the background material for exercises provide pointers to reference books (and the literature).

A single-semester OS course consists of 15 weeks of course material. In my experience, many undergraduate students have difficulty doing a substantial programming exercise in less than one and a half to two weeks. This means that my students have been able to complete perhaps six to eight programming assignments in a semester. This manual provides enough exercises to allow you to choose a subset that best suit your students' background and your preferences. Most of the exercises include options that allow you to vary the assignments from term to term (thereby reducing the probability of public solutions from previous terms). As mentioned above, my intention is to release frequent editions; I expect that the new editions will have new exercises that require new solutions.

Also provided is a solution to each exercise. Thus more difficult exercises can be chosen, and as necessary you can distribute parts of the solution that are not published in the manual.

None of these exercises in this manual are as difficult as building a new kernel from scratch. Rather they emphasize having students study the existing Linux kernel by modifying or extending its components. The first ones are easy, and the background material is comprehensive. Later exercises increase in difficulty with decreasing amounts of "hand-holding" information. Exercises 1 and 2 would ordinarily require a week or less to complete, but the last third of the exercises are likely to require a couple of weeks each. If your students need extra practice with C programming, you might carefully consider using Laboratory Exercises 1 and 2 as tutorial mechanisms. This may require that you provide a little extra assistance, especially with the concurrency elements of Laboratory Exercise 2.

Any hands-on study of an OS must commit to a particular version of the OS. This first edition was originally written for Version 2.0.36, then it was updated for Linux Version 2.2.12 before it was published. With the rapid evolution of Linux, Version 2.2.12 may be out of date by the time the book is published.

Acknowledgements

This manual represents years of effort – mine and other people's –learning about Linux. I benefited considerably from the assistance, insight, and contributions of the teaching assistants for Computer Science 3753: Operating Systems at the University of Colorado: Don Lindsay, Sam Siewert, Ann Root, and Jason Casmira. Phil Levis provided interesting and lively discussions of Linux and the exercises. When I first installed Linux on a machine, it worked, but not as well as it would work after Adam Griff polished the installation.

Many of the exercises are derived from projects and exercises in the undergraduate and graduate operating system classes at the University of Colorado. In particular, Exercise 3 was created by Sam Siewert in spring 1996 for Computer Science 3753. Exercise 4 takes some material from another exercise created by Sam Siewert. Exercise 9 comes from a course project that Jason Casmira did in Computer Science 5573 (the graduate OS class) in fall 1998. Exercise 10 was first designed by Don Lindsay in fall 1995, and then refined by Sam Siewert in spring 1996. Exercise 1 also appears in the author's companion OS textbook [Nutt, 2004] and Exercise 2 is an extension of one that also appears in that book. Exercises 11 and 12 are similar to exercises for Windows NT that appear in another one of the author's manuals [Nutt, 1999]; Norman Ramsey created the original Windows NT exercises, and Ann Root introduced them to me.

Many reviewers helped make the manual much better than its original draft. Richard Guy was willing to suffer through the first draft of the manuscript at UCLA. Paul Stelling (UCLA) did a careful reading of the draft, correcting errors and providing insight into good and bad aspects of it. Simon Gray (Ashland University) provided particularly lucid and insightful comments on the exercises. The following also provided helpful comments that greatly improved the manual: John Barr (Ithaca College), David Binger (Centre College), David E. Boddy (Oakland University), Richard Chapman (Auburn University), Sorin Draghici (Wayne State University), Sandeep Gupta (Colorado State University), Mark Holliday (Western Carolina University), Kevin Jeffay (University of North Carolina at Chapel Hill), Joseph J. Pfeiffer (New Mexico State University), Kenneth A. Reek (Rochester Institute of Technology), and Henning Schulzrinne (Columbia University).

The Addison Wesley staff was very helpful in preparing this work. Molly Taylor and Jason Miranda provided wide ranges of assistance in handling reviews and otherwise supporting the early stages of development. Lisa Hogue finished the process, particularly by saving the day by finding a version of the Linux source code that could be distributed with [the print version of] the book. Laura Michaels did her usual diligent job in copy editing the work,[1] Gina Hagen helped with production and Helen Reebenacker was the production editor. Last, but not leas, Maité Suarez-Rivas, the acquisition editor, recognized the need for the book and relentlessly pushed forward to make it happen. All of these people helped to produce the content, but of course, any errors are my responsibility.

Gary Nutt
Boulder, Colorado

[1] Laura's fine copy edits have not yet been incorporated into this preliminary draft of the web version of the book.

Preface to the Web Version of the First Edition

The print edition of this book, published by Addison Wesley, appeared in 2000, ten years prior to the release of this web edition. From a business perspective, Addison Wesley did a huge favor to the community, since anecdotal evidence suggests that the first edition was widely used, even though it was rarely adopted for classroom use and was never financially successful for the publisher. Thank you Addison Wesley!

Early in 2010, Addison Wesly returned the copyright for the book to me. I have decided to make minor changes to adopt the book for web distribution. The text is essentially the same as the first edition of the Addison Wesley print version. Part 1 of the web edition is available under the Creative Commons Attribution-Noncommercial-No Derivative Work 3.0 License for open textbooks. This allows people to read and copy the work, but reserves rights related to selling or deriving new materials from this work.

The Exercises in Part 2 are published on my commercial site, where they will be accessible for a modest fee; contact me at Gary.Nutt@colorado.edu if you wish to access the Exercises. If you are an instructor who wishes to review the material, please send me (Gary.Nutt@colorado.edu) authentication that you are an instructor and let me know that you would like to review the Exercises. If you have adopted the book for a course, send me authentication to receive a solution set for the Exercises.

Gary Nutt
Boulder, Colorado

PART 1: OVERVIEW OF LINUX

Linux is a contemporary, open implementation of UNIX, available at no cost on the Internet. Since its introduction in 1991, it has become a highly respected, robust operating system (OS) implementation, enjoying great success as a platform for studying contemporary operating systems, especially the internal behavior of a kernel. Just as significant, Linux is now used in many corporate information processing systems. Thus studying Linux internals provides valuable career training as well as being an excellent means of educational experimentation. Today several new companies provide industrial-strength support for Linux.

1 The Evolution of Linux

Linux has evolved within the UNIX community. The UNIX operating system was introduced to the public in a classic research paper in 1973 [Ritchie and Thompson, 1973]. UNIX established two new trends in operating system (OS) design: Previous operating systems were huge software packages – typically the largest software package that ran on the computer – and they were designed for a specific hardware platform. By contrast UNIX was intended to be a small, no-frills OS that could be made to run on any small computer. Second, the UNIX philosophy was that the OS *kernel* should provide the minimum essential functionality, and that additional functionality should be added (as user programs) on an as-needed basis. UNIX was revolutionary, and within half a dozen years it had become the preferred OS by programmers in multivendor hardware environments (universities, research laboratories, and system software development organizations).

Even though the UNIX kernel could be ported onto a new hardware platform without redeveloping the entire OS, the source code was owned by AT&T Bell Laboratories. Other organizations could obtain the right to use the source code (for example, to port it onto their preferred computer hardware) by paying a licensing fee to AT&T. By 1980, many universities and research labs had obtained the source code and were busily modifying it to meet their own needs, the most prominent work being done at the University of California at Berkeley under a Defense Advanced Research Projects Agency (DARPA) research contract. Commercial computer vendors had also begun to use the UNIX source code to derive their own version of the UNIX operating system.

By 1985 there were two primary versions of UNIX (running on many different hardware platforms): The main line version from AT&T Bell Labs (called *System V UNIX*), and an alternative version from the University of California at Berkeley (called *BSD UNIX*). The DEC VAX version of BSD UNIX was more specifically referred to as Version 4 BSD UNIX, or simply *4.x BSD UNIX*. Both versions implemented system call interfaces that were recognizable as UNIX, though they differed from one another in several details. By this time, there were also substantial differences in the way the two OS kernels were implemented. The competition between System V and BSD UNIX systems was very active, with programmers swearing by one version or the other. Ultimately the leading commercial proponent of 4.x BSD UNIX (Sun Microcomputers) and AT&T reached a business agreement by which these two leading versions would be merged into a common version of UNIX (the Sun Solaris operating system).

Meanwhile other computer makers pushed for an alternative implementation of the UNIX system call interface. An important event was the formation of a committee to develop a standardized UNIX system call interface – *POSIX.1*.[2] Once POSIX.1 had been established, everyone was free to design and build their own kernel that provided the functionality specified by this application programming interface (API). For example at Carnegie Mellon University, a group of OS researchers led by Richard Rashid developed the *Mach* operating system with a POSIX/UNIX system call interface. Mach was an alternative to the 4.x BSD and System V UNIX kernel implementations. Eventually a version of Mach was used as the basis of the Open Systems Foundation OSF-1 kernel. The technique that was first used to implement the POSIX/UNIX interface in Mach, was to simply incorporate substantial amounts of BSD UNIX code into the Mach kernel. However by the time Version 3 was released, Mach had been redesigned as a *microkernel* with servers; though the Version 3 Mach microkernel contained no licensed source code, the BSD server still implemented the 4.x BSD system call interface using the BSD source code [Tanenbaum, 1995].

On a smaller scale Andrew Tanenbaum designed and implemented a complete version of UNIX named MINIX: "The name MINIX stands for mini-UNIX because it is small enough that even a nonguru can understand how it works." [Tanenbaum, 1987]. MINIX implements a once-popular version of the AT&T UNIX system call interface known as *Version 7* UNIX; this version was the basis of both BSD and System V UNIX. Tanenbaum distributed MINIX as a supplement to his OS textbook, providing a comprehensive discussion of how the kernel was designed and implemented. As he notes: MINIX "… was written a decade after UNIX, and has been structured in a more modular way." [Tanenbaum, 1987]. That is, MINIX is based on a microkernel with servers, as opposed to the monolithic design of BSD and System V UNIX. MINIX was initially quite successful as a pedagogical operating system, but it eventually lost some of its support because of its main feature – its simplicity. MINIX was small enough to study, but not robust enough to use as a practical OS.

In 1991, Linus Torvalds began creating the first version of Linux.[3] He was apparently inspired by the success of MINIX, but he intended for his OS to be more robust and useful than MINIX. Torvalds released his Linux source code to anyone who wanted to use it (by making it freely available over the Internet under the GNU Public License). It quickly caught on as a significant implementation of the POSIX system call interface – Linux was the first version of UNIX for which the source code was completely free. Soon, people all over the world began to contribute their own modifications and enhancements to Torvalds' original code. Today the Linux distribution includes the OS and a spectrum of supplementary tools, written by many different contributors. Besides the original Intel 80386/80486/80586 (also called the "x86" or "i386") implementation, there are now implementations for the Digital Alpha, Sun Sparc, Motorola 68K, MIPS,

---

[2] The system call interface is often simply called "POSIX", though that can be misleading since the POSIX committee developed several different application programming interfaces – APIs – and only the first of these standards addresses the kernel system call interface. In this book we consider only POSIX.1, so we will often use the more popular, but less accurate, designation of POSIX to refer to the POSIX.1 system call interface.

[3] The `comp.os.minix` newsgroup had a few postings from Torvalds in early 1991 in which he let the world know that he was working on a public implementation of POSIX.

and PowerPC. By 1996 – five years after it was created – Linux had become a significant OS. By 1997 it became an important part of the commercial OS scene, while continuing its first role as a free implementation of the UNIX interface.

Following the UNIX philosophy, Linux is actually the nucleus or "kernel" of an operating system rather than the complete OS. The UNIX kernel was introduced to the world as a minimal operating system that implemented the bare necessities, leaving the software to provide the desired bells and whistles to libraries and other user mode software. The early UNIX kernel followed the principle of "small is beautiful." In the intervening quarter of a century, this minimal kernel has continually had features added to it; as a contemporary UNIX kernel can no longer be thought of as a minimal set of OS functionality.

Linux started with a clean slate in 1991; it has attempted to return to the "small is beautiful" philosophy of early UNIX while still implementing the POSIX interface. However, since it is freely available, anyone can add features to the kernel, so it is susceptible to the same "creeping featurism" that caused the UNIX kernel to grow large. The Linux kernel is still relatively small … but growing very rapidly. The Version 2.2 Linux kernel is larger and more complex than the Version 2.0 kernel (which is larger and more complex than Version 1).

This book is about the design and implementation of the Version 2.2 Linux kernel – specifically, the exercises and explanation were taken from the latest stable release of Linux in early 2000, Version 2.2.12. Part 1 of this book describes the overall organization of the kernel. The lab exercises focus on different aspects of the kernel, so each exercise provides considerably more detail about the part of the kernel that is relevant to the exercise.

2 General Kernel Responsibilities

The UNIX family of operating systems divide the functionality into two classes: The kernel executes with the CPU in supervisor mode (see Sections 2.2.2 and 3.2 below) and all other OS components execute with the CPU in user mode. In this laboratory manual, you will focus on the Linux kernel. In general, the kernel is responsible for abstracting and managing a machine's hardware resources, and for managing the sharing of those resources among executing programs. Since Linux implements a UNIX style interface, the general definition of the resource abstractions and sharing models is already defined. The kernel must support processes, files, and other resources so that they can be managed with the traditional UNIX system calls. Linux differs from other systems that implement a UNIX interface in the data structures and algorithms it uses to meet those responsibilities.

2.1 Resource Abstraction

Resource abstraction refers to the idea of creating software (usually) to simplify the operations that must be applied to the hardware to cause it to function properly. For example a device driver is a software resource abstraction that hides the details of how the physical device is manipulated to perform an I/O operation. Once a device driver has been implemented, other software can then use it to read or write the device without having to know details such as how to load commands and data into the device's controller registers. Abstraction can also be used to create resources that have no particular underlying hardware; messages and semaphores are example of this type of resource. Abstract data types and objects are common software mechanisms that are used for creating abstract resources.

Historically (and in Linux), the computer's components are abstracted into processes and resources; a file is a special case of a resource. A *process* is an abstraction of the CPU operation. In conventional computers, an object program (produced by the compiler and loader) is placed in the machine's executable memory. A process is an abstract entity that executes the object program

Linux is a *multiprogrammed* OS, meaning that it has been designed so that multiple programs can be loaded into the executable memory at once, and the OS will execute one program for a while, then switch to one of the others that are currently loaded in memory. That is, the executable memory is *space-multiplexed* – meaning that it is divided into blocks called memory partitions, and each partition contains a program ready for execution. The CPU can then be *time-multiplexed* to execute the programs currently loaded in different memory partitions. The OS must keep track of which program is being executed for which user (using some previously allocated set of resources); the process abstraction allows the OS to control the execution of each instance of a program. For example, the OS can decide which process should execute, and which should be made to wait for the CPU to become available. All of the aspects of abstracting the operation of the CPU into the process notion are part of *process management*.

Resource management is the procedure for creating resource abstractions and for allocating and deallocating the system's resources to and from the processes as they execute. For example, a portion of executable memory (often called the "RAM" or random access memory) must be allocated to a process when the process begins, and when the process releases memory, it must be made available to other processes. The memory resource manager is responsible for controlling the use of the machine's executable memory. A system resource is any logical component that is managed by the OS, and which can only be obtained by having the process request the use of the resource from the OS. The CPU is a specialized OS resource that is controlled by the kernel, though historically it has been treated as a special case compared to other system resources. Executable memory is another system resource, and it too has a specialized resource manager (the memory manager). What are the other resources that the kernel manages? All the system's devices are resources with their own abstraction models. Storage devices are normally abstracted so that information stored on them appears as a logical byte stream – commonly known as a *file*. In UNIX, there is an attempt to make every resource other than the CPU and executable memory look like a file.

2.2 Sharing Resources

The Linux notion of a resource implicitly relies on the idea that there are processes that can request resources, use them, and then release them. When a process requests a resource, it ordinarily needs exclusive use of the resource. *Exclusive use* of a resource means that when a unit of the resource is allocated to one process, then no other process has any access to that unit of resource. There are two critical aspects to this resource management: The competition to acquire units of a resource, and the assurance of exclusive use.

2.2.1 Managing Competition for Resources

Competition for a resource is controlled by that resource's manager – the OS must contain a component that will be responsible for accepting requests to use each resource, for allocating the resource to a process, and for retrieving a resource when a process releases it. The simplest model for using a resource is the following:

- A process P requests K units of resource X from the manager for resource X.
- If K units of the resource are available, the manager allocates them to P (marking its internal "assigned" and "available" data structures to indicate that they are allocated and unavailable to other processes).
- If there are not K units available, the manager for X blocks process P while it waits for K units of X to become available. P will be prevented from further execution until the resource manager for X allocates the K units to P.
- When at least K units of X become available through a deallocation by some other process P', the manager for X chooses one of the waiting processes whose request can be satisfied, say P, then allocates the K units of resources to the process.
- Process P is then once again ready to execute using the K units of resource X.

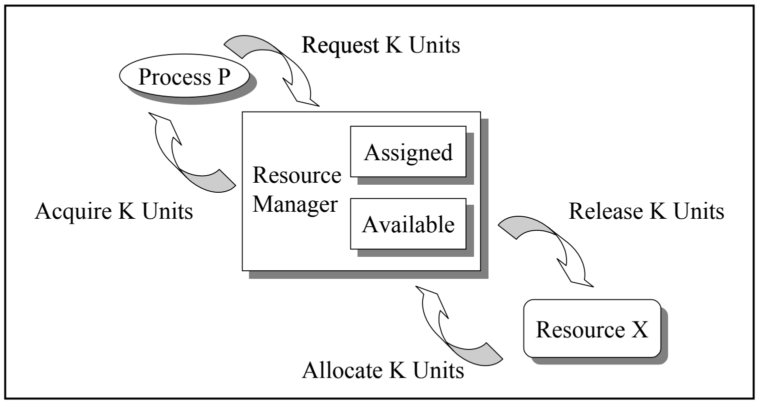


Figure 1: Resource Manager Schematic

Observe that the CPU resource also conforms to this same model of behavior: When processes P_i and P_j are ready to execute, but the CPU resource is being used by some other process P_k then P_i and P_j are blocked waiting for the CPU to become available. When P_k gives up the CPU, then the scheduler will choose a waiting process, say P_i, and allocate the CPU to it. The scheduler is just a resource manager for the CPU resource.

2.2.2 Exclusive Use of a Resource

The OS uses many different mechanisms to assure that a process has exclusive use of an allocated resource, depending on the nature of the resource. The way that the CPU is given exclusively to a process differs from the way memory or a device might be dedicated to the process. This is because the CPU is time-multiplexed, meaning that when a process is allocated the CPU, no other process can use it for a selected period of time. Choosing efficient and effective mechanisms to assure exclusive use of each resource is one of the challenges of designing any OS. A process is given exclusive use of the CPU by assuring that other processes cannot interrupt the process's execution (unless they are "more important" than the running process). Exclusive use of memory is assured through hardware memory protection mechanisms that prevent the CPU from accessing memory that is not allocated to the process currently using the CPU. Exclusive use of devices is assured by preventing the CPU from executing an I/O instruction for the device unless it is performed in behalf of the process that has been allocated the device.

Creating mechanisms to guarantee exclusive use of a resource can be very difficult. Over the years, OS and hardware designers have evolved to a model of operation in which part of the mechanism to assure exclusive use is implemented in the OS and part of it in hardware. A key aspect of this approach is that the hardware "knows" when the OS is executing on the CPU. This allows the hardware mechanisms to react to instructions issued by *trusted software*, and to ignore attempts to perform instructions that might violate exclusive use (executed by software that is *untrusted* by the OS). Contemporary CPUs such as the Intel 80286 (and above) microprocessors are built with a *mode bit* that can be set to *supervisor* or *user mode*. When the CPU is in supervisor mode – which is also called *kernel mode* – the presumption is that the CPU is executing trusted software – software that has been carefully designed and analyzed so that it performs exactly as intended. The hardware will execute any instruction in its repertoire and reference any memory address if the CPU is in supervisor mode. Otherwise (when the CPU is in user mode), the CPU is assumed to be executing untrusted software so the CPU will not execute *privileged instructions* (such as I/O instructions) and will only reference the memory allocated to the process currently using the CPU.

The mode bit is a very powerful mechanism. It is used by Linux and other contemporary operating systems to esure exclusive use of the system's resources: The entire Linux kernel executes in supervisor mode (so it is often called *kernel mode* as well as supervisory mode in all UNIX-like systems), and all other software executes with the CPU in user mode. The Linux kernel is trusted software, and all other programs are untrusted software that have not been analyzed to determine whether or not they are correct. This means, for example, that only kernel code can actually execute device I/O instructions; all user mode software must ask the kernel to perform an I/O operation by making a system call to the kernel. An obvious challenge in designing the hardware and OS is to determine a safe means by which user mode software causes the CPU to change modes to execute the kernel in supervisor mode, and to change the mode back to user mode when the kernel finishes its work. CPU modes are discussed more in Section 3, and complete details for the way the Intel 80386 and higher hardware – referred to as the *i386* architecture in the Linux source code –and Linux accomplish this is deferred until Exercise 5.

2.2.3 Managed Sharing

In some cases, two or more processes need to share a resource. For example, one process might read a file containing the number of hours each employee worked, while the second process writes payroll checks. While the second process is writing the check for record i, the first process could be reading the hours for record i+1. How can the first process tell the second process the amount of the check it should write? If the two processes shared a block of memory, then the first process could write the amount for record i into the shared memory then immediately start reading record i+1. When the second process was ready to write the check for record i it would read the amount from the shared memory.

The barriers that the system uses to enforce exclusive use of a resource (for example to prevent one process from reading or writing the memory that had been allocated to another process) generally prevent sharing strategies. In order to accommodate sharing when it is needed, the OS must make some provision for violating the exclusive use mechanism when both processes wish to share a resource. Just as the mechanism that is used to enforce exclusive use varies among resources, so too does the mechanism for allowing resource sharing. This adds a new level of complexity to the design of the resource manager.

2.3 A Partition of OS Functions

Over the years, OS designers have partitioned the functions that must be performed into a set of four classes:

- **Process and Resource Management**. This is the heart of a multitasking OS. This class of functionality creates the software environment that has the illusion of multiple virtual machines with their own logical resources being used by the multiple processes.
- **Memory Management**. This part of the system is responsible for allocating memory to processes, for protecting allocated memory from unauthorized access, and sometimes for automatically moving information back and forth between the memory and storage devices. The Linux memory manager employs *virtual memory* based on paging. These systems provide the most general functionality, accomplishing all the goals of allocation, protection, and automatic movement. However, there is a cost in time, space, and hardware for virtual memory.
- **Device Management**. Computers can be configured with a wide array of different device types. Some devices are low-speed character devices, e.g., a keyboard, touch-sensitive screen, or remote control; others are high-speed block devices, e.g., a packet network interface. The device manager function is to provide controlling functions for this array of devices.
- **File Management**. This part of the OS implements a specialized resource abstraction for storing information on the computer's storage devices (such as disk drives). In Linux, a file is a named sequence of bytes that can be read and written as blocks of one or more bytes. The file manager also provides directories for organizing collections of files. Like most contemporary operating systems, Linux uses a hierarchical file directory. The file manager uses the functionality of the device manager to implement these abstraction.

The UNIX philosophy – adhered to in Linux – is to implement the minimum amount of mechanism for the functions in the kernel, with the remainder being implemented in user space software (libraries, runtime software, middleware, and application programs). This approach attempts to implement as much of the OS mechanism in the kernel as possible, and to implement as much of the OS policy as possible in user space software.

A simple example of this philosophy is the way files are handled. Applications programs nearly always manipulate records of data instead of linear streams of bytes. In the UNIX philosophy, only byte stream files are implemented in the kernel. If an application intends to organize the stored information as records, then it is the application's responsibility to define the structure of the data (for example as a C `struct`), to translate the structure into a byte stream before it is written into a file, and to translate the byte stream back into a structure after it has been read from the file. The kernel file mechanism has no knowledge of any application's data structures – it only handles byte streams. User space code (libraries such as the `stdio` library or the application software) defines the data structures used with each byte stream file.

3 Kernel Organization

The Linux kernel uses the same software organization as most predecessor UNIX kernels: It is designed and implemented as a *monolithic* kernel.[4] In the traditional UNIX kernel, process and resource management, memory management, and file management are carefully implemented within a single executable software module. Data structures for each aspect of the kernel are generally accessible to all other aspects of the kernel. However device management is implemented as a separate collection of device drivers and interrupt handlers (that also execute in kernel mode). Each device driver is intended to control a particular type of hardware device, for example a 1.44 MB floppy disk. The philosophy behind this design is that the main part of the kernel should never be changed; the only new kernel space software that should be added to the system would be for device management. Device drivers are relatively easy to write and add to the kernel (compared to, say, adding a feature to process management).

As technology evolved, this design assumption became a serious barrier to enhancing the kernel. The first UNIX systems did not support the spectrum of hardware components that are common in today's machines, especially including bitmap displays and networks. The interface between the kernel and the device drivers was designed to support disks, keyboards and character displays. As hardware evolved to include these newer devices, it became increasingly difficult to provide appropriate kernel support wholly within a device driver.

Linux addressed this problem by providing a new "container" in which to implement extensions to the main part of the kernel – modules (see Figure 2). A *module* is an independent software unit that can be designed and implemented long after the operating system has been compiled and installed, and which can be dynamically installed as the system runs. The interface between a module and the kernel is more general than the one used with UNIX device drivers, providing the systems programmer with a more flexible tool with which to extend the kernel functionality than device drivers. Interestingly, modules have proven to be so flexible that they are sometimes used to implement device drivers. You will learn to write a module to extend the kernel in Exercise 4, and to implement a device driver in Exercise 9.

[4] There is a trend in many modern operating systems, such as Mach and other research operating systems, to use a microkernel organization where the normal kernel functions are distributed between a minimal microkernel that executes in supervisory space and a collection of servers that execute in user space. Your textbook provides more information about microkernel organizations.

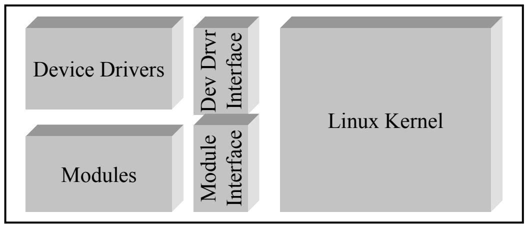


Figure 2: Kernel, Device Drivers, and Modules

3.1 Interrupts

The kernel reacts to service requests from any active, external entity. In some cases the external entity is a process that performs a system call (described in Section 3.2); in other cases a hardware component can request service from the kernel by issuing an interrupt. An *interrupt* is an electronic signal produced by an external component (such as a device) that is fielded by the CPU hardware, causing the CPU to begin executing a program sequence that is independent of the one it was executing at the time the interrupt occurred. Usually, the interrupt means that a device has finished an I/O operation that was started earlier by a device driver. However sometimes the interrupt signals an event such as the physical clock device having just "ticked." In the i386 architecture the hardware behavior of an interrupt can be represented by the algorithmic description shown in Figure 3. The basic part of the fetch-execute hardware algorithm is that the program counter (PC) holds the address of the next instruction to be executed. When it is time to execute the instruction, it is copied from the memory into the instruction register (IR). The control unit part of the CPU then decodes and executes the instruction from the IR.

```
InterruptRequest = FALSE;
…
while (haltFlag not set during execution) {
 IR = memory[PC];   /* Fetch into instruction register */
 PC++;              /* Point the PC at the next instruction */
 execute(IR);       /* Execute the current instruction */
 if (InterruptRequest) {
                    /* Interrupt the current process */
   save_PC();       /* Save the PC */
   restore_PC(IRQ); /* Branch to the ISR for this IRQ */
 }
```

Figure 3: The Hardware Process's Algorithm

When an external component wishes to obtain service, it makes an *interrupt request* (IRQ) that sets a conceptual interrupt request flag, `InterruptRequest`, in the CPU. As shown in Figure 3, the control unit hardware checks `InterruptRequest` at the end of each fetch-execute cycle. If an IRQ has occurred, the hardware algorithm causes the CPU to cease executing the sequence of instructions addressed by the hardware program counter (PC) and, instead, to branch to an *interrupt service routine* (ISR) for this

IRQ (by writing the address of the ISR into the PC with the `restore_PC(IRQ)` function in the algorithm description). The ISR address is stored in a kernel table that was setup when the kernel was initialized. When the branch occurs, the hardware saves the original PC value (address of the next instruction to be executed) of the interrupted process.

When the ISR begins executing, the CPU registers will contain values being used by the interrupted process (excluding the PC). The ISR immediately saves all the general and status registers of the interrupted process and copies its own values into every CPU register so that it can run the main part of the ISR routine to handle the interrupt occurrence.

A problem can occur if an interrupt occurs while one ISR is in the midst of execution when the second IRQ occurs. That is, before the processor can finish handling the first interrupt, it could restart due to the occurrence of a second interrupt. Depending on exactly which part of the ISR is currently being executed for the first interrupt, the CPU state could be lost or the interrupt processing for the first device operation might never be completed. Of course, some parts of the ISR can be interrupted and then resumed later without harm. The problem occurs if one ISR begins to execute and begins saving the state of the process it interrupted; if it is in the middle of saving CPU register contents when another IRQ occurs, then the second ISR would start saving registers, saving some values from the original process and some from the interrupted ISR execution. When the first ISR is eventually allowed to complete, it will continue saving the register contents of some process other than the one it originally interrupted.

It is easy to see that the second ISR will not behave correctly, and may even cause the first I/O operation to fail. This *race condition* between the execution of the ISR and the occurrence of another IRQ must be avoided if the machine is to perform I/O operations reliably.

The race condition can be handled by incorporating another mechanism to explicitly prevent interrupts from interrupting the handler. Machines, like the i386, that incorporate interrupts also include an interrupt-enabled flag. Linux provides one kernel function (`cli()`), to set the interrupt-enabled flag to `FALSE`, and another kernel function (`sti()`) to set the flag to `TRUE`. These two system functions are used throughout the kernel when it is important to avoid a race condition caused by interrupts. How does a designer know when to use `cli()` and `sti()`? By carefully analyzing the context in which any piece of the code executes. Clearly this is more difficult in a monolithic kernel than it would be in a kernel that used a more modular approach.

Interrupts do not alter the control flow of the process executing on the processor, they simply delay the process's execution by running extra programs "behind the process's back." Once `sti()` has been executed, the `InterruptEnabled` flag is set to `TRUE`.

3.2 Using Kernel Services

As suggested by Figure 4, user programs view the kernel as a large abstract data type (abbreviated as ADT – similar to an object) that maintains state, and which has a number of functions on its public interface – the *system call interface*. This ADT provides OS service to a user program whenever the program calls a function on its public interface;

the exact nature of the service depends on the particular function that is called, its parameters, and the state of the OS when the function is called. Hence, the public interface specification for the kernel is the main description of the OS functionality. In the case of Linux, the system call interface is nominally defined by the POSIX.1 specification. It is only nominal, since any version of Linux may not implement some of the fine details of the specification, and because the specification does not completely specify the semantics of the interface. However, the POSIX.1 specification is sufficiently complete that programmers are generally willing to accept it as a basic OS interface specification.

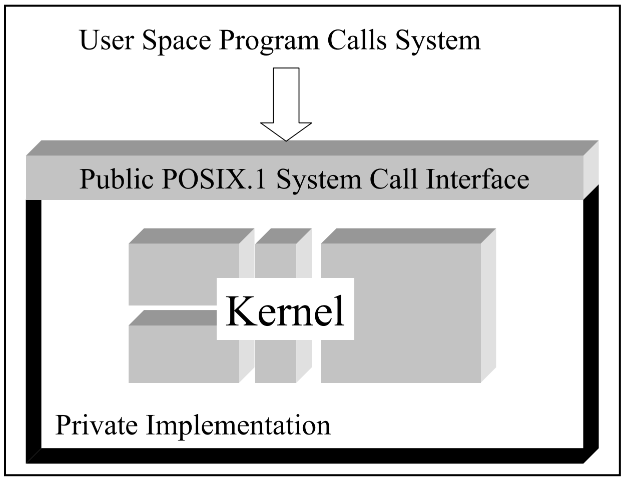

User Space Program Calls System
Public POSIX.1 System Call Interface
Kernel
Private Implementation

Figure 4: The Kernel as an ADT

Continuing the ADT analogy, the implementation of the POSIX.1 system call interface is private, so the implementations may differ from system to system, and even among versions of the Linux implementation. In theory – and generally in practice – the boundaries between device drivers or modules and the kernel are not discernible to the user program. These are characteristics of the internal design, not of the public interface. Thus, the POSIX.1 system call interface defines functions that normally are implemented in the kernel, device drivers, or modules. If the function is implemented in a device driver or module, when a user program invokes the function, the kernel fields the call and passes it through an internal interface to the appropriate device driver or module. In a Linux implementation of the POSIX.1 system call interface, some functions may even be implemented as user space programs – for example most versions of Linux rely on library code to implement the thread package [Beck et al., 1998].

If the kernel ADT were actually implemented as a C++ object, it would be a *passive* object. That is, the kernel software does not have any internal thread of execution or process – it is simply a collection of functions and data structures that maintain state. Any process that uses kernel services – the process is an *active* entity – makes a kernel request by (logically) making a procedure call on the system call interface. That is, a process that is executing outside the kernel begins to execute the kernel code when it makes the system call; this is contrasted with the idea that the process executing user code might obtain service from a distinct kernel process. In Linux, the process executing the user program also executes the kernel programs when a system call in made.

Conceptually, then, whenever a process executing an application program desires service from the OS, it simply calls the appropriate OS function through the POSIX.1 system call interface. Before the call the process is executing the application program, and after the call (but before the return) the process is executing the kernel software. However, recall that the kernel is executed with the CPU in supervisor mode but the application program is executed in user mode. When the process performs the system call, the CPU must change from user to supervisor mode, and when the system call returns the CPU must change from supervisor to user mode. The return case is simple to handle, since one of the privileged instructions that can be executed when the CPU is in supervisor mode is to switch the CPU mode from supervisor to user mode. Here is the dilemma: How can a user mode program switch the CPU to supervisor mode with the assurance that once the switch is done that the CPU will be executing trusted kernel code and not untrusted user code? If the program is executing in user mode, it must be able to direct the hardware to switch to the supervisor mode – the instruction must not be a privileged instruction. If there were an ordinary (not privileged) instruction to switch the mode, then what would prevent a program from executing the instruction then to execute privileged instructions as untrusted software?

Normally, the compiler translates any function call into machine language statements that prepare actual parameters for passing, then by performing a procedure call instruction on the entry point of the target function. In the case of kernel calls, this would have two significant implications:

- Every kernel routine (and hence the entire kernel) would be linked into every application program.
- Every kernel routine would immediately have to execute a machine instruction to switch the CPU to supervisory mode before performing the work.

Notice that since the kernel code would be linked into the user program, the code would be readable by the process while it was executing in user mode. It would be possible to copy the machine instruction for switching to supervisory mode and use it to switch the mode to run untrusted software.

CPUs that incorporate the mode bit also usually incorporate a hardware *trap* instruction. A trap instruction causes the CPU to branch to a prespecified address (sometimes as a function of an instruction operand), and also to switch the CPU to supervisory mode. Of course the details of each machine's trap instruction are unique to that machine. A trap instruction is not a privileged instruction, so any program can execute a trap. However the destination of the branch instruction is predetermined by a set of addresses kept in supervisory space which are configured to point to kernel code.

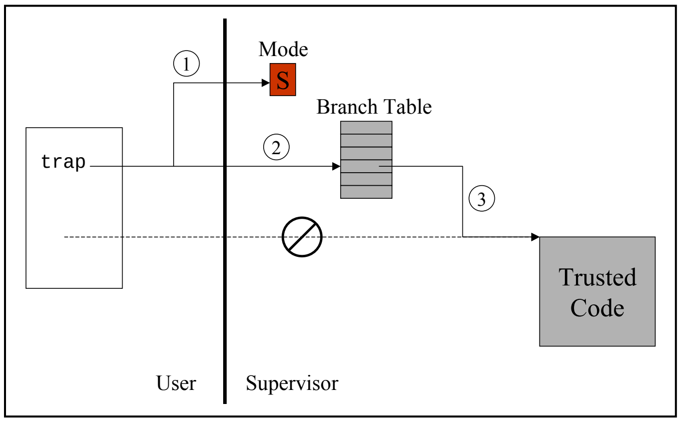


Figure 5: The Trap Instruction Operation

Suppose the assembly language representation for a trap is

```
        trap     argument
```

Figure 5 pictorially describes the behavior of a trap instruction:

1. Switch the CPU to supervisor mode
2. Look up a branch address in a kernel space table
3. Branch to a trusted operating system function.

The trap instruction provides a safe way for a user-mode process to execute only predefined software when it switches the mode bit to supervisor mode.

Now we can summarize the actions performed to accomplish a system call. For a system call, F, a stub procedure is used to invoke F (the stub is also named “F”). The stub is linked into the (user space) calling program. When a process executes the call to F at runtime, control is transferred to the stub procedure rather than directly to the kernel. The stub procedure validates the parameter values being passed to the kernel procedure, and could in principle authenticate the process that calls the stub procedure. Finally the stub procedure executes a trap instruction that switches the CPU to supervisory mode and branches (indirectly through a kernel table containing kernel function entry points) to the entry point for the target kernel function. Since the trap instruction is compiled into the stub procedure, user space programs cannot easily determine the destination address for the trap, and even if they could, they would be unable to branch indirectly through the kernel space function table. As a result, a user space program cannot directly invoke a kernel function, but must use the system-provided stub for doing so. In Exercise 5, you will add a system call to Linux by adding an entry to the system table for your kernel function.

3.3 Serial Execution

Linux is a multiprogramming kernel; however kernel functions are normally executed as if they were in a critical section. That is, once a process calls a system function, the function normally runs to completion and returns before the CPU will be allocated to a different process. However, IRQs can interrupt the execution of a system call to perform an ISR. This is called a *single-threaded kernel*, since (ignoring the ISRs) only one thread of execution is allowed to execute in the kernel at a time; one thread of execution cannot start a function call and become interrupted by the scheduler to let another process run (and possibly make a kernel call). There are at least two important implications of this approach:

- A kernel function can update various kernel data structures without being concerned that another process will interrupt its execution and change related parts of the same data structures. Race conditions among processes do not occur. (However, race conditions between the process and interrupt-induced execution may occur.)
- If you are writing a new kernel function you must always keep in mind that you cannot write code that will block waiting for a message or other resource that can only be released by some other process. This might produce deadlock in the kernel.

3.4 Daemons

Earlier in this section, we explained how normal processes execute kernel code, that is, that there are no special "kernel process" which executes kernel code. While this is accurate, there are several user-transparent processes called *daemons*, that are started when a Linux machine is started, and which must exist for correct operation of the operating system. For example, if the machine is connected to a network, there must be a process to respond to incoming network packets; another daemon process logs system messages and errors, and so on. The particular daemons that are running on any Linux installation varies according to the way the system administrator has set up the machine (more details of this are explained in the next subsection on booting up a Linux machine).

By convention, daemon processes execute programs whose name ends with a character "d", for example the network daemon is usually called `inetd`, the system logging daemon is called `syslogd`, and so on. You can make a good guess as to what daemons are running on your Linux machine by typing the following line to your shell

```
ps aux | more
```

The `ps` command reports the process status, and the `–aux` parameters indicate that you want a report in user format (the `u` parameter) for all processes (the `a` parameter) including those without a controlling terminal (the `x` parameter). As you scan the list, look for commands that end with the character "`d`" and which have a `TTY` field of "`?`" (no controlling terminal). You will normally see `syslogd`, `klogd`, `crond`, and `lpd` running on your system. You can find out what each of these daemons are doing by looking at the manual page (for example use "`man syslogd`" to read the manual page for `syslogd`) for the program that the daemon is running.

3.5 The Booting Procedure

Every computer has a sequence of instructions that are executed whenever the computer is powered up or restarted. Similarly, every OS has a sequence of instructions that it executes prior to beginning normal operation. The hardware bootup procedure is determined by the computer hardware designers when they construct the machine. Contemporary i386 machines have ordinarily been designed to support Microsoft operating systems, so they incorporate the Basic Input/Output System (BIOS) at a prespecified location in the system's read-only memory (ROM). When the computer is started, it begins executing from BIOS code. BIOS first runs the manufacturer's hardware booting sequence by executing the Power-On Self Test (POST). The POST performs various diagnostic tests to check the memory and devices for their presence and for correct operation. The POST usually takes several seconds – possibly even a couple of minutes. When the POST has completed, the machine is ready to load and start the operating system.

3.5.1 The Boot Sector

The OS will be loaded from the system's *boot disk*. An i386 system treats the `A:` floppy disk drive as the default boot disk if it contains a diskette; otherwise one of the system's hard disk drives is treated as the boot disk. A boot disk must be configured to contain a *boot record* at the first logical sector on the boot disk. The boot record fits into a single 512-byte disk sector (so it is often referred to as the *boot sector*). The boot sector is organized as shown in Figure 6 [Beck, 1998; Messmer, 1995].

`0x000`	`0x002`	<a jump instruction to `0x`*`0XX`*>
`0x003`	…	Disk parameters (used by BIOS)
`0x`*`0XX`*	`0x1fd`	Bootstrap program
`0x1fe`	`0x1ff`	0xaa55 (magic number for BIOS)

Figure 6: Boot Sector

When the POST finishes executing, the BIOS copies the boot record from the disk into memory. The boot record also includes various parameters that describe the physical description of the disk – the number of tracks, the number of sectors per track, and so on – stored in well-known locations in the boot record. Once loaded, the BIOS program can read these parameters from the well-known addresses. Further, when the boot record has been loaded, BIOS branches to the first location in the program. It then immediately branches to location `0x0`*`XX`* (the value of *`XX`* is `3e` for the Linux boot record and `1e` for the MS-DOS boot record). The small program stored in the boot record can now load a more sophisticated loader from other sectors on the disk; the more sophisticated loader loads the OS.

Hard disks can have an additional level of abstraction, called *disk partitioning*. If a hard disk is partitioned, each resulting partition is treated like a physical disk above the abstract machine that accesses the physical disk (BIOS, in MS-DOS). A hard disk can be partitioned to have up to 4 different logical disks, each with its own set of logical sectors. If a disk partition is a *bootable disk partition*, then its logical sector number zero will be a boot sector. In a partitioned disk, the physical head 0, track 0, sector 1 (the first logical disk sector) contains a *partition sector* rather than a boot sector. The partition sector provides information to describe how the hard disk is partitioned into logical disks. Roughly speaking, the partition sector starts off with a 446-byte program. When the hardware is powered up, it will go to head 0, track 0, sector 1 and begin executing code. If the disk is partitioned, there will be a 446-byte bootstrap program at the first byte in the sector. If the disk is not partitioned, there will be a jump instruction to start the execution on the boot program stored at location `0x0`*`XX`* in the sector. After the bootstrap program, there is a 64-byte partition table for the disk. It contains space for the four partition entries, each of which describes the portion of the physical disk that is used for its partition (starting sector of the partition, ending sector, number of sectors in the partition, and so on). The last two bytes of the partition sector contain a magic number of `0xaa55` to identify the partition sector.

An i386 computer can be booted with Linux by using a Linux boot disk where the boot record contains a Linux boot program rather than a Windows OS boot program. In cases where the boot disk is a partitioned hard disk, it is possible to load different operating systems in different partitions. Windows operating systems can handle multi partition disks by designating one of the partitions as being the *active partition*. The system will always boot from the active partition.

Linux provides a special *Linux Loader* (LILO) that can be placed in the system boot record to allow the user to choose at boot time which partition should be the active partition. In this case LILO is loaded in the boot record so that the BIOS will run it after the POST.

3.5.2 Starting the Kernel

When the machine is powered up, the hardware fetch-execute cycle begins, causing the control unit to repeatedly fetch and decode instructions, and the arithmetic-logic unit to execute them. Intuitively, the hardware is now executing some "process" that we will call the *hardware process*. The hardware process is not a Linux process since it starts when the hardware starts, long before the kernel has even been loaded. After the POST has completed, the boot record has been read, and the loader has placed the OS into primary memory, the booting procedure begins to run kernel code to initialize the computer's hardware. The computer is then ready to start the kernel by first setting the CPU in supervisor mode and branching to the main entry point in the kernel. This main entry point is not an ordinary C program with the conventional `main` header line, since it is started in the boot sequence (rather than being started as a conventional main program).

Once the main entry point has been entered, the kernel initializes the trap table, the interrupt handler, the scheduler, the clock, modules, and so on. Near the end of this sequence, it initializes the process manager, meaning that it is prepared to support the normal process abstraction. Logically, the hardware process then creates the *initial process*. The initial process is allocated the first entry in the kernel's process descriptor table, so it is also referred to internally as process 0, or `task[0]`, or `INIT_TASK`. The initial process then creates the first useful Linux process to run the `init` program, then the initial process begins to execute an idle loop. That is, the initial process's only duties after the kernel is initialized is to use idle CPU time – it uses the CPU when no other process wants to use the CPU (so it also sometimes referred to as the *idle process*).

The first real process continues initializing the system, but now using higher level abstractions than were available to the hardware process. It starts various daemons, the file manager, creates the system console, runs other `init` programs from `/etc`, `/bin`, and/or `/sbin`, and runs /`etc/rc` (if necessary).

Starting the kernel is a complex procedure, and there are many opportunities for something to go wrong. There are several places in the procedure where various alternatives are checked in case the default boot algorithm fails. By the time the kernel initialization has been completed, the initial process will be enabled and several other daemons will have been started. Chapter 3 of [Beck, et al., 1998] provides a more detailed discussion of the steps involved in starting up the kernel.

3.6 Logging Into the Machine

During initialization the kernel will have created one process on each communication port that can be used to support user logins. These processes are each running a copy of the `getty` program (see

Figure 7). The `getty` process initializes itself, then waits for a user to begin using the port. When the port begins to be used, `getty` runs the `login` program; it expects a user identification on the first line and a password on the second line. After the port's `login` program obtains the identification and the password, it verifies the user's identity by looking up the given identification and password in the system's `/etc/passwd` file. Each entry has the form:

```
jblow:eNcrYPt123:3448:35:Joe Blow:/home/kiowa/jblow:/bin/bash
```

Each line in this file is a record for a different user, with fields in the record separated by the ":" character. In the example record, the user login is "`jblow`", "`eNcrYPt123`" is an encrypted copy of the password, the next two numbers are the user ID and group ID, the fifth field is the user's real name, "`/home/kiowa/jblow`" is the user's home directory, and "`/bin/bash`" is the path to the shell command line interpreter preferred by the user. If this authentication is successful, the login process changes the current directory to the user's home directory, then executes the specified shell program so that the user can interact directly with the login process using the shell.

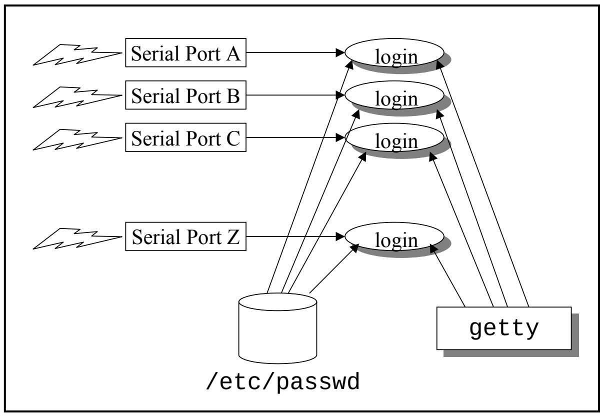


Figure 7: The `getty` Program

A user who logs into a UNIX machine simply begins using a process that was created when the machine was started. (There are actually many more details to starting a shell, but conceptually, this description is accurate.) Thus the user's process is executing a shell program (for example the Bourne shell, the C shell, or the Korn shell) with a unique copy of the data and stack segments for the shell execution. The unique data includes the user's environment variables such as the PATH variable. When the user logs off the machine, the `login` process returns from its shell call and waits for the next user to login.

3.7 Control Flow in the Machine

Ultimately, the behavior of the computer is controlled by the action of the hardware process – remember that there is only one hardware process. It will execute the algorithm shown in Figure 3 until the computer is halted. Even though the kernel may switch from running one program – an ISR, a user program, and so on – the hardware process operates at a level of abstraction *below* the kernel. Therefore the hardware process has no notion of software process, ISR, or other kernel abstraction; instead, it just fetches and executes instructions. At this low level view of the behavior of the computer, even interrupts are nothing more than branch instructions.

Figure 8 provides an intuitive visual description of the behavior of the hardware process. When the machine is powered up, the hardware process starts executing the fetch-execute cycle on the BIOS program (including the POST). Eventually it runs kernel code, then (in this hypothetical example), it executes code on behalf of Linux Process i, then back to the kernel, then to an ISR, and so on.

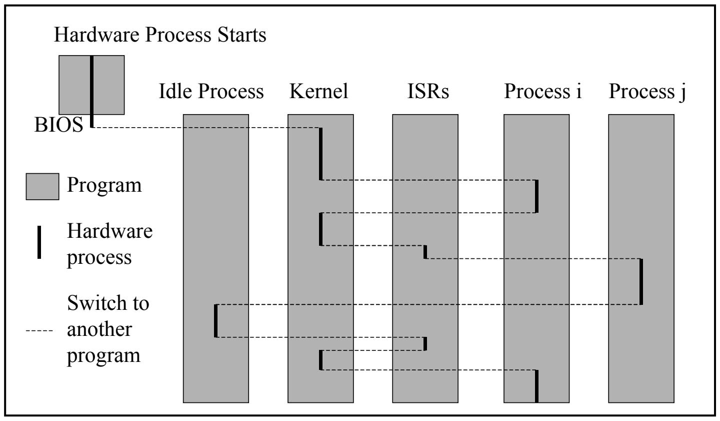


Figure 8: The Hardware Process

4 Process and Resource Management

The process manager is responsible for creating the process abstraction that programmers use, and providing facilities so that a process can create, destroy, synchronize and protect other processes. Similarly, resource management involves creating appropriate abstractions to represent entities that a process might request (and block their execution if the resources are unavailable); besides abstraction, resource managers must provide an interface by which a process can request, acquire, and release resources.

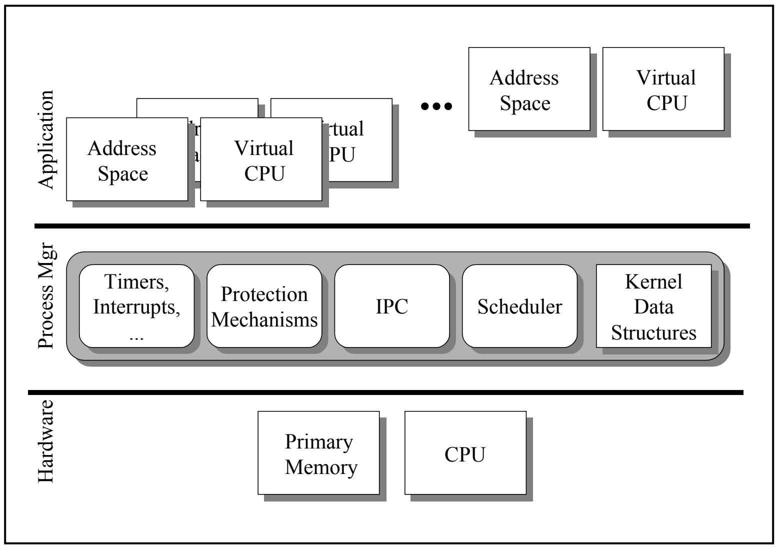


Figure 9: Process Abstraction

It is useful to summarize the execution of a program at various levels of abstraction to get a firm understanding of the job of the process manager (see Figure 9):

- **Hardware Level**. The hardware simply fetches an instruction from the memory location addressed by the PC, executes it, then fetches the next one, and so on. The hardware does not distinguish one program from another – they are all simply instructions in the primary memory. As a result, there is only the notion of the hardware process executing stored instructions, but no further abstraction such as a Linux process.
- **Process Manager Level**. The process manager creates a collection of idealized *virtual machines*, each of which has the characteristics of the host CPU when it is running in user mode. The process manager uses the hardware level to create a Linux process by using timers, interrupts, various protection mechanisms, interprocess communication (IPC) and synchronization mechanisms, scheduling, and a collection of data structures. Applications interact with the process manager (using the system call interface) to create instances of the virtual machine (with `fork()`), to load the address space with a particular program (using `exec()`), to synchronize a parent and a child virtual machine (with `wait()`), and so on.
- **Application Level**. Conventional Linux processes are used at the application level. The process's address space is the memory of its virtual machine (containing the text, stack, and data segments), the virtual CPU instructions are the user mode hardware instructions augmented by the system call interface functions, and the virtual resources (including devices) are manipulated via the system call interface.

The process manager must manipulate the hardware process and physical resources to implement a set of virtual machines. It must multiplex the execution of the virtual machines on top of the single physical machine. (Linux supports multiprocessor CPUs, but in this manual we focus only on the single CPU implementation.) It must also provide facilities to accommodate sharing – protection and synchronization.

Within the kernel code, the terms *task* and *process* are used almost interchangeably. In this book, we will generally use the task terminology when the execution takes place in supervisor mode and the process terminology when execution occurs in user mode. When a process is created, it is normally created with a new address space – a set of virtual addresses that the process can read or write as it executes. From the kernel perspective, this means that when the hardware process is executing in behalf of process i, it can read and write only the addressable machine components that correspond to the virtual address in process i's address space.

4.1 Running the Process Manager

The portion of the kernel that handles process scheduling (like all other parts of the kernel) is only executed when a process begins executing in supervisor mode – either due to a system call or an interrupt (see Figure 10).

- **System call**. Suppose a process is executing user mode software and it makes a system call. The process traps into the kernel code to the entry point for the target function. The function is executed by the kernel task corresponding to the user process that made the system call. When the function completes its work, it returns to the kernel so that it can perform a standard set of tasks in the "`ret_from_sys_call`" code (a block of assembly language code in the kernel). This code block dispatches any accumulated any system work – such as handling signals, completing certain forms of pending interrupt process (called "executing the bottom halves of pending interrupt handlers"), or scheduling other tasks. The system call is completed with the calling process in the `TASK_RUNNING` (ready to use the CPU when it is available) state, or in the `TASK_INTERRUPTIBLE` or `TASK_UNINTERRUPTIBLE` state if the kernel cannot complete the requested work immediately. If the task is blocked (or it has used all of its time slice) the scheduler is then called to dispatch a new task.
- **Interrupt (IRQ)**. When an IRQ occurs, the currently running process completes executing its current instruction, then the task starts executing the ISR (interrupt service routine) for the corresponding IRQ. Linux differentiates between fast and slow interrupts. A *fast interrupt* is one that takes very little time to complete, so while it is being processed other interrupts are disabled, the interrupt is handled, the interrupts are reenabled, then the user process is continued. A *slow interrupt* involves more work: After the interrupts are disabled, the task executes the ISR. The ISR can have a bottom half, which has work that needs to be performed before the interrupt handling is completed, but which does not have to be done in the ISR itself. The pending bottom half work is scheduled in a kernel queue for the next time that bottom halves are to be processed (see the `ret_from_sys_call` processing in the system call description). Fast interrupts can, themselves, be interrupted; therefore, the queue of bottom half work can build up when nested interrupts occur. If the same ISR is invoked two or more times before the bottom halves are run, then the corresponding bottom half will only be run once even though the ISR executed multiple times. When the slow interrupt finishes the ISR, it executes the `ret_from_sys_call` block of code.

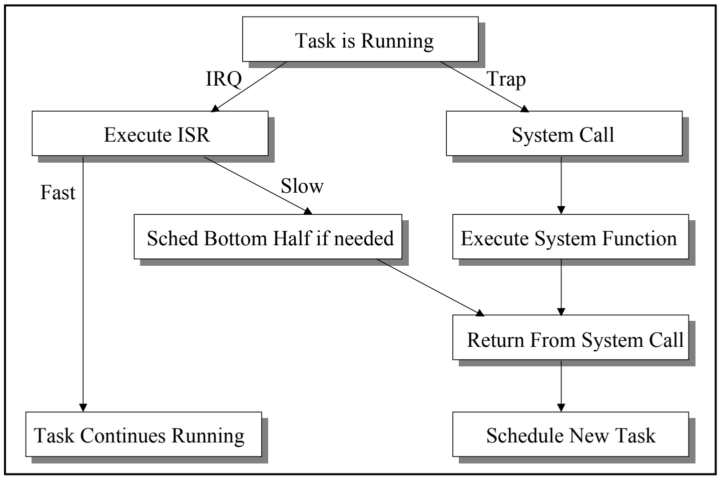


Figure 10: Part of Task Control Flow in the Kernel

4.2 Creating a New Task

A new task/process is created when its parent process invokes the `fork()` system call. When the kernel creates a new task, it must allocate a new instance of the *process descriptor* so that it will have a data structure to hold all the information it needs to manage the new task. In the Linux kernel, the process descriptor is an instance of the `struct task_struct` data type. The *process table* keeps the collection of process descriptors for each task/process. Older versions of the Linux kernel incorporated a static array of pointers to `task_struct` instances. The current version has replaced the static table with a linked list of pointers. In either case, the idle task occupies the first slot in the list or table (hence the designation of `task[0]` for the idle process).

Each process descriptor may be linked into one or more lists (besides the process table), depending on the current state of the process. For example, if the process is in the `TASK_RUNNING` state, it is in a list pointed to by a static kernel variable (named `current`) indicating which processes are currently ready to use a CPU. If it is blocked waiting for an I/O operation to complete, then it appears in a list of processes waiting for an interrupt from the device. You will begin a detailed study of the `struct task_struct` in Exercise 1, culminating in Exercise 9.

The `fork()` system call creates a new process by performing the following steps:

1. Allocate a new instance of a `struct task_struct` and link it into the `task` list.
2. Create a new kernel space stack for the task to use when it is executing in the kernel.
3. Copy each field from the parent's task descriptor into the child's task descriptor.
4. Modify the fields in the child's descriptor that are specific to the child.
   - Save the new process identifier (PID)
   - Create links to this task's parent and siblings.

   - Initialize process-specific timers (such as creation time, time left in the current time slice, and so on)
5. Copy other data structures that are referenced in the parent descriptor and which should be replicated for the child process.
   - Create a file table and a new file descriptor for each open file in the parent's file table.
   - Create a new user data segment for the child task, then copy the contents of the parent's user data segment into this new segment. (This could be very time-consuming, since the data segment could contain megabytes of information.)
   - Copy information regarding the signals and signal handlers.
   - Copy virtual memory tables.
6. Change the child's state to `TASK_RUNNING` and return from the system call.

Of course an `execve()` system call will also greatly influence the contents of the process descriptor, since it causes the process to load and execute a different program than it was executing when it called `execve()`. Briefly, the `execve()` code causes the kernel to find the new executable file in the file system, to check access permissions, to adjust the memory requirements as necessary, then to load the file for subsequent execution. This will require that the details of the memory references in the process descriptor be updated.

The Linux kernel also includes a system call named `clone()`, that is intended to be used to support threads. The `clone()`and `fork()`call both invoke the internal kernel function named `do_fork()`, so they behave almost the same. The difference is in the way that the parent and child data segment is handled: While `fork()` copies the data segment, `clone()` shares it (and most of the rest of the parent's address space) between the parent and child tasks. Beck, et al. [1998] point out that even though the POSIX interface specifies calls to invoke threads, and that `clone()` is intended to support threads, that the 2.0.x versions of the kernel use the Pthread library to implement them.

4.3 The Scheduler

The *scheduler* is the kernel component responsible for multiplexing the CPU among the programs in memory, that is, among the processes that are in the `TASK_RUNNING` state. It incorporates the policy used to select the next process to be allocated the CPU. The `schedule()` kernel function can be called via a trap, and it is also called as part of the `ret_from_sys_call` code block – so it always runs as a task that is associated with a user process or with an interrupt. The scheduler is responsible for determining which runnable task has the highest priority, then for dispatching that task (allocating the CPU to it). In Exercise 9 you will study the Linux scheduler in detail.

4.4 IPC and Synchronization

There are two distinct synchronization mechanisms used in Linux – one used within the kernel code itself, and the other to provide synchronization mechanisms for user processes. The kernel always executes as a single task invoked via the system call interface, or as activity caused by an interrupt. Kernel activity can never be interrupted by a system call, since when either an interrupt is being handled or a system call is being processed, then no user process can issue a system call. Therefore, the primary need for synchronization within the kernel is to be sure that interrupts do not occur while the current kernel code is in a critical section. This is satisfied by disabling interrupts at the beginning of the critical section (using the `cli()`— CLear Interrupt – kernel function), then reenabling them (using the `sti()`— SeT Interrupt – function) when the critical section is complete.

The external synchronization mechanism is based on an event model. The kernel implements an abstract data type called a `wait_queue` to manage events. Each event has a corresponding `wait_queue`; a task can be added to a designated `wait_queue` using the `add_wait_queue()`, kernel function. Correspondingly, the `remove_wait_queue()` kernel function removes a single task from the designated wait queue. This abstract data type is the basis of any system call to perform synchronization, such as the System V semaphore system calls. You will learn more about the `wait_queue` in Exercise 8.

There are four different mechanisms by which a user process can perform IPC using the kernel:

1. **Pipes (and named pipes)**. These mechanisms export the same interface as files, so much of the implementation is like a file. A pipe uses a 4 KB circular buffer to move information from one address space to another via the file `read()` and `write()` system calls. From a process management perspective, this procedure is straightforward. There is more discussion of pipes in Exercise 2.
2. **System V IPC**. This interface allows user processes to create IPC objects (such as semaphores) in the kernel. The kernel allocates the data structure instance, then associates it with a user-space identifier that is shared among user processes. In the case of semaphores, one process creates the semaphore, then others manipulate it using the external reference and operations. The semaphore semantics are implemented using a `wait_queue` instance. System V messages generalize this approach so that structured messages can be saved/retrieved from kernel data structures.
3. **System V shared memory**. The shared memory facility is the generalization of the System V IPC mechanism. In this case, a block of memory is allocated in the kernel, then an external reference is used by different processes to access the memory. You will study the shared memory implementation in Exercise 6.
4. **Sockets**. This mechanism is a special case of the network socket funtionality. A socket implementation requires that the kernel allocate and manage buffer space and socket descriptors. These kernel entities are normally used by the network code, though they can also by used in the “UNIX name domain” as a specialized form of pipe. They differ from pipes in that a socket is full duplexed, and is designed to use a protocol for reading and writing information.

4.5 Protection Mechanism

The primary form of protection used in the kernel is address space isolation. This depends on a process being able to define a set of virtual addresses that cannot be read nor written by any other process (except using the IPC and memory sharing mechanisms mentioned in the previous section). In Linux the address space protection scheme is built into the virtual memory mechanism, discussed in the next section.

The other predominant protection mechanism is file protection. Processes and files are associated with users: Each file has an owner user, identified by a user ID (`uid`). A user can be associated with one or more groups, each having a group ID (`gid`). Each file enables read, write, and execute permission for the file owner, a process that is in the same group as the owner, or any other process (called world permissions). The kernel enforces this model by carefully checking permissions whenever it manipulates a file.

5 Memory Management

The memory manager is responsible for:

- Allocate blocks of the system's primary (or executable) memory upon request.
- Ensure exclusive control of a block once it has been allocated.
- Provide a means by which cooperating processes can share blocks of memory with one another.
- Automatically move information between the primary and secondary memory.

Linux uses a demand paged virtual memory model as the basis of its memory management design. In this model each process is allocated its own *virtual address* space. Processes reference virtual addresses, and the system maps each such reference into a primary (also called *physical*) memory addresses prior to accessing the memory location. The kernel and the hardware, together, ensure that the contents of the virtual memory location are placed into the physical memory, and that the corresponding virtual address is bound to the correct physical memory location when it is referenced by the process.

Like all demand paged virtual memory managers, the basic unit of memory allocation and transfer is a *page*. In the i386 implementation each page contains 2^{12} (4,096) bytes. Since Linux uses a paged virtual memory approach, the general characteristics of the manager's responsibilities are that:

- Blocks are allocated and deallocated as physical memory page frames.
- The protection mechanism is on a page-by-page basis.
- The memory sharing is based on pages.
- Automatic movement through the memory hierarchy is controlled by moving pages back and forth between the secondary and primary memories.

5.1 Managing the Virtual Address Space

Each process is created with its own virtual address space. In the i386 architecture a virtual address is 32 bits wide, meaning that the space contains addresses for 4 GB. Since a page is 2^{12} bytes, this means there are 2^{20} pages in an address space.

Each virtual address space is divided into segments – a 3 GB *user segment* and a 1 GB *kernel segment*. A programmer can use any address in the user segment to reference all of its code and data by having the OS *map* information into specific virtual addresses. Unmapped virtual addresses are simply not used. The virtual addresses in the kernel segment are permanently mapped and are associated with fixed physical memory addresses used by the kernel. This means that every process's virtual address space shares the same kernel segment, since the kernel virtual addresses all map to the physical addresses used by the kernel.

Each of the kernel and user segments is further partitioned into code and data *sectors*. Each virtual address contains a sector identification and offset within the sector. When the CPU is in the instruction fetch phase, it always references a code sector (in either the user or kernel segment). Therefore, the compiler does not bother to generate the sector identification as part of the address used by an instruction (there is no need to have special object code formats for Linux).

Whenever a process is executing, its state includes a *segment selector*. If the process is executing in user space, then the segment selector is set to `user`, but if it is executing in the kernel segment the selector is set to `kernel`. The memory manager forms the virtual address by using the value of the segment selector with the offset address provided by the process. Linux provides macros for setting the segment selector for each of the four segments in the virtual address space.

When a process is initialized, it defines its virtual address space by specifying the set of virtual addresses that it will be using. The kernel provides a function named `do_mmap()`, that reserves a set of virtual addresses in the virtual address space. Part of the procedure to determine the virtual addresses to be used, then, is to read the executable file to determine the load map (and virtual addresses) for the executable file. The `do_mmap()` function is called by `execve()` when it loads a file. Other parts of the OS can also call `do_mmap()`; for example, the `brk()` system call also invokes `do_mmap()` when it needs to add more space to the heap used to support `malloc`(). Once virtual addresses have been mapped, the corresponding file contents are associated with a block of virtual addresses. However, the contents of the virtual addresses are stored in the secondary memory (on the disk) where they will remain until they are referenced as the process executes in the virtual address space.

5.2 The Secondary Memory

A paging system is designed to have (the mapped part of) each process's virtual address space defined in the secondary memory. The Linux system may be configured so that an entire disk partition is dedicated to holding these virtual address spaces, or the virtual address spaces may be saved in individual files kept in an ordinary partition. For historical reasons, this disk partition is called a *swap device,* or if files are used then each file is called a *swap file*. (In some operating systems, for example Windows NT, these entities are more accurately called paging disks and paging files.) There can be a compile-time constant (`MAX_SWAPFILES`) number of swap devices or files – the default value is 8.

When a virtual address is mapped, corresponding physical space is reserved in the swap device/file. However, no primary memory is allocated or loaded as a result of the mapping operation. When the process is ready to run, then the pages that contain the entry point of the program it is using are loaded into the primary memory. As the process executes, the memory manager translates each virtual address referenced by the process into the physical address containing the information that was copied from the swap device/file when the page was loaded.

5.3 Handling Missing Pages

The system address translation mechanism (see the next section) can determine that the process is referencing a *missing page* that is not currently loaded in the primary memory, but which is defined in the swap device/file. When the process references a missing page, the memory manager copies the missing page from the secondary memory into the primary (physical) memory. Page loading requires that the memory manager find the target page in the swap device/file, copy it into a physical memory page frame, adjust the appropriate page translation table(s), then to complete the memory reference.

Each page frame in the primary memory has a kernel data structure of type `mem_map_t` to provide details about the state of that page frame. A node of type `mem_map_t` may appear in various lists, depending on whether the page is unallocated, locked, shared, and so on. If the page is loaded, then the `mem_map_t` instance references the inode and offset of the file where the page is stored in the swap device/file. All other information required to manage the page frame (such as locking and sharing) is kept in the `mem_map_t` data structure.

Suppose that the manager has determined that it needs to load a page into a page frame. The memory manager first attempts to acquire a *block* of new page frames for the process by calling an internal kernel function, `__get_free_pages().` In this function, a parameter is used to select a strategy to be used in seeking a page frame – for example if an interrupt routine is searching for space, it is treated differently than if a user process is requesting space. The size of the block is a kernel compile-time parameter, normally set to allow blocks to be up to 32 pages. Page frames are managed as a free list of blocks of size 2^k page frames. Page frame blocks are allocated by finding the right size of block, or by splitting a larger block. If there is no block available, then the page manager attempts to free space. Blocks may have been allocated to the file system to buffer pages being read or written by block devices (see Exercise 12), so the page manager first looks to see if there is a block that can be deallocated from the buffer pool. If no block can be found there, the page manager next attempts to reclaim page frames that were reserved for System V shared memory (see Exercise 6). If the block request still cannot be satisfied, the page manager begins looking at all page frames allocated to hold user space portions of virtual address spaces using an approximation of a global least-recently-used (LRU) replacement algorithm.

If a page has to be removed from the physical memory, then the page manager checks to see if it is *dirty* – meaning that it has been written to, so that it is different from the copy that exists in the swap device/file. If a page is not dirty there is no need to write it back to the swap device/file.

5.4 Address Translation

The Linux memory manager uses a demand paged virtual memory strategy. Processes issue virtual addresses (that have already been mapped), and the memory manager is responsible for determining if the corresponding page is loaded in some physical memory page frame or not. If it is not loaded, the page is found in the secondary memory (the disk), then it is loaded into a page frame in the physical memory as described in the previous section. If the page is currently loaded in some page frame in the physical memory, the virtual address is translated into the appropriate physical memory address. The physical memory location can then be read or written as desired.

Linux defines an architecture-independent memory model that transcends current-day CPUs and memory management units (MMUs), so it includes components are not used in the i386 implementation. In the general model, a virtual address is translated into a physical address using a three-level map. A virtual address, j, is partitioned into four parts:

- A *page directory* offset, j.pgd.
- A *page middle directory* offset, j.pmd.
- A *page table* offset, j.pte
- And the offset within the page, j.offset.

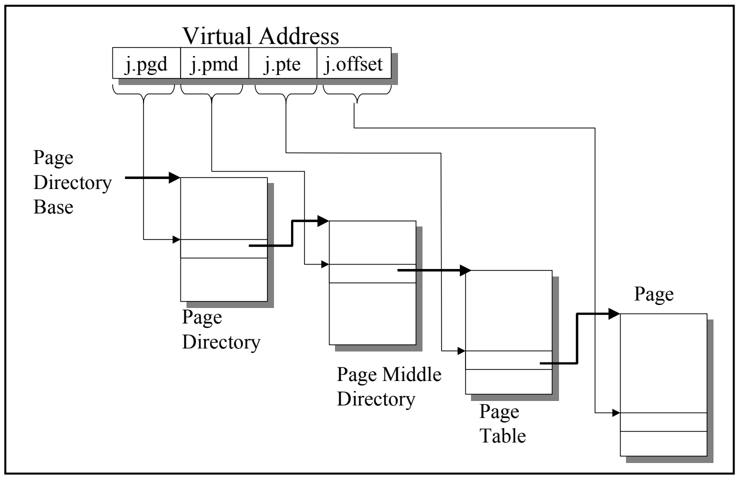


Figure 11: Virtual Address Translation

If a page is loaded into the physical memory, the physical address, i, for a virtual address j is determined by:

i = PTE(PMD(PGD(j.pgd) + j.pmd) + j.pte) + j.offset

where PGD represents the page directory table, PMD represents the page middle directory table, and PTE represents the page table. Conceptually (see Figure 11), this means that the virtual address is first divided into the four parts, then the j.pgd part is used to locate an entry in the page directory. This page directory entry references the base of a page middle directory and the j.pmd part of the address is used as offset into the designated page middle directory. This references an entry in the page middle directory that points at the base of the page table to be used. The j.pte portion of the virtual address is an offset in the page table that references a page descriptor containing the physical address of the first location in the page frame that contains the target page. Finally the page offset, j.offset, is added to the page frame address to determine the physical memory address, i, for the virtual address, j. Of course, if any map is undefined, a missing page interrupt will be generated during address translation, causing the page manager to load the page and to then map the virtual address.

The i386 microprocessor and compatible MMUs do not have sufficient hardware to support the full three-level translation process, so in this architecture only two of the three levels of translation are implemented. This is accomplished by reducing each page middle directory to contain only a single entry; this means that the j.pmd part of the address is not used since the page directory entry points directly at the single entry in the page middle table.

6 Device Management

There are two basic approaches to device management in Linux: Using polling to determine when a device has completed an operation, and using interrupts. In the polling case (see Figure 12) the user process performs an I/O request (for example a `read` system call on a device) at the system call interface, causing the device driver to start the device. The corresponding task then periodically polls the device to inspect its status to determine when the device has completed the operation.

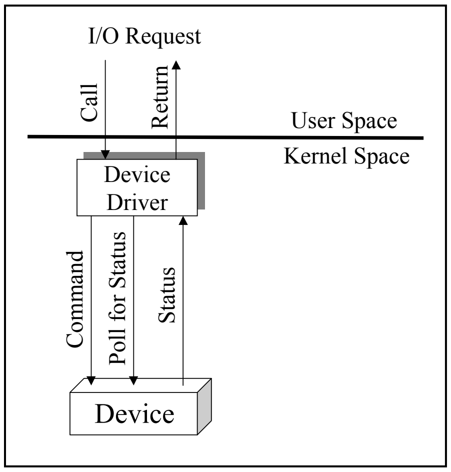


Figure 12: Polling Mode I/O

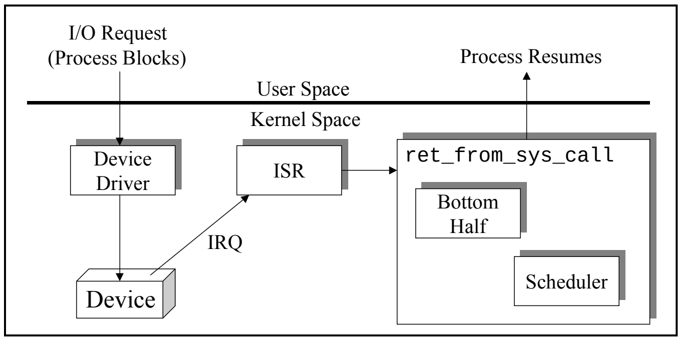


Figure 13: Interrupt Mode I/O

In interrupt mode I/O, a device driver, interrupt request (also called IRQ), interrupt handler (also called interrupt service routine or ISR), and a device bottom half may be involved in carrying out the operation. As described in Section 4.1 and summarized in Figure 13, when a process issues an I/O request to the kernel the device driver checks the status of the device, and if it is available, starts the device on the I/O operation. The task blocks in state `TASK_INTERRUPTIBLE` while the device is performing the I/O it requested. Meanwhile, the device operates autonomously until it completes the operation, at which time it issues an IRQ to the CPU. The IRQ causes the ISR that is associated with the IRQ to be executed; the ISR performs a minimum of the processing tasks associated with the completion of the operation, and marks its bottom half for later processing if necessary. Once the ISR (and all other ISRs that might have interrupted the original ISR) has completed, the `ret_from_sys_call` code block is executed. This block of code (so-named because it is also run when any system call is completed) runs all marked bottom halves, marks the completed processes as ready (`TASK_RUNNING`), then calls the scheduler. Ultimately the scheduler will dispatch the subject task and it will resume operation following its I/O request.

Interrupt mode is generally the preferred I/O technique, since it allows the CPU to be allocated to other processes whenever one process is waiting for an I/O operation to complete. However, polling may be used in any of the following situations:

- The I/O operation is so critical to the complete operation of the system that it is deemed worthwhile to have the CPU poll the device for completion without the possibility of other processes running in the meantime.
- The I/O operation is very fast, and the overhead of interrupt management is not considered to be worthwhile.
- The hardware device does not have an interrupt capability.

For the remainder of this discussion, we will focus on interrupt I/O mode.

6.1 The Device Driver

Device drivers and devices are referenced using major and minor numbers. A *major number* is associated with each device driver that will be used by the Linux kernel. (Device driver major number reservation is a manual procedure whereby a designated person decides which device drivers are "standard" and should have a permanent major number assigned to them.) For example, major number 6 is associated with parallel interfaces, 21 with a SCSI interface, and so on. Temporary major numbers can be assigned while a device driver is being debugged. Each device driver can be used with more than one different physical device, for example a single device driver for a hard disk drive can be used with multiple disk drives on a given machine. The *minor number* is used to distinguish among the physical devices that use a particular device driver.

UNIX devices are classically divided into block devices and character devices. A block device is one whose physical characteristics allow blocks of data to be referenced in any order. A character device reads or writes a single character at a time, and the data must be read or written in sequential (byte stream) order. Even though some block devices allow reading of the blocks in random order, buffering is used with block devices but not with character devices. (The file manager is able to predict the order in which blocks will be read, so it can direct the input and output buffering – see the next section.) The APIs for the two classes are different, meaning that the functions implemented in a block device driver are different than for a character device driver. Linux maintains the general distinction, but the differences between the APIs for devices of the two classes are much smaller than in UNIX systems.

When a machine is booted, the device drivers are normally *registered* with the OS. (Linux modules can be used to implement device drivers, and they need not be registered until run time – see Exercises 4 and 10.) There is a kernel function that is called to register a device. The kernel function requires the major number, a character string name for the device, and a list of type `struct file_operations` that identifies the entry points for every device operation supported by the driver:

```
check_media_change()
fasync()
fsync()
ioctl()
lseek()
mmap()
open()
read()
readdir()
release()
revalidate()
select()
write()
```

The API for the device driver is the same as the file operation API for the file manager. This registration process saves the device driver name and file operations in either the `chrdevs[]`or `blkdevs[]` kernel table (depending whether the device is a character or block device), using the major number as the table index. A temporary major number is assigned to the driver if it is registered with a major number of 0.

6.2 Handling Interrupts

Section 4.1 provides a general description of how IRQs and ISRs are used to respond to the occurrence of an interrupt. The ISR is associated with a particular IRQ using an explicit kernel function, `request_irq()`. This function has arguments to specify the IRQ, whether the ISR is for a fast or slow interrupt, the entry point of the ISR, and an argument of type `void *` that can be used to pass arbitrary information (specified as an argument to `request_irq`). If the IRQ already has an ISR associated with it, the request fails.

There are a limited number of IRQ identifiers in a computer – the number being determined by the hardware. If two different devices must use the same IRQ, a master ISR must demultiplex IRQs to a collection of ISRs. It does this by polling the hardware to determine which device actually caused the IRQ, then it selects the proper ISR from the collection.

The ISR can be constructed to perform any work that is needed to complete the I/O operation after the devices finishes. For example, it might check the device status to ensure that the operation completed successfully. If the operation was a read operation, it might also transfer data from the device buffer to primary memory.

Recall that ISRs are expected to run quickly, even if they do not have interrupts disabled. In cases where one instance of an ISR might get interrupted by a second instance of the same ISR (such as multiple clock ticks that might occur while some series of ISRs are being executed), parts of the ISR work can be relegated to a bottom half. The ISR can then mark the bottom half for later execution, effectively postponing some of its processing. When `ret_from_sys_call` runs, it will detect the marked bottom half and run it before calling the scheduler.

A maximum of 32 different bottom halves can be marked for execution. If there is still more postponed processing required (and no available bottom half marker slot), the work can be enqueued in a task queue for subsequent execution. A *task queue* is a list of functions with the prototype

```
void (*routine)(void *);
```

that will be executed, one after the other, after the bottom halves have all been run. The task queue functionality distinguishes among functions that are scheduled with interrupts disabled, from within an ISR or bottom half, or under any other circumstance.

7 File Management

The Linux file manager defines a single internal (application software) view of files that can be used to read and modify files written onto a storage device using a variety of file managers. For example a disk that contains files written using MS-DOS (or a Windows OS with DOS compliant format) can be read or rewritten by a Linux application using the Linux file manager. The internal Linux view of a file is that it is a named byte stream that can be saved in the secondary storage system. The byte stream is fragmented into a set of blocks, then the blocks are stored on the secondary storage device according to a strategy chosen by a particular type of file system. The file manager fetches and puts the blocks that contain the portions of the byte stream that an application program writes or reads. That is, the file manager is responsible for determining which disk blocks should be read or written whenever any part of the byte stream is read or written.

The Linux file manager is designed so that application programs use a fixed set of functions to manipulate files – `open()`, `close()`, `lseek()`, `read()`, `write()`, `ioctl()`, and so on, as specified by the POSIX interface. A file system independent part of the file manager handles generic aspects of the work such as checking access rights, determining when disk blocks need to be read or written, and so on. There is another part of the file manager that handles all file system dependent aspects of the job, such as determining where blocks are located on the disk, directing the device driver to read or write specific blocks, and so on. The two combined parts enable Linux to provide a fixed set of operations at the API, yet to handle files on disks even if the files were written using a Windows OS, MINIX, or some other OS.

The Linux file manager API is built on an abstract file model that is exported by the *Virtual File System* (VFS). The VFS implements the file system independent operations. OS designers provide extensions to the VFS to implement all required file system dependent operations; the Version 2.x release can read and write disk files that conform to the MS-DOS format, the MINIX format, the `/proc` format, a Linux-specific format called *Ext2*, and others. Of course this means that specific file system dependent components are included in Version 2.2.12 to translate VFS operations and formats to/from each of the external formats that are supported. This discussion focuses on the VFS model. In Exercises 11 and 12 you will create your own DOS-like file system to use with VFS.

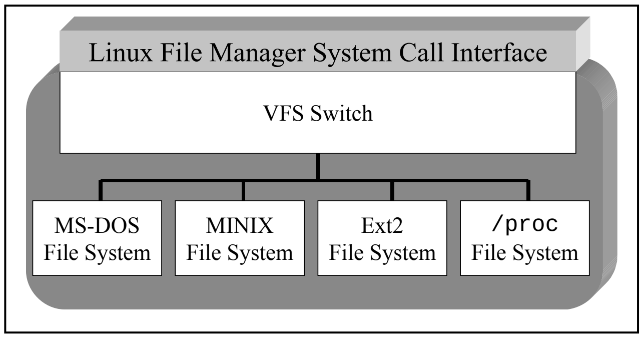


Figure 14: The Virtual File System Switch

The heart of VFS is the *switch*, which provides the canonical file management API to user space programs, and which also establishes an internal interface used by the different file system translators that support MS-DOS files, MINIX files, Ext2 files, and so on (see Figure 14). A new file system can be supported by implementing a new file system dependent (“translator”) component. Each such translator provides functions that the VFS switch can call (when it gets called by a user program), and which can translate the external file representation into the internal one. Thus the translator is responsible for determining the strategy used to organize disk blocks on the disk device, for reading and writing disk properties, for reading and writing external file descriptor information, and for reading and writing the disk blocks that contain file data.

The VFS file system model is patterned after conventional UNIX file systems. A VFS file descriptor is called an *inode* (though it has its own unique format to support the multiple file system approach). While VFS will contain its own format for a file descriptor, each file system dependent translator converts the contents of the external descriptor into the VFS inode format when the file is opened. VFS then operates on its own inode data structure. Conversely, when the file is closed, the contents of the internal inode are used to update the external file descriptor.

The VFS inode contains the file access rights, owner, group, size, creation time, last access time, last time the file was written, and so on. The inode also reserves space for the pointer mechanism that the specific file system uses to organize the disk blocks, even though VFS does not know how these pointers will be organized (that information is encapsulated in the file system specific translator component). VFS also supports directories, so it presumes that the external file directories contain at least the name of each file stored in the directory, and the address of its file descriptor (almost all file systems contain this minimum amount of information).

VFS assumes a minimum structure on the disk organizations:

- The first sector on the disk is a *boot block* used to store a bootstrap program (recall the earlier discussion of the bootstrap process in Section 3.5). The file system does not really use the boot block, but it does presume that it is present on every file system.
- There is a *superblock*, containing disk-specific information such as the number of bytes in a disk block.
- There are external file descriptors on the disk that describe the characteristics of each file.
- There are data blocks that are linked into each file to contain the data.

Before VFS can manage a particular file system type, a translator for that type must be written, then it must be registered with VFS. The VFS `register_filesystem()` function informs VFS of basic information it will need including the name of the file system type and the entry point of a file system's `read_super()` function that will be used when the file system is mounted.

7.1 Mounting the File System

Computers with storage devices that have removable media (such as tape drives and floppy disk drives) must be able to change the system's file structure each time a medium is placed in, or removed from, the device. VFS uses the conventional UNIX mechanism to allow such file systems to be combined into the system's directory hierarchy (but of course Linux allows heterogeneous file systems to be combined whereas conventional UNIX only combines homogeneous file systems). The `mount()` command appends a new file system into an existing directory hierarchy. It does this by replacing a directory in the previously mounted file system by the root of the new file system when the corresponding replaceable medium is placed on the system.

For example, suppose a system contains an optical disk reader. When a particular optical disk is placed in the reader, the `mount` operation requests that the file manager graft the directory subtree of the optical disk onto an existing file system tree. The device root is treated as a directory in the existing file system hierarchy (see Figure 15). In the example, the file named "`c`" has an absolute path name of `/a/c` since directories "`a`" and "`b`" are combined as the mount point. After the file system has been mounted, it can be accessed through all normal directory operations, including relative and absolute path names and directory traversal operations. When the file system is to be removed, an `unmount()` command is sent to the file manager. This command prevents access attempts to a part of the file hierarchy no longer existing.

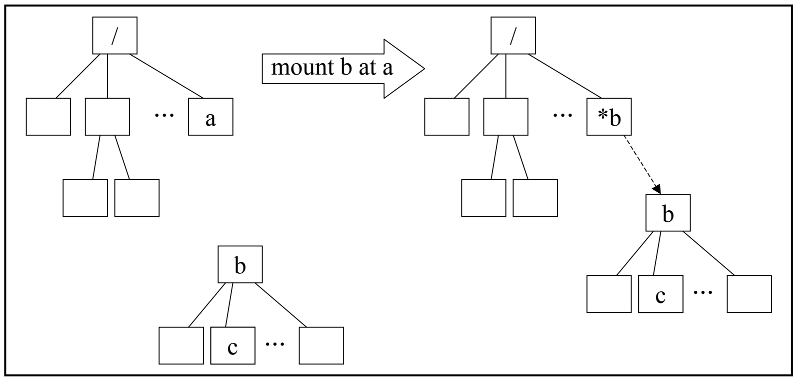


Figure 15: The UNIX mount Command

When a file system is mounted, VFS creates an instance of the `struct super_block` data structure to hold information it needs to administer the new file system. VFS then calls the new file systems's `read_super()` to retrieve the information contained in the file system's analog of the superblock (from the storage device) and to translate this information and save it in the `struct super_block` data structure. This superblock includes various fields that VFS needs to administer the file system, particularly the field:

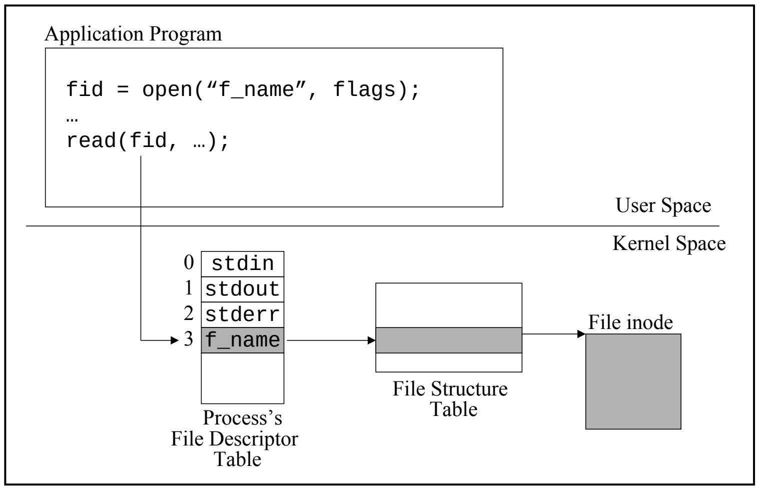
```
struct super_operations *s_op;
```

This `s_op` field specifies a collection of functions to write the superblock back to the disk, to read and write inodes on the disk, to check the superblock status, and so on. Thus, `read_super()` is the "bootstrap" function that is bound to VFS when the file system type is registered. After VFS reads the superblock at mount time, the VFS can use the `super_operations` to manipulate the on-disk superblock as needed. Another way of thinking about the `super_operations` is that they define a public interface for a private on-disk superblock abstract data type, thus making it possible for the translator to "implement" superblocks and inodes as it chooses.

7.2 Opening a File

When a file is opened, the file manager searches the storage system for the specified path name. Searching can be an extended procedure, since it is necessary to open each directory in the path name (starting at the highest-named directory in the path), to search the path for the next file or directory in the path name, to open that directory or file, and so on. If the search encounters a mount point, then it moves from one file system to another and continues the search. This means that the search may begin in, say, an Ext2 file system, but ultimately find the file in an MS-DOS file system. Once the file has been found in a directory VFS checks the file and user permissions to ensure that the user's process has permission to open the file. If the process has the correct permissions, VFS will set up various table entries to manage the I/O. First, an entry is created in the process's *file descriptor table* (see Figure 16). There is one file descriptor table for each process, and each open file has an entry in the table. An entry is identified by a small integer value returned by the call; this identifier is used for all subsequent references to the file. Note that when the process is created the file descriptor table is created with three entries: `stdin` has an entry identifier of 0, `stdout` has a value of 1, and `stderr` has a value of 2. The next successful `open` or `pipe` call will create an entry in the file descriptor table at location 3.

The entry in the file descriptor table points to an entry in the open file table called a *file structure* (it has type `struct file`). The file structure entry keeps status information specific to the process that opened the file for the process. For example, the value of the file position for this process's use is in the file structure entry; if two different processes have the file open, then each will have its own file structure entry, and hence its own copy of the file position. The file structure entry references the VFS inode after it has



been constructed into primary memory.

Figure 16: Kernel File Tables

The in-memory inode is created by the VFS by reading the external file descriptor (using the `read_inode()` function specified in the superblock `s_op` list) and translating it to the VFS inode format. Just as the superblock provides a field for a list of `super_operations`, the inode provides a field to define a list of inode operations

```
struct inode_operations *i_op;
```

The `struct inode_operations` are a list of operations that VFS will need to manipulate blocks that make up the file, for example `create()`, `lookup()`, `link()`, and `mkdir()`. There is also a list of default operations that are used if the `i_op` list does not provide an operation.

When inodes are allocated in the VFS, they are allocated to a single, large linked list. This is useful whenever the VFS intends to scan all inodes. However, access is slow when the VFS wants to reference a particular inode. Therefore, inode references are also entered into an open hash table, where they can be accessed with a few (possibly a single) table probe.

Changes to the in-memory version of the inode are not propagated to the external file descriptor on the storage device the moment the file manager changes them. Instead, the in-memory version of the inode is used to update the external file descriptor in the secondary memory periodically, when the file is closed, or when the application issues a `sync` command. If the machine halts while a file is open, changes to the in-memory inode are likely to be different than the contents of the external file descriptor on the disk if there have been any recent changes to the inode. The result can be an inconsistent file system, since the most recent information about the file is lost when the in-memory inode is destroyed. For example, if block pointers in the in-memory inode have changed (with corresponding changes to disk blocks), then the disk may have an external file descriptor that is inconsistent with pointers in various storage blocks on the disk.

7.3 Reading and Writing the File

An instance of `struct file` (entry in the file structure table) is created when the file is opened. There is a field in the `struct file` for

```
   struct file_operations *f_ops;
```

that specifies the entry points for `read()`, `write()`, `lseek()`, `select()`, and so on, that are used to perform the file system specific I/O operations. There is also a set of default `file_operations` specified in the inode, so that if any particular file operation (like `lseek()`) is not specified in the `f_ops` list, the function from the default list will be used.

The essential purpose of these routines is to move information back and forth between user space addresses and the secondary storage blocks. The `read()` function invokes the file system specific operations to transfer data from the disk to a system buffer, then to copy the requested number of bytes from the system buffer to the user space address. The `write()` function moves data from a user space address into a system buffer where it will be written to the disk. These routines must also marshall and unmarshall the byte stream.

Files are organized as sequential byte streams. One implication of this is that applications normally read or write the file as a sequence of bytes rather than randomly accessing characters in the file. As a consequence, buffering can be used to substantially increase the performance of the system. In read buffering, the file system reads ahead on the byte stream, fetching disk blocks that hold parts of the byte stream that appear after the byte currently addressed by the file pointer. In write buffering, when a disk block becomes full, it is staged for writing whenever the disk device becomes available. The result is that the application need not wait for incoming buffers since they will have already been read when they are referenced, and it need not wait for write operations since they will take place as a background activity after the application has logically written the bytes to the stream.

Linux attempts to use as much of the primary memory as it can to provide input and output buffers. It creates a dynamic *buffer cache* whose size depends on the number of page frames that are currently required to support virtual memory. Pages that are not needed by the virtual memory system can be used in the buffer cache. When a page fault occurs, the buffer cache is one of the first places that the virtual memory system will search for page frames.

Block buffering is a relatively complex task. This part of the file manager must satisfy each of the following constraints:

- Each file open for reading should have one or more buffers containing information that has been read from the disk before the process has actually requested the information.
- Each file open for writing has a set of buffers containing information that is to be written to the device whenever the device is available.
- Output full blocks that are the same as blocks on the storage device should not be written.
- When the virtual memory does not need all of physical memory, the buffer cache should be made larger; when the virtual memory needs more physical memory, the buffer cache size should be reduced (without effecting ongoing I/O).
- Buffers can be of varying size.
- Buffers can be shared among processes.

7.4 The Ext2 File System

The Ext2 file system is patterned after the BSD Fast File System [McKusick, et al., 1996]. Specifically, the disk is formatted to contain a set of *block groups*, each of which contains a superblock, group descriptor information, a table of inodes for the files in the block group, and the data blocks. The superblock contains the following information:

- Number of inodes
- Number of free inodes
- Block details
- Allocation limits
- Last mount and write times
- Errors and status
- Other housekeeping information

A block descriptor is used to keep the details of the blocks within a particular block group. A block can contain up to 8192 blocks for disk block sizes of 1024 bytes (each block is of size 1024 = `EXT2_MIN_BLOCK_SIZE` < `EXT2_BLOCK_SIZE` < `EXT2_MAX_BLOCK_SIZE` = 4096 bytes). The block descriptor also includes the set of inodes for the files in the block group.

The inode uses a variant of an indexed allocation scheme for organizing blocks. It contains pointers to 15 different storage blocks (see Figure 17). The first 12 blocks of the file are indexed directly from the first 12 of 15 pointers in the inode. The last three pointers are used for indirect pointers through *index blocks*. If the file manager is configured with the maximum size 4KB blocks, the 12 direct pointers in the inode accommodate files up to 48KB. If a file requires more than 12 blocks, the file system allocates an index block and links it into the *single indirect* (thirteenth) pointer of the inode. Hence, blocks 13 to k are indirectly addressed from the inode via the indirect block identified by the thirteenth pointer in the inode. Similarly, larger files use the fourteenth pointer to address a *double indirect* block, and the largest files use the fifteenth pointer to point at a *triple indirect* block.

How big can UNIX files be? This depends on the size of the blocks and the size of the disk addresses used in the system. To simplify the arithmetic, suppose an indirect block can store 1,000 disk addresses (ordinarily, the number of disk addresses would be a power of two such as 1,024). Then the single indirect block will provide pointers to an additional 1,000 disk blocks. Blocks 0 through 11 are accessed via the direct pointers in the inode, but blocks 12 through 1,011 are accessed indirectly through the single indirect block. The fourteenth block pointer in the inode is the double indirect pointer. It points to a block that contains pointers to blocks of the type referenced from the single indirect field. The double indirect pointer addresses a block pointing to 1,000 indirect blocks, so blocks 1,012 to 1,001,011 are accessed through the double indirect list. The fifteenth block pointer is the triple indirect pointer. It points to a block that contains double indirect pointers. Again, if each block can store 1,000 block addresses, then the triple indirect pointer indirectly addresses blocks 1,001,012 to the maximum-sized file (under these assumptions) of 1,001,001,011 blocks.

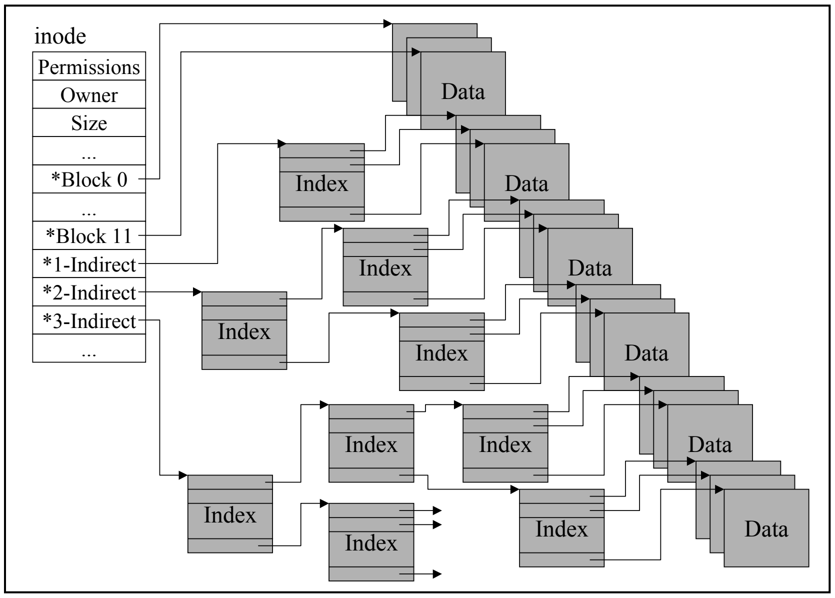


Figure 17: Ext2 File Structure

With this block allocation strategy, very large files are possible, even though as files grow larger, the access times are greater due to the indirection. There are other considerations (designed into the inode structure) that prevent a file from reaching this maximum size. For example, with the block sizes given previously, a file using the triple indirect index would require that there be a device capable of storing 4000 GB. Current versions of BSD UNIX do not use the triple indirect pointer, partly because of incompatibility of file sizes with storage device technology and partly because the 32-bit file offsets used in the file system preclude file sizes larger than 4GB.

8 Learning More About Linux

Linux continues to grow in popularity, so there are many excellent sources of information to learn more details. The Exercises in the next part of the book focus on details for learning about certain aspects of Linux, but the areas not covered by the Exercises are discussed in these other sources.

The Linux Documentation Project (`http://metalab.unc.edu/mdw/Linux.html`) is an excellent on-line collection of articles about all aspects of Linux. The second edition of the *Linux Kernel Internals* book by [Beck, et al., 1998] is also an excellent reference for describing the organization of the Version 2.0 kernel, and for explaining many of the details of how individual parts of the kernel are designed.

Vahalia's book on UNIX internals [Vahalia, 2000], Bach's book on UNIX [Bach, 1986] System V internals, and McKusick, et al.'s book [McKusick, et al., 1996] on BSD 4.4 internals describe how traditional UNIX systems are built, and are excellent references for understanding Linux.

In the end, people who study Linux internals must read the source code. There is a very complete source code browser site at `http://lxr.linux.no/source/`. It is expected that you will have a copy of the source code on your laboratory machine. It may be loaded anyplace in the file system hierarchy, but the normal place it is loaded is at the path `/usr/src/linux`, that is, `linux` is the root directory of the subtree that contains all the source code. Exercise 3 includes a discussion of the organization of the source code; this book will frequently refer to source code files by assuming that the current directory is the root directory of the source subtree (for example `/usr/src/linux`).

PART 2: EXERCISES

This manual's are intended to complement the materials that you learn in your undergraduate OS course. In the lecture part of your course, the focus is on concepts and issues, with side discussions on Linux, UNIX, and/or Windows NT. This manual's goal is to put those theories into practice with a series of hands-on exercises using the Linux kernel.

The exercises use the public distribution of the Linux source code, Version 2.2.12. By the time you have completed these exercises, you will have studied and experimented with considerable OS software, and will have gained a deep insight into how Linux and other modern operating systems are designed.

The exercises address almost all the aspects of the OS. Following is a summary of what you will do in each. Here is a link to the commercial web site that contains the Exercises.

1 Observing Linux Behavior

Learn about the kernel by having you use the `/proc` file system interface to read the values of various kernel variables. You will gain an intuitive understanding of kernel behavior, and learn a valuable tool for inspecting the kernel.

2 Shell Program

Gain experience with programming applications that consist of multiple processes. While not focusing on kernel internals, it does provide prerequisite experience with concurrency and file descriptors.

3 Kernel Timers

Your first opportunity to inspect the Linux kernel source code. You will learn how interval timers are implemented in the kernel, and then use them to profile a user program.

4 Kernel Modules

Write your first software which will execute in supervisor mode as part of the kernel. Your solution code will use the loadable kernel module feature first to read the kernel's time variable and then to report it using the `/proc` file system interface.

5 System Calls

Add a new system call to the existing Linux kernel. The system function that you implement is simple, but you will learn the details of how the system call mechanism works, and then use it to change the system call interface to your copy of Linux.

6 Shared Memory

Modify the existing implementation of the System V shared memory facility. The modification is not difficult, but it will give you the opportunity to study how this functionality is designed and implemented.

7 Virtual Memory

Learn how Linux implements virtual memory, and then instrument the existing code to provide reports on the virtual memory system's performance.

8 Synchronization Mechanisms

Learn about a new synchronization mechanism to the kernel, called an event. Events are shared among processes. One process creates an event, and then a community of processes share the event, much like a barrier.

9 The Scheduler

Modify the Linux scheduler so that it uses an alternative policy. You will learn the details of how the scheduler works and then how to change it.

10 Device Drivers

Write your first device driver to implement a FIFO (or pipe) among processes. You will learn about the issues for writing drivers without manipulating hardware.

11 File Systems

Implement a file directory operations on a raw floppy disk driver.

12 File I/O

Complete the MS-DOS-compatible file system started in Exercise 11 by providing the functions for reading and writing the data.

Three factors make kernel programming challenging:

- The lack of good documentation
- The need to be when you change the kernel
- The absence of a robust and safe debugging environment.

There is no way around these problems except to "bite the bullet" and dive into the project.

As you work through these exercises, notice that to arrive at a solution you generally will not have to write much software. However you will have to *read* a lot of softrware in order to come up with a working solution. In addition, you will quickly observe that ou need to have a good working knowledge of C or C++. The kernel is written in C and assembly language (though you will rarely encounter the assembly language, and can solve all these exercises without reading it). The discussion includes many code fragments, most taken verbatim from the Linux source code. Also as part of my pedagogical style, I have included some of my own comments into the code fragments. Each is written with the C++ comment style ("`// Here is a pedagogical comment`") rather than the C comment style ("`/* Here is a verbatim comment */`").

The exercises increase in difficulty from 1 through 12. Considerable assistance is offered in solving the early exercises and much less with the later ones. In this sense, you should solve the last exercise by solving all of those that precede it. Realistically, that is too many exercises to do in one term. Figure 18 shows the specific dependencies among the exercises. The width of the arrow is intended to suggest the true strength of the dependency. For example, whereas you'll find it helpful to do Exercise 1 before doing Exercise 4, you really should do Exercise 11 before doing Exercise 12. If you are working on an exercise and are stuck, you might browse through earlier exercises to see if there is something that can help you solve your problem.

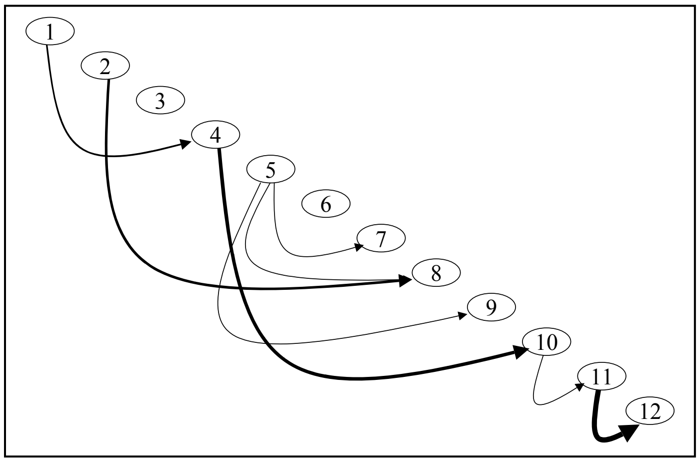


Figure 18: Exercise Dependencies

Further Study

The exercises in this book teach you how the Linux kernel is designed and organized. Each exercise focused on one or a few aspects of the kernel and then provided you with an exercise to experiment with this aspect of the kernel.

All parts of ther kernel were at least touched on, but some were not not discussed very deeply and might be topics for further study on your part, including the following:

- The virtual memory system is pervasive in the kernel data structures. Because the Linux virtual memory uses dynamic memory allocation, added complexity exists in understanding how and when page frames are taken away from one process (or buffer pool) and added to another. The discussion of virtual memory in this lab manual barely scratched the surface of the virtual memory system.
- Most of the code in a kernel is for the file system. The Linux file system is certainly one of the larger and more complex parts of the kernel. Its increased complexity results from the need to handle buffering. Many details of the file system were omitted, so a next phase of study should focus more on the existing file systems.

You can continue your study of kernels ni any of several ways. This manual's goal has been to provide you with enough background and experience so that you can read kernel and modify kernel code. By necessity, the exercises have to be bounded to small problems so that they are good learning tools. After solving the exercises in this manual you will be prepared to tackle most kernel problems (though some situations will require considerable additional study of the code). Here are a few suggestions for further study:

- Redesign and reimplement any module of the kernel.
- Extend the functionality; for example write your own implementation of threads.
- Write a new file system that is used with VFS.
- Work on another kernel.
- Write a new kernel.
- Start a company …

References

1. Bach, Maurice J., *The Design of the Unix Operating System*, Prentice Hall International, London, 1986.
2. Beck, Michael, Harald Bőhme, Mirko Dziadzka, Ulrich Kunitz, Robert Magnus, and Dirk Verworner, *LINUX Kernel Internels*, Second Edition, Addison-Wesley, Reading, MA, 1998.
3. Deitel, Harvey M., *An Introduction to Operating Systems*, Second Edition, Addison-Wesley, Reading, MA, 1990.
4. Johnson, M. K., *Linux Kernel Hacker's Guide*, Version 0.7, Linux Document Project, `http://metalab.unc.com/mdw/Linux.html`.
5. Linux web pages at `http://metalab.unc.com/mdw/Linux.html.`
6. McKusick, Marshall Kirk, Keith Bostic, Michael J. Karels, and John S. Quarterman*, The Design and Implementation of the 4.4 BSD Operating System*, Addison-Wesley, Reading, MA, 1996.
7. Messmer, Hans-Peter, *The Indispensable PC Hardware Book*, Second Edition, Addison-Wesley, Reading MA, 1995.
8. Nutt, Gary J., *Operating System Projects for Windows NT*, Addison-Wesley, Reading, MA, 1999.
9. Nutt, Gary J., *Operating Systems*, Third Edition, Addison-Wesley, Reading, MA, 2004.
10. Ousterhout, John K., Hervé Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson, "A Trace-Driven Analysis of the UNIX 4.2 BSD File System," *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, ACM, 1985, 15–24.
11. Pomerantz, Ori. *The Linux Kernel Module Programming Guide*, Version 1.0, 1999, Linux Documentation Project, `http://metalab.unc.com/mdw/Linux.html`.
12. Ritchie, Dennis and Ken Thompson, "The UNIX Time-Sharing System," *Communications of the ACM*, 17, 7 (July 1974), 1897–1920.
13. Rusling, David A. *The Linux Kernel*, Version 0.8-3, 1996-99, Linux Documentation Project, `http://metalab.unc.com/mdw/Linux.html`.
14. Silberschatz, Abraham and Peter Baer Galvin, *Operating Systems Concepts*, Fifth Edition, Addison-Wesley, 1998.
15. Solomon, David A., *Inside Windows NT*, Second Edition, Microsoft Press, Redmond, WA, 1998.
16. Stallings, William, *Operating Systems*, Second Edition, Prentice Hall, Englewood Cliffs, NJ, 1995.
17. Stevens, W. Richard, *Advanced Programming in the UNIX Environment*, Addison-Wesley, Reading, MA, 1993.
18. Tanenbaum, Andrew S., *MINIX version*, Prentice Hall, Englewood Cliffs, NJ, 1987.
19. Tanenbaum, Andrew S., *Modern Operating Systems*, Prentice Hall, Englewood Cliffs, NJ, 1992.
20. Tanenbaum, Andrew S., *Distributed Operating Systems*, Prentice Hall, Englewood Cliffs, NJ, 1995.
21. Vahalia, Uresh, *UNIX Internals: The New Frontiers*, Second Edition, Prentice Hall, Englewood Cliffs, NJ, 2000.
22. Welsh, Matt, "Implementing Loadable Kernel Modules for Linux," *Dr. Dobb's Journal*, May, 1995.